WEUNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

DESENVOLVIMENTO DE UM MOTOR DE JOGOS 3D, UTILIZANDO WEBGL

DANIEL PEREIRA

BLUMENAU
2012

2012/2-07

DANIEL PEREIRA

DESENVOLVIMENTO DE UM MOTOR DE JOGOS 3D, UTILIZANDO WEBGL

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciência da Computação — Bacharelado.

Prof. Dalton Solano dos Reis , M.Sc. - Orientador

BLUMENAU
2012

2012/2-07

DESENVOLVIMENTO DE UM MOTOR DE JOGOS 3D, UTILIZANDO WEBGL

Por

DANIEL PEREIRA

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. Dalton Solano dos Reis, M.Sc. – Orientador, FURB
Membro:	Prof. Nome do professor, Titulação – FURB
Membro:	Prof. Nome do professor, Titulação – FURB

Blumenau, 10 de dezembro de 2012

Dedico este trabalho a minha mãe, Nilva Schmitz, por todo apoio, amor, força e auxílio que me foram dedicados.

AGRADECIMENTOS

À minha mãe, Nilva, acima de tudo e todos. Jamais poderei ser grato o suficiente pelas coisas que ela fez em meu favor, sendo obrigada muitas vezes a abrir mão de sonhos que carregou durante a vida toda.

Aos meus tios, José e Alice, por todo o suporte concedido desde minha vinda para Blumenau. A difícil tarefa de morar sozinho, acompanhada do primeiro emprego, teria sido muito mais difícil não fosse pela ajuda de ambos.

Ao meu orientador, Dalton Solano dos Reis, por ter acreditado na conclusão deste trabalho e por estar sempre disponível quando necessário.

Remembering that you are going to die is the best way I know to avoid the trap of thinking you have something to lose. You are already naked, there is no reason not to follow your heart.

Steve Jobs

RESUMO

Este trabalho apresenta a implementação de um motor de jogos desenvolvimento em WebGL e também uma aplicação que o utiliza. O motor de jogos disponibiliza funcionalidades como grafo de cena, gerenciamento de objetos da cena, texturas e iluminação. Para o desenho da cena e dos objetos é utilizada a linguagem JavaScript, utilizando elementos disponíveis na linguagem HTML5. O trabalho aborda as técnicas utilizadas para abstração da linguagem GSGL e da API do WebGL e apresenta o resultado obtido com os testes de desempenho realizados através de um dos navegadores web suportados.

Palavras-chave: Motor de jogos. WebGL. HTML5.

ABSTRACT

This work describes the implementation of a game engine developed on WebGL, along with an application that uses it. There are features available in the game engine such as scene graph, scene object management, textures and lighting. JavaScrit is the language used to render the objects on the screen, using elements available on the HTML5 language. The work addresses the used techniques to abstract the GLSL language and the WebGL API. It also shows the results obtained through tests executed on the supported browsers.

Key-words: Game engine. WebGL. HTML5.

LISTA DE ILUSTRAÇÕES

Figura 1 – *Pipeline* utilizado na WebGL.......17
Figura 2 - Diferença entre os dados trabalhados pelo *vertex shader* e pelo *fragment shader*..18
Figura 3 - Arquitetura de um motor de jogos.......20
Figura 4 - Exemplo de grafo de cena.......21
Figura 5 - Diferença entre o modelo de luz global e local.......22
Figura 6 - Diferença reflexão conforme o ângulo que o raio da luz difusa atinge.......23
Figura 7 - Geometria usada para a a reflexão especular.......23
Figura 8 - Ambiente visualizado por meio da CopperLicht.......25
Figura 9 - Exemplo exibindo o estado atual da conversão do CubicVR para WebGL.......26
Figura 10 - Exemplo de aplicação que utiliza a biblioteca Three.js.......26
Figura 11 - Diagrama de casos de uso.......29
Quadro 1 - Caso de uso UC01.......29
Quadro 2 - Caso de uso UC02.......30
Quadro 3 - Caso de uso UC03.......30
Quadro 4 - Caso de uso UC04.......31
Quadro 5 - Caso de uso UC05.......31
Figura 12 - Diagrama de pacotes do motor de jogos.......32
Figura 13 - Diagrama de classes do pacote `br.furb.fwgl.core`.......33
Figura 14 - Diagrama de classes do pacote br.furb.fwgl.objects.......34
Figura 15 - Diagrama de classes do pacote `br.furb.fwgl.io`.......35
Figura 16 - Diagrama de sequência da ação de desenhar um cubo em uma cena.......37
Quadro 6 - *vertex shader* utilizado no motor de jogos.......40
Quadro 7 - *fragment shader* utilizado no motor de jogos.......41
Quadro 8 - Código-fonte do método initShaders.......42
Quadro 9 - Método `draw` da classe `Light`.......43
Quadro 10 - Método `draw` da classe `Camera`.......43
Quadro 11 - Método `draw` da classe `Cube`.......45
Quadro 12 - Arquivo configurado para iniciar o desenvolvimento com o motor de jogos.......48
Quadro 13 - Criação dos objetos da câmera e luz e adição deles à cena.......48
Quadro 14 - Criação do cubo e textura.......49

Quadro 15 - Obtenção do elemento *canvas* e desenho da cena na tela 49
Quadro 16 - Importação do XML através da classe XMLReader. 50
Quadro 17 - Representação de uma cena em um arquivo XML 50
Figura 17 - Desenho exibido na tela após o código finalizado... 51
Figura 18 - Visualizador de grafo de cena carregado 52
Figura 19 - Problema que ocorre nas texturas ao abrir a tela pela primeira vez 54
Figura 20 – Cenário utilizado nos testes de desempenho 55
Figura 21 - Gráfico medindo o número de quadros por segundo obtidos pela aplicação 56
Figura 22 - Relatório da execução do *profile* da aplicação 56
Figura 23 - Dados comparativos entre o resultado dos três navegadores testados 57
Quadro 18 - Código fonte do modelo simples de herança 64

LISTA DE TABELAS

Tabela 1 - Número de quadros por segundo por número de objetos na cena 55
Tabela 2 - Número de quadros por segundo para os três navegadores testados 57

LISTA DE SIGLAS

3D – 3 Dimensões

API – *Application Programming Interface*

CSS – *Cascading Style Sheets*

DOM – *Document Object Model*

FWGL – Furb WebGL

GLSL - *openGL Shading Language*

HTML – *Hyper Text Markup Language*

RGB – *Red Green Blue*

SGI – *Silicon Graphics International*

SVG – *Scalabe Vector Graphics*

UML – *Unified Modeling Language*

V-ART – *Virtual aRTiculations for virtual reality*

WTP – *Web Tools Platform*

WWW – *World Wide Web*

XML – *eXtensive Markup Language*

SUMÁRIO

1 INTRODUÇÃO 12
1.1 OBJETIVOS DO TRABALHO 13
1.2 ESTRUTURA DO TRABALHO 13
2 FUNDAMENTAÇÃO TEÓRICA 14
2.1 HTML5 14
2.2 JAVASCRIPT 14
2.3 OPENGL ES 2.0 15
2.4 WEBGL 16
2.4.1 *Pipeline* gráfico 16
2.5 MOTOR DE JOGOS 19
2.5.1 Grafo de cena 20
2.5.2 Iluminação 21
2.6 V-ART 24
2.7 TRABALHOS CORRELATOS 25
3 DESENVOLVIMENTO 28
3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO 28
3.2 ESPECIFICAÇÃO 28
3.2.1 Casos de uso 29
3.2.2 Diagrama de classes 31
3.2.3 Diagrama de sequência 36
3.3 IMPLEMENTAÇÃO 37
3.3.1 Técnicas e ferramentas utilizadas 37
3.3.2 O motor de jogos 38
3.3.3 Operacionalidade da aplicação 46
3.3.4 Visualizador de grafo de cena 51
3.4 RESULTADOS E DISCUSSÃO 53
3.4.1 Testes de desempenho 54
3.4.2 Comparativo entre o trabalho desenvolvido e os trabalhos correlatos 58
4 CONCLUSÕES 59
4.1 EXTENSÕES 59

1 INTRODUÇÃO

A internet vem experimentando uma alta taxa de crescimento. Inicialmente reservada apenas para uso militar, do governo e das universidades, esta rede internacional de computadores agora age como um meio de comunicação para empresas privadas e uso pessoal. É acessada por centenas de milhões de pessoas mundo afora, sendo que a World-Wide Web (WWW) é o tipo de serviço que mais cresce dentro dela (LOMBARDI, 2011). Segundo Castro (2003), "A World-Wide Web (também chamada Web ou WWW) é, em termos gerais, a interface gráfica da Internet. Ela é um sistema de informações organizado de maneira a englobar todos os outros sistemas de informação disponíveis na Internet".

Os navegadores web talvez sejam um dos tipos de aplicativo mais utilizados na história. Eles têm evoluído significativamente nos últimos quinze anos e, atualmente, navegadores web são executados em diversos tipos de hardware, desde celulares e tablets até computadores pessoais. Os navegadores utilizam da *Hyper Text Markup Language* (HTML) para exibir os dados e da linguagem JavaScript para tornar as páginas dinâmicas ou para criar aplicativos para web (GROSSKURTH; GODFREY, 2006).

Entre os aplicativos possíveis de serem criados para web, uma área bastante promissora é a de jogos em três dimensões (3D). Segundo Prada (2010), "o 3D é o futuro, não somente dos jogos como do conteúdo multimídia na web. Não é difícil deduzir que os *games*, que já fazem amplo uso das três dimensões nos consoles, também incorporem a profundidade quando executados direto do navegador".

Um dos meios possíveis de se desenvolver jogos para web é utilizando WebGL. WebGL é uma biblioteca multiplataforma executada em conjunto com a versão 5 da HTML e é baseada na especificação 2.0 da Open Graphics Library for Embedded Systems (OpenGL ES) (KHRONOS, 2011b). Segundo Prada (2010), "as versões Beta do novo Firefox 4.0, assim como os desenvolvimentos mais recentes do Safari e do Google Chrome, já possui suporte para tal recurso[..]. Os resultados são bastante promissores e já podem ser observados em alguns sites."

Dentro do exposto, é demonstrado o desenvolvimento de um motor de jogos 3D, que utiliza a biblioteca WebGL. O trabalho também demonstra um visualizador de grafo de cena utilizando o motor de jogos desenvolvido, exibindo os resultados obtidos com o desenvolvimento deste trabalho.

1.1 OBJETIVOS DO TRABALHO

Este trabalho tem como objetivo desenvolver um protótipo de motor de jogos 3D, utilizando a biblioteca WebGL.

Os objetivos específicos do trabalho são:

a) disponibilizar o gerenciador de objetos;
b) disponibilizar gerenciador de grafo de cena;
c) disponibilizar leitura de objetos oriundos de um arquivo XML que siga o padrão estabelecido;
d) disponibilizar a interação do usuário para movimentação de câmera no ambiente de jogo mantido pelo motor de jogos;
e) disponibilizar a implementação de iluminação do ambiente;
f) disponibilizar uma aplicação exemplo utilizando o motor construído;
g) disponibilizar suporte a texturas.

1.2 ESTRUTURA DO TRABALHO

A estrutura deste trabalho está divido em quatro capítulos, sendo que o primeiro capítulo apresenta uma introdução ao tema abordado, os objetivos e estrutura deste trabalho. O segundo capítulo contém a fundamentação teórica utilizada no decorrer do trabalho. Ao fim do capítulo também é feita a descrição de alguns trabalhos correlatos referentes a este tema.

O terceiro capítulo trata sobre o desenvolvimento deste trabalho. Nele são expostos os requisitos a serem atendidos, a especificação do trabalho e as técnicas e ferramentas utilizadas. Também são exibidos trechos de código-fonte necessários para um melhor entendimento do trabalho.

Por fim, o quarto capítulo aborda as principais conclusões e contribuições, assim como traz uma lista de possíveis trabalhos de extensão.

2 FUNDAMENTAÇÃO TEÓRICA

A seção 2.1 apresenta o HTML5 e algumas de suas novidades. Na seção 2.2 é apresentada a linguagem de programação JavaScript. A seção 2.3 descreve o OpenGL ES. Na seção 2.4 é detalhada a WebGL e seu funcionamento. A seção 2.5 apresenta o conceito de motor de jogos. A seção 2.6 descreve a biblioteca V-ART. Por final, na seção 2.7, são apresentados os trabalhos correlatos.

2.1 HTML5

A *Hiper Text Markup Language* (HTML) é a linguagem de marcação utilizada para a criação de páginas Web. A HTML foi primeiramente projetada para descrever semanticamente documentos científicos. Apesar disso, seu *design* e adaptações com o passar dos anos permitiram que seu uso descrevesse outros tipos de documentos, tais como páginas Web.

Sua especificação é limitada a prover um nível de marcação semântica para páginas acessíveis na Web, desde documentos estáticos até aplicações dinâmicas (W3C, 2011a). A versão 4 da HTML foi lançada em 1999 e, desde então, a Web sofreu grandes mudanças. A especificação da versão 5 da HTML ainda não foi finalizada, entretanto, a maioria dos navegadores modernos já dão algum suporte para a linguagem. Entre as novas funcionalidades disponíveis na HTML5 destacam-se o elemento *canvas*, utilizado para desenhar, os elementos de áudio e vídeo para reprodução de mídia, melhor suporte para armazenamento local dos dados, entre outros (W3SCHOOLS, 2011).

2.2 JAVASCRIPT

JavaScript é a linguagem da Web. Iniciou como um meio de manipular alguns tipos selecionados de elementos em uma página Web (como imagens e campos de formulário) e ganhou destaque com o passar dos anos, devido a crescente necessidade do usuário interagir

com as páginas (STEFANOV, 2010, p. 1).

O núcleo da linguagem de programação JavaScript é baseado nos padrões da linguagem ECMAScript (ECMA, 2011). A terceira versão do padrão foi oficialmente aceita em 1999 e é a implementação atualmente rodando através dos navegadores web. A quarta versão foi abandonada e a quinta versão foi aprovada em dezembro de 2009, 10 anos após a anterior (STEFANOV, 2010, p. 5).

A linguagem JavaScript utiliza o *Document Object Model* (DOM) para interagir com a HTML. A função do DOM é permitir que programas e *scripts* acessem e atualizem dinamicamente o conteúdo, estrutura e estilo dos documentos. O documento pode ser processado e o resultado deste processamento pode ser incorporado a página atual (W3C, 2011b).

JavaScript é também uma linguagem pouco usual. Ela é baseada em funções, que também podem ser definidas como classes. Apesar de não haver suporte a algumas funcionalidades como herança e sobrescrita de métodos, a criação de um novo objeto através de uma função ocorre de maneira semelhante. Existem algumas técnicas para resolver esse problema, sendo que uma delas é o modelo baseado em classes, proposto por John Resig. Esse modelo é constituído de uma função anônima que cria uma função chamada `Class`. Seguindo esse modelo, toda classe criada necessita ser uma sub-classe de `Class`. O construtor é representado por uma função chamada `init`, sendo possível também fazer a sobrescrita dos métodos. Para chamar o método da classe-pai utiliza-se o comando `this._super()` (RESIG, 2008). O Anexo A contém o código-fonte do modelo apresentado.

2.3 OPENGL ES 2.0

A *Open Graphics Library for Embedded Systems* é uma *Application Programming Interface* (API) para renderização de gráficos em 3D em dispositivos móveis ou embarcados. A OpenGL ES 2.0 é a primeira biblioteca para dispositivos móveis e embarcados com hardware 3D programável, incluindo telefones celulares, *Personal Digital Assistant* (PDA), consoles, aplicações e veículos. Com a OpenGL ES 2.0, a programação de *shaders* fez seu caminho para os dispositivos pequenos e portáteis (KHRONOS, 2011a).

Segundo Amoroso (2008), "um *shader* é um conjunto de instruções para o processamento de efeitos de renderização em uma imagem tridimensional. Com um *shader*,

as imagens são constantemente processadas, o que gera maior flexibilidade para programadores e maior qualidade de imagem para os jogadores".

Como a OpenGL ES 2.0 é derivada da OpenGL 2.0, é possível programar conteúdo para jogos de maneira extremamente rica. Com OpenGL ES 2.0, muito da capacidade de programação gráfica dos *desktops* está agora disponível em sistemas embarcados. A OpenGL ES 2.0 implementa um *pipeline* gráfico com suporte a programação de *shaders* e consiste de duas especificações: a *OpenGL ES 2.0 API specification* e a *OpenGL ES Shading Language Specification* (OpenGL ES SL) (MUNSHI; GINSBURG; SHREINER, 2008, p. 3).

2.4 WEBGL

WebGL é uma API baseada na versão 2.0 do OpenGL ES. Esta API é exposta através do elemento *canvas*, presente na HTML5, sendo manipulada utilizando o DOM, também presente na HTML5. A WebGL já vem instalada por padrão nos navegadores mais populares, com exceção do Internet Explorer. A especificação da API é mantida pelo grupo Khronos, o mesmo grupo responsável pela especificação do OpenGL ES, sendo que a versão disponibilizada atualmente é a 1.0 (KHRONOS, 2011b).

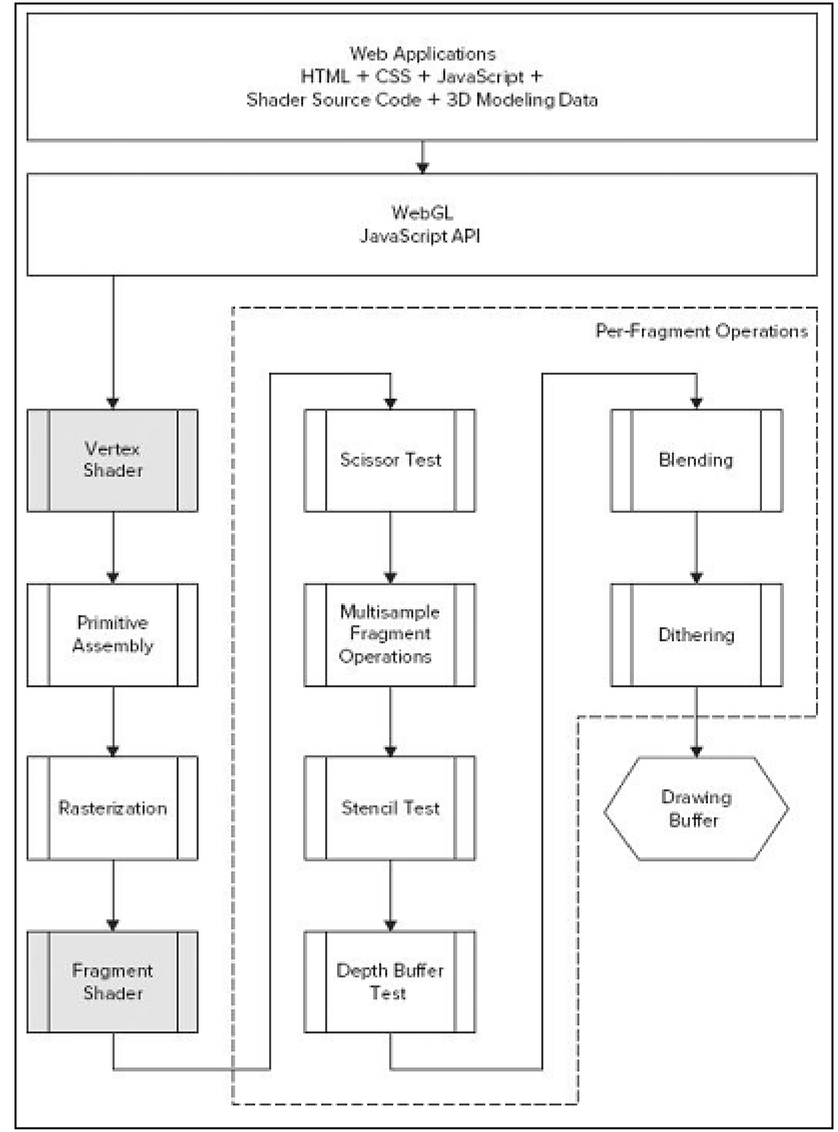
A WebGL é uma API baseada em s*haders*, utilizando a *openGL Shading Language* (GLSL) para cumprir esta função, similar ao que ocorre com o OpenGL ES 2.0. Porém, apesar das semelhanças, há concessões feitas para linguagens como JavaScript, que não tratam gerenciamento de memória. Isso é necessário, pois a GLSL possui como base a linguagem C, que trabalha com gerenciamento de memória (KHRONOS, 2011c). O GLSL é executado em WebGL através de seu *pipeline* gráfico.

2.4.1 *Pipeline* gráfico

Uma aplicação web tipicamente consiste de arquivos HTML, *Cascading Style Sheets* (CSS) e JavaScript que executam em um navegador. Adicionalmente a esses arquivos, uma aplicação WebGL também contém o código-fonte de seus *shaders* e dados representando modelos 2D ou 3D.

O navegador web não requer nenhum *plugin* para executar o WebGL, pois isso é

suportado nativamente por ele. O JavaScript é o responsável por chamar a API WebGL para enviar a informação ao seu *pipeline*, contendo a maneira em que eles devem ser desenhados. Além da maneira como devem ser desenhados, ainda é enviado o código-fonte dos seus dois estágios programáveis, o *vertex shader* e o *program shader* (ANYURU, 2012, p.7). A Figura 1 exibe o *pipeline* da WebGL.



Fonte: Anyuru (2012, p.8).

Figura 1 – *Pipeline* utilizado na WebGL

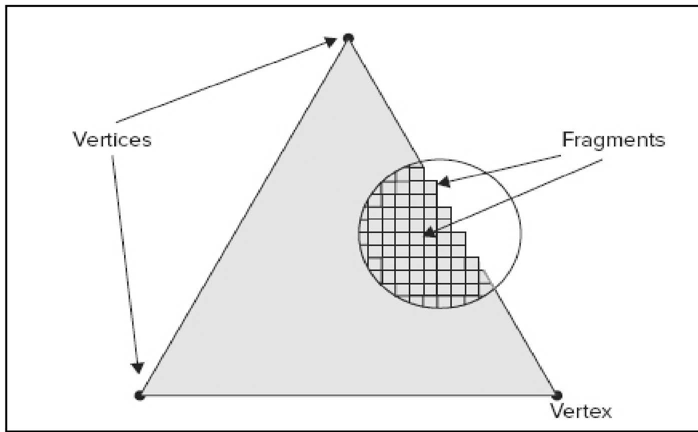
Para obter-se cenas realistas em 3D, não basta saber como desenhar o objeto em determinadas posições, é preciso levar em conta outros aspectos, como a maneira como os objetos irão se parecer quando uma fonte de luz brilhar sobre eles. O nome desse processo é chamado *shading*. Em WebGL existem dois estágios de *shading*: o *vertex shader* e o *fragment shader* (ANYURU, 2012, p. 9).

O *vertex shader* é o primeiro lugar onde os dados, como os vértices, são processados após serem enviados pela API do JavaScript. Através dele é possível manipular o vértice de

diversas maneiras. Antes do processo de *shading* iniciar, geralmente ocorre a transformação do vértice através da sua multiplicação com a matriz de transformação, o que posiciona o objeto em posições específicas na cena. O código do *vertex shader* é escrito utilizando a GLSL. Seus atributos geralmente contêm dados específicos para cada vértice, enquanto variáveis do tipo `uniform` são constantes para todos os vértices. Ainda existem variáveis chamadas `varying`, que são utilizadas para passar informações ao *fragment shader* (ANYURU, 2012, p. 9-10).

Após o *vertex shader,* ocorrem dois passos antes dos dados chegarem ao *fragment shader*: a montagem primitiva e a rasterização. Na montagem primitiva é feita a montagem dos vértices em primitivas geométricas, tais como triângulos, linhas ou pontos. Para cada uma dessas primitivas é feita uma verificação que busca descobrir se ela está dentro da região 3D visível na tela no momento. As primitivas geométricas que estejam visíveis são enviadas para rasterização e as que não estejam são removidas do processo. Caso alguma primitiva esteja parcialmente visível, ela é marcada, para ser revista mais adiante no processo. Ao término da montagem primitiva, inicia-se a rasterização. Esta é responsável por converter as primitivas em fragmentos, que são enviados ao *fragment shader* (ANYURU, 2012 p. 12-14).

Um fragmento pode ser descrito como um pixel que poderá ser desenhado na tela. Entretanto, nem todos os fragmentos serão enviados ao buffer de desenho, pois algumas operações podem descartá-los nos últimos passos do *pipeline*. O código do *fragment shader* também é escrito utilizando a GLSL, contendo algumas variáveis pré-construídas. Assim como o *vertex shader,* também conta com variáveis do tipo `uniform` e `varying`, sendo que a última é responsável por receber o conteúdo enviado pelo *vertex shader*. Ainda existe um tipo de variáveis chamado *sampler,* específico para tratamento de texturas (ANYURU, 2012 p. 14-16). A Figura 2 demonstra a diferença entre vértices e os tipos de informações trabalhados.



Fonte: Anyuru (2012, p. 16).

Figura 2 - Diferença entre os dados trabalhados pelo *vertex shader* e pelo *fragment shader*

A saída do *fragment shader* é a variável `gl_FragColor` com a cor do fragmento em questão. Após sua execução, ainda ocorrem algumas operações. A primeira operação feita é chamada *Scissor Test*, que determina se o fragmento pertencente às primitivas parcialmente visíveis está dentro da área visível. Em sequência é feita a operação chamada *Multisample Fragment Operations.* Ela modifica o alfa do fragmento para reproduzir o efeito de *anti-aliasing*, técnica que melhora a aparência das bordas de um polígono para aparecerem de maneira mais suave na tela (ANYURU, 2012, p. 17-18).

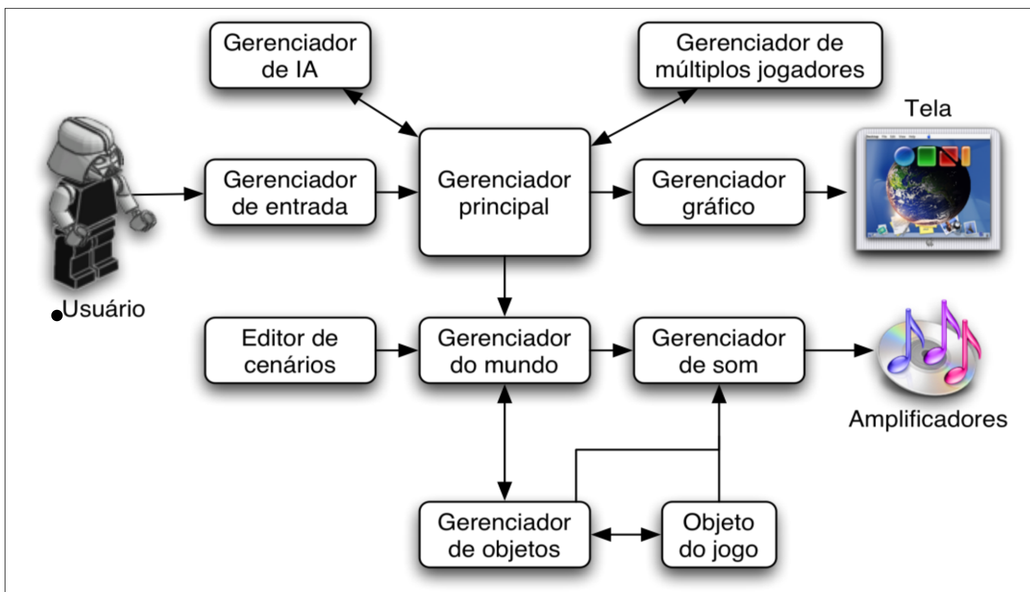
A próxima operação a ser feita chama-se *Stencil Test,* responsável por executar testes no fragmento para decidir se ele deve ou não ser descartado. Após o *Stencil Test,* é executado o *Depth Buffer Test.* Este teste é responsável por determinar quais pixels devem ser exibidos na tela quando ocorre a sobreposição de fragmentos. Essa verificação é controlada através do *z-buffer*; fragmentos com valores menores estão mais perto da tela, portanto devem ser desenhados, enquanto os mais distantes devem ser descartados. Ao fim, existem ainda as operações de *Blending,* que permitem combinar a cor do fragmento atual com a cor de outro fragmento e de *Dithering,* utilizado para organizar as cores de uma maneira que se crie a ilusão de que haja mais cores disponíveis do que realmente há (ANYURU, 2012, p. 18-19).

2.5 MOTOR DE JOGOS

Um motor de jogos é uma abstração dos detalhes relacionados as tarefas comuns no desenvolvimento de jogos, como renderização, física e entrada de dados, permitindo aos desenvolvedores focar nos detalhes que tornam seus jogos únicos (WARD, 2008).

Um motor de jogos é uma entidade complexa, que consiste em algo maior que uma camada de renderização que desenha triângulos. Também é mais do que apenas uma coleção de técnicas desorganizadas. Um motor de jogos deve lidar com os problemas de gerenciamento de um grafo de cena como um *front end* que provê eficientemente a entrada para o renderizador, sendo essa uma implementação feita em software ou hardware (EBERLY, 2001, p. xxvii).

Segundo Pamplona (2005, p. 2), "a idéia é que os motores implementem funcionalidades e recursos comuns à maioria dos jogos de determinado tipo, permitindo que estes recursos sejam reutilizados a cada novo jogo criado". A Figura 3 exibe a arquitetura típica de um motor de jogos 3D.



Fonte: Gomes e Pamplona (2005, p. 2).

Figura 3 - Arquitetura de um motor de jogos

A arquitetura demonstrada possui dez componentes. O gerenciador de entradas identifica os eventos respectivos a entrada de dados e os envia para o gerenciador principal. O gerenciador gráfico é responsável por transformar um modelo matemático do estado atual do jogo em uma visualização para o usuário. A função do gerenciador de som é executar os sons a partir de eventos ocorridos no jogo. Para gerenciar o comportamento dos objetos desenvolvidos pelo designer é utilizado o gerenciador de inteligência artificial (PAMPLONA, 2005, p. 20).

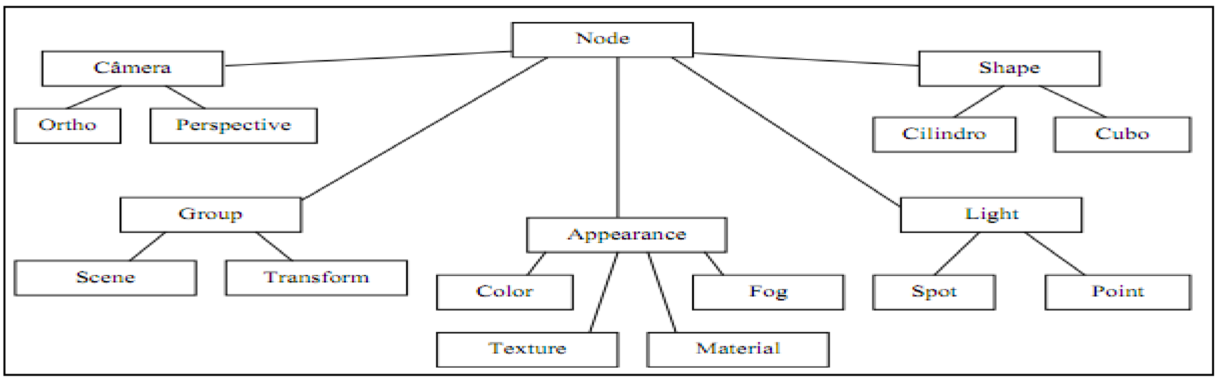
Além dos componentes citados ainda há o gerenciador de objetos, responsável por carregar e controlar o ciclo de vida de um grupo de objetos no jogo. Um objeto no jogo é um objeto que possui dados relevantes para uma entidade que faça parte do jogo, controlando a posição, velocidade, força, entre outros aspectos. O gerenciador do mundo armazena o estado atual do jogo através do gerenciador de objetos. Também estão disponíveis na arquitetura o editor de cenários, responsável por criar mundos que são carregados pelo gerenciador de mundos e o gerenciador principal, responsável por gerenciar os demais componentes (PAMPLONA, 2005, p. 20-21).

2.5.1 Grafo de cena

Segundo Pozzer (2007, p. 1), grafos de cena "são estruturas de dados, organizadas através de classes, onde por meio de hierarquia de objetos e atributos, pode-se mais

facilmente especificar cenas complexas. Cada objeto ou atributo é representado por uma classe, que possui informações sobre sua aparência física, dentre outros fatores."

Um grafo de cena trata problemas que surgem em composições ou gerenciamentos de cenas. Popularizado pelo *Silicon Graphics International* (SGI) *Open Inventor*, API 3D que oferece uma solução abrangente para os problemas interativos da programação gráfica, o grafo de cena protege o desenvolvedor de se preocupar com os detalhes que compõe a renderização, fazendo com que ele foque nos objetos do qual deseja renderizar, ao invés de se preocupar com a lógica da renderização em si (WALSH, 2002). A Figura 4 apresenta um exemplo de grafo de cena.



Fonte: Pozzer (2007, p. 2).

Figura 4 - Exemplo de grafo de cena

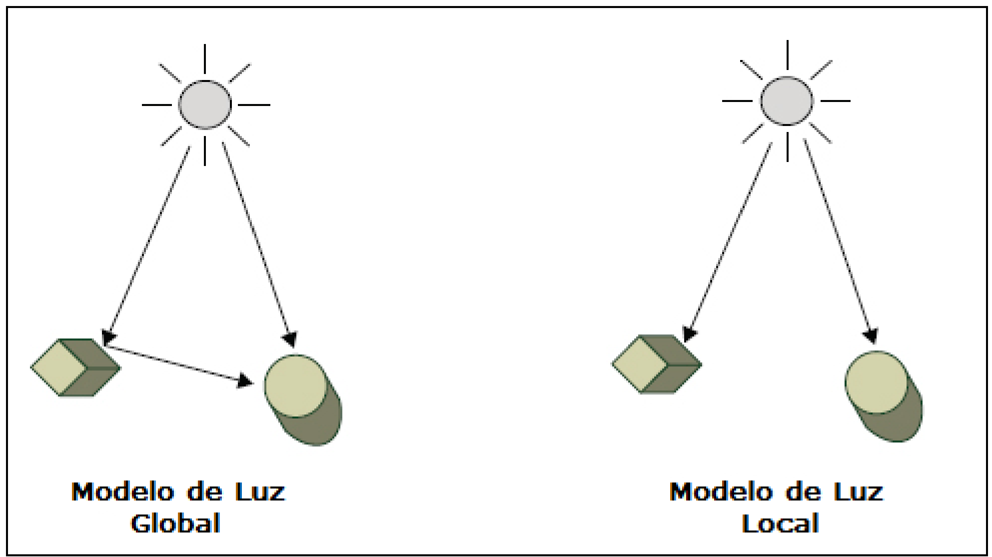
O uso da hierarquia de objetos feita pelo grafo de cena é uma maneira mais plausível de representar a cena, podendo gerar cenas ou objetos mais complexos através do agrupamento das entidades básicas. Para se implementar o grafo de cena, é possível utilizar uma hierarquia de classes, conforme demonstrado na Figura 4 (POZZER, 2007, p. 3).

A principal classe é a classe Node, de onde derivam as demais classes. A classe Group contém a descrição de um ou mais elementos na cena. Essa classe aplica as transformações geométricas de maneira equivalente para todos os elementos geométricos do grupo. A classe Appearance armazena as características que representam a aparência de cor dos objetos. Já a classe Light específica as fontes de iluminação presentes na cena. Por último, a classe Shape define os objetos geométricos e a classe Camera define a manipulação de câmera da cena (POZZER, 2007, p. 3-4).

2.5.2 Iluminação

Ao simular luz em gráficos 3D, é possível utilizar dois tipos diferentes de modelos de

luz: o modelo de luz global e o modelo de luz local. Um modelo de luz global usa informações de outros objetos diretamente iluminados pela luz. Este modelo permite que objetos sejam iluminados por luzes que sejam refletidas através de outros objetos. O modelo de luz local apenas contabiliza a luz que provém diretamente do ponto de luz. Isso faz com que os objetos não sejam iluminados pela luz que outros objetos refletem (ANYURU, 2012, p. 249). A Figura 5 exibe a diferença entre essas duas situações.



Fonte: Anyuru (2012, p. 250).

Figura 5 - Diferença entre o modelo de luz global e local

Como WebGL geralmente é utilizado para gráficos 3D em tempo real, quase sempre é utilizado um modelo de luz local. Um método comum utilizado para representar o modelo de luz local é o modelo de reflexão Phong, apresentado na seção 2.5.2.1.

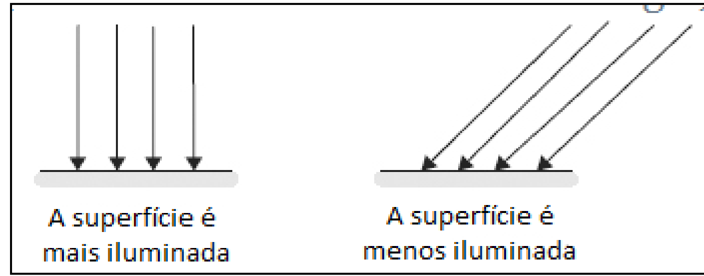
2.5.2.1 Modelo de reflexão Phong

O método de reflexão Phong é assim nomeado em homenagem ao seu desenvolvedor, Bui Tuong Phong, que o criou em 1973. Esse modelo se baseia no fato de que a cor de um objeto real do mundo é indicada através da cor de luz de que deixa sua superfície. Ou seja, um objeto vermelho irá refletir luz vermelha em sua maioria. Neste modelo, o resultado da cor de um ponto é a soma de três diferentes reflexões: ambiente, difusa e especular. Para calcular essa reflexão, também é considerado que há três diferentes luzes para cada uma das reflexões (ANYURU, 2012, p. 251).

A luz de ambiente é uma luz que reflete por toda a cena, fazendo com que todos os lados de um objeto sejam iluminados de maneira equivalente. Ao multiplicar uma luz que

contenha uma cor *Red Green Blue* (RGB) com um objeto que também tenha uma cor RGB, será obtido como resultado a cor de reflexão demonstrada na superfície do objeto.

A luz difusa leva em consideração a direção da luz de entrada para calcular a quantidade de luz refletida. Para este tipo de reflexão, a luz é refletida uniformemente em todas as direções. Raios de luz que atingem uma superfície em um ângulo perpendicular refletem mais luz do que raios de luz que atingem a superfície com um ângulo que fique entre a superfície e os raios de luz. A Figura 6 exibe essa diferença.

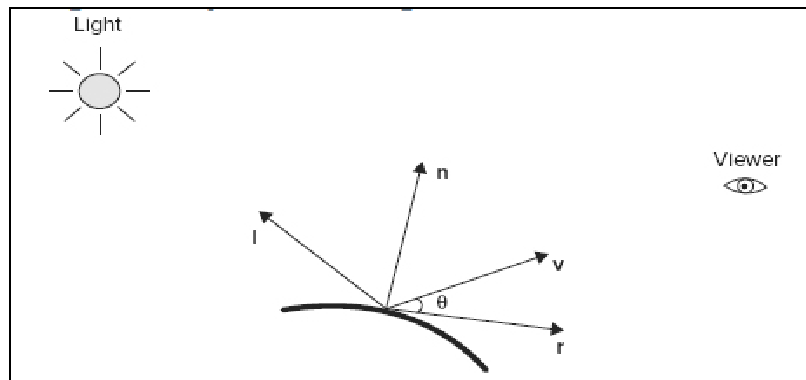


Fonte: Anyuru (2012, p. 253).

Figura 6 - Diferença reflexão conforme o ângulo que o raio da luz difusa atinge

A equação para calcular a reflexão da luz difusa é similar à equação da luz de ambiente, porém também é levado em consideração o cosseno do menor ângulo entre a superfície e o vetor que aponta para a direção da luz (ANYURU, 2012, p. 252-253).

A reflexão especular, ao contrário das reflexões de ambiente e difusa, leva em conta a direção de visão. A geometria da reflexão especular é demonstrada na Figura 7.



Fonte: Anyuru (2012, p. 256).

Figura 7 - Geometria usada para a a reflexão especular

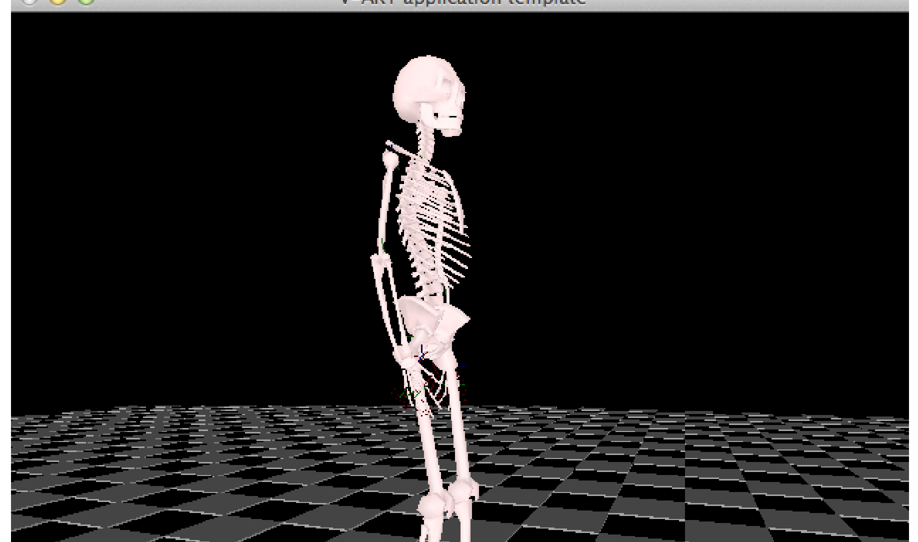
Na Figura 7, o raio de luz vem de uma direção específica informada pelo vetor **l**, que aponta em direção a fonte de luz. A matriz normal da superfície é representada por **n.** Para objetos muito brilhantes, a luz geralmente é refletida na direção **r**. Se os objetos forem menos brilhantes, a luz acaba sendo refletida ao redor do vetor **r**. Caso o ângulo entre o vetor **v** e o vetor **r** seja zero, a luz será refletida na direção do observador. Quanto maior o ângulo entre esses dois vetores, menos luz é refletida em direção ao observador (ANYURU, 2012, p. 256-257).

2.6 V-ART

O *Virtual ARTiculations for virtual reality* (V-ART) é um *framework* desenvolvido em C++ para facilitar a criação de programas com ambiente 3D, em especial os que contêm humanoides. Ele é inteiramente orientado a objetos, possui um sistema de suporte a animações e permite a representação de articulações biologicamente corretas. O V-ART foi projetado para ser multiplataforma e independente de API gráfica (VART, 2007).

Através do V-ART é possível importar (parcialmente) modelos geométricos no formato Wavefront OBJ. Existem também recursos mais avançados para humanoides e modelos articulados em geral, descritos em XML (SCHNEIDER, 2012). A Fonte: **Scheiner (2012).**

Figura 8 exibe uma cena construída através do V-ART.



Fonte: Scheiner (2012).

Figura 8 - Cena construída através do V-ART

O V-ART foi desenvolvido com o foco na simulação de personagens articulados. Uma articulação é definida como uma conexão entre duas ou mais peças esqueléticas, compostas por até três *Degrees of Freedom* (DOF). Uma articulação com apenas um DOF é chamada uniaxial, sendo o tornozelo um exemplo desse tipo de articulação. Uma articulação biaxial representa uma articulação com dois DOFs, tais como os joelhos e o punho. Finalmente, uma articulação com três DOFs é chamada de triaxial, como é o caso do ombro (AULA DE

ANATOMIA, 2012).

2.7 TRABALHOS CORRELATOS

Atualmente, existem algumas ferramentas disponíveis para o desenvolvimento de jogos utilizando WebGL. O CopperLicht (AMBIERA, 2011) é um motor de jogos 3D para a biblioteca WebGL e JavaScript, para criação de jogos e aplicações 3D executadas através de um navegador web. Utiliza o *canvas*, suportado pelos navegadores web modernos e é apto a renderizar gráficos 3D através de aceleração de hardware sem nenhum *plugin*. Possui suporte aos formatos de arquivo `3ds`, `obj`, `x`, `lwo`, `b3d`, `csm`, `dae`, `dmf`, `oct`, `irrmesh`, `ms3d`, `my3D`, `mesh`, `lmts`, `bsp`, `md2` e `stl`.

O CopperLicht é gratuito para uso, porém é possível adquirir uma licença comercial. Esta licença inclui acesso a documentação do código, *scripts* que permitem a criação de uma versão comprimida do motor de jogos, acesso aos *patches* preliminares, suporte de alta prioridade e atualizações gratuitas durante um ano. A Figura 9 apresenta a visualização de um ambiente 3D por meio da CopperLicht.

Fonte: Ambiera (2011).

Figura 9 - Ambiente visualizado por meio da CopperLicht

O CubicVR (CUBIC, 2011) é um motor de jogos escrito na linguagem C++. Atualmente a equipe de desenvolvimento está fazendo a conversão do motor de jogos para rodar nos navegadores web, utilizando WebGL e JavaScript. O trabalho está nas fases iniciais, sendo possível visualizar algumas demonstrações do que já foi feito até o momento.

O CubicVR é um motor de jogos de código aberto e gratuito. Atualmente seus fontes podem ser acessados através do *site* de armazenamento de código GitHub. A Figura 10 exibe a

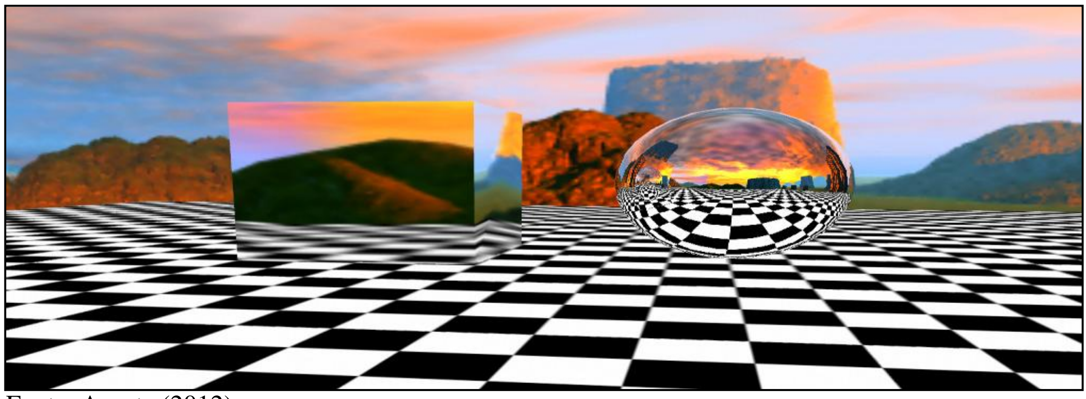
visualização de um protótipo, utilizando o CubicVR.

Fonte: Cubic (2011).

Figura 10 - Exemplo exibindo o estado atual da conversão do CubicVR para WebGL

O Three.js é uma biblioteca gráfica 3D para navegadores web modernos que pode renderizar seus objetos para o *canvas,* WebGL e *Scalabe Vector Graphics* (SVG). Ela dá completo suporte as funcionalidades do WebGL, enquanto permite que seja utilizado o mesmo código para os outros renderizadores. A biblioteca abstrai muito do trabalho de baixo nível feito pelo WebGL, como trabalhar com *shaders*.

A biblioteca também possui *Helpers* para as formas geométricas mais comuns, tais como esferas, cubos e cilindros, além de um sistema de partículas completamente funcional, mapeamento de texturas e suporte a detecção de colisões em nível básico (WILLIAMS, 2011, p. 117). A Figura 11 exibe um exemplo de aplicação criada com a biblioteca Three.js.



Fonte: Acosta (2012).

Figura 11 - Exemplo de aplicação que utiliza a biblioteca `Three.js`

As principais características de cada um dos motores de jogos são listadas no Quadro 1.

Características / trabalhos correlatos	CopperLicht	CubicVR	Three.js
Gerenciador de objetos	x	x	x
Grafo de cena	x		x
Textura	x	x	x
Iluminação	x	x	x
Leitura de cena através de arquivos	x		x
Editor de mundo	x		
Abstração do WebGL			x
Suporte a vários renderizadores			x

Quadro 1 - Características dos trabalhos correlatos

3 DESENVOLVIMENTO

Neste capítulo são apresentadas as etapas envolvidas no desenvolvimento do motor de jogos FURB WebGL (FWGL). São apresentados os principais requisitos, a especificação, a implementação e, ao final, os resultados e discussões.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

O motor de jogos deverá:

a) fornecer um *framework* para desenvolvimento em WebGL (Requisito Funcional - RF);
b) fornecer um grafo de cena para gerenciar e manipular os objetos hierarquicamente (RF);
c) fornecer recursos para a movimentação da câmera (RF);
d) fornecer recursos para que sejam feitas transformações geométricas (RF);
e) fornecer recursos para o gerenciamento de luz no ambiente (RF);
f) fornecer recursos para importar objetos que sigam o padrão estabelecido (RF);
g) utilizar a linguagem JavaScript para implementação (Requisito Não-Funcional - RNF);
h) utilizar a biblioteca WebGL para desenvolvimento da parte gráfica (RNF);
i) funcionar de maneira equivalente nos navegadores que suportam WebGL (RNF).

3.2 ESPECIFICAÇÃO

A especificação do motor de jogos foi desenvolvida utilizando o paradigma de orientação a objetos, através de diagramas da *Unified Modeling Language* (UML), utilizando a ferramenta Enterprise Architect. Na seção a seguir são apresentados os diagramas de caso de uso, diagramas de classe e diagramas de sequência.

3.2.1 Casos de uso

Nesta seção são descritos os casos de uso das funcionalidades disponibilizadas pelo motor de jogos, ilustrados pela Figura 12.

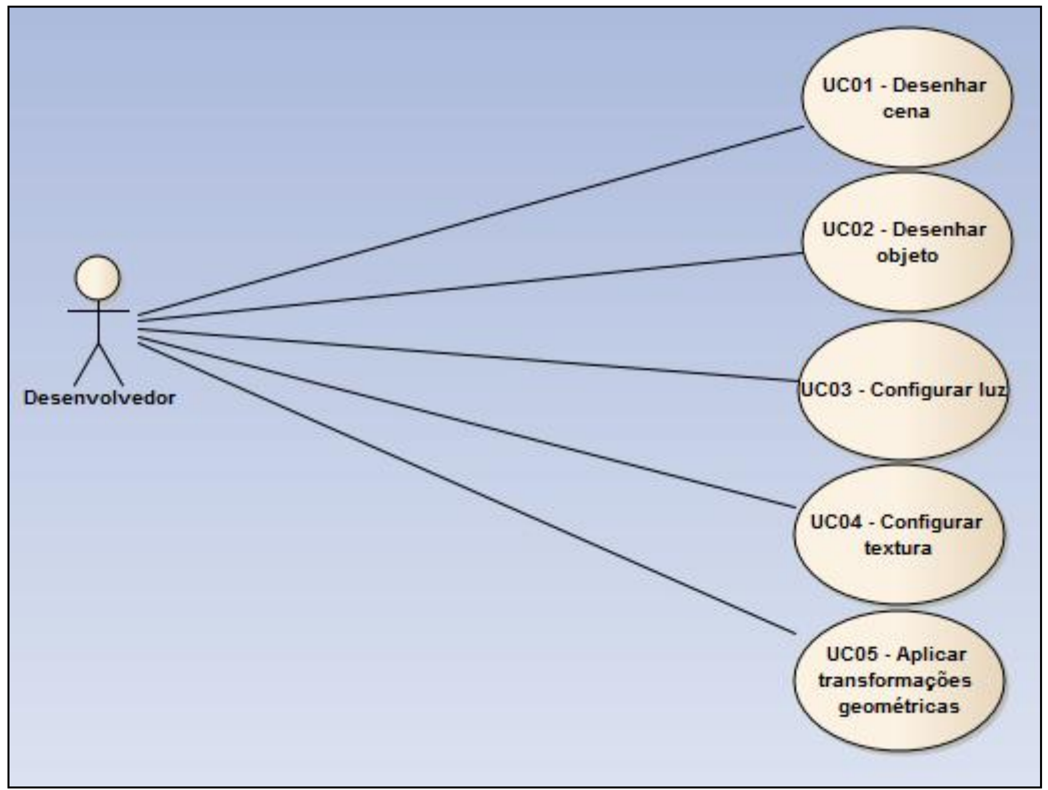


Figura 12 - Diagrama de casos de uso

3.2.1.1 Desenhar cena

Este caso de uso descreve como o `Desenvolvedor` deve proceder para criar a cena e desenhá-la a partir do elemento *canvas*. Detalhes são apresentados no Quadro 2.

UC01 – Desenhar cena	
Descrição	Permite que o `Desenvolvedor` crie uma cena e a desenhe na tela.
Pré-Condição	Motor de jogos carregado. Elemento *canvas* carregado.
Cenário Principal	1. O `Desenvolvedor` cria uma cena. 2. O `Desenvolvedor` cria uma câmera. 3. O `Desenvolvedor` associa o elemento *canvas* à cena. 4. O `Desenvolvedor` requisita que a cena seja desenhada.
Pós-Condição	Cena desenhada no elemento *canvas*.

Quadro 2 - Caso de uso UC01

3.2.1.2 Desenhar objeto

Este caso de uso descreve como o `Desenvolvedor` deve criar um objeto e adicioná-lo na cena. Detalhes são apresentados no Quadro 3.

UC02 – Desenhar objeto	
Descrição	Permite que o `Desenvolvedor` crie um objeto e o exiba na cena.
Pré-Condição	Motor de jogos carregado. Cena criada.
Cenário Principal	1. O `Desenvolvedor` cria o objeto. 2. O `Desenvolvedor` adiciona o objeto à cena. 3. O `Desenvolvedor` requisita que a cena seja desenhada.
Fluxo alternativo	a) Após o passo 1, o `Desenvolvedor` adiciona o objeto criado como filho de um objeto já existente.
Pós-Condição	Cena com os objetos desenhados.

Quadro 3 - Caso de uso UC02

3.2.1.3 Configurar luz

Este caso de uso descreve como os procedimentos para o `Desenvolvedor` configurar a luz em uma cena do motor de jogos. Detalhes são apresentados no Quadro 4.

UC03 – Configurar luz	
Descrição	Permite que o `Desenvolvedor` configure os parâmetros de luz de uma cena do motor de jogos.
Pré-Condição	Motor de jogos carregado. Cena criada.
Cenário Principal	1. O `Desenvolvedor` cria uma representação da luz. 2. O `Desenvolvedor` informa as cores e localização onde a luz permanecerá na cena. 3. O `Desenvolvedor` requisita que a cena seja desenhada.
Pós-Condição	Cena desenhada refletindo os efeitos de luz configurados.

Quadro 4 - Caso de uso UC03

3.2.1.4 Configurar textura

Este caso de uso descreve como o `Desenvolvedor` deve configurar a textura em um objeto da cena. Detalhes são apresentados no Quadro 5.

UC04 – Configurar textura	
Descrição	Permite que o `Desenvolvedor` aplique uma textura em um objeto da cena.
Pré-Condição	Motor de jogos carregado. Objeto criado.
Cenário Principal	1. O `Desenvolvedor` cria uma representação da textura. 2. O `Desenvolvedor` informa a imagem que deseja utilizar na textura. 3. O `Desenvolvedor` adiciona a textura ao objeto.
Pós-Condição	Objeto configurado para ser exibido conforme a textura configurada

Quadro 5 - Caso de uso UC04

3.2.1.5 Aplicar transformações geométricas

Este caso de uso descreve como o Desenvolvedor interage dom o motor de jogos para aplicar transformações geométricas em um objeto desenhado na cena. Detalhes são apresentados no Quadro 6.

UC05 – Aplicar transformações geométricas	
Descrição	Permite que o `Desenvolvedor` aplique uma transformação geométrica em um objeto da cena.
Pré-Condição	Motor de jogos carregado. Objeto criado. Cena desenhada.
Cenário Principal	1. O `Desenvolvedor` recupera o objeto da cena. 2. O `Desenvolvedor` aplica uma transformação geométrica no objeto recuperado. 3. O `Desenvolvedor` requisita que a cena seja desenhada novamente.
Pós-Condição	Posição do objeto alterada, conforme transformações geométricas aplicadas

Quadro 6 - Caso de uso UC05

3.2.2 Diagrama de classes

Nesta seção são descritas as classes e estruturas que constituem o motor de jogos desenvolvido. Na Figura 13 são exibidas as dependências entre os pacotes, assim com as classes que os compõem.

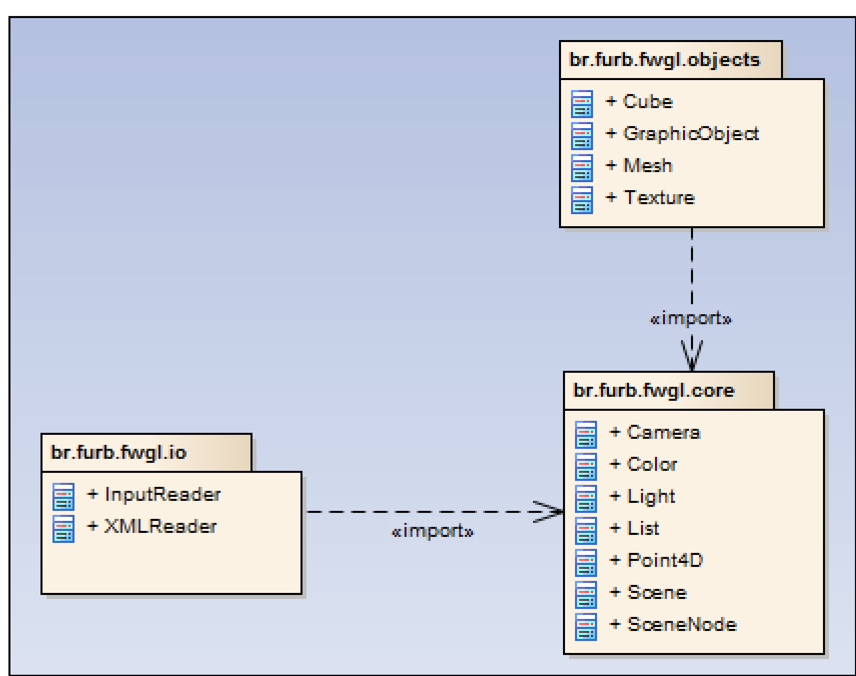


Figura 13 - Diagrama de pacotes do motor de jogos

3.2.2.1 Pacote `br.furb.fwgl.core`

O pacote `br.furb.fwgl.core` contém o núcleo do motor de jogos implementado neste trabalho. É através desse pacote que uma cena é criada e gerenciada, permitindo ao desenvolvedor a criação do seu aplicativo sem obrigá-lo a conhecer o funcionamento arquitetural do WebGL. Dentro deste pacote existe o arquivo Setup.js, que executa uma função anônima que dá suporte ao modelo de classes conhecido em linguagens como o Java. Ao importar o código-fonte do motor de jogos, este arquivo deve ser o primeiro a ser declarado, garantindo assim que a estrutura de classes já tenha sido inicializada. A Figura 14 apresenta as classes que compõe este pacote, assim com seus principais métodos e atributos.

A classe `Scene` é a principal responsável por fazer o gerenciamento da cena. É através desta classe que o desenvolvedor controla quais objetos estarão presentes na tela, assim como as definições de luz e câmera presentes na cena. Para adicionar uma nova câmera à cena, o desenvolvedor deve utilizar o método `addCamera`, passando um objeto da classe `Camera` como parâmetro. Para adicionar uma luz o método a ser utilizado chama-se `addLight`, que recebe um objeto da classe Light como parâmetro. Os objetos são adicionados através do método `addObject` e precisam ser filhos da classe `GraphicObeject`. A classe `Scene` também é responsável por armazenar o grafo de cena, garantindo que sua chamada ocorra conforme a hierarquia especificada pelo desenvolvedor. Por último, é de responsabilidade da classe `Scene`

iniciar o procedimento de desenho da cena, conforme as configurações definidas e a disposição dos objetos adicionadas à ela.

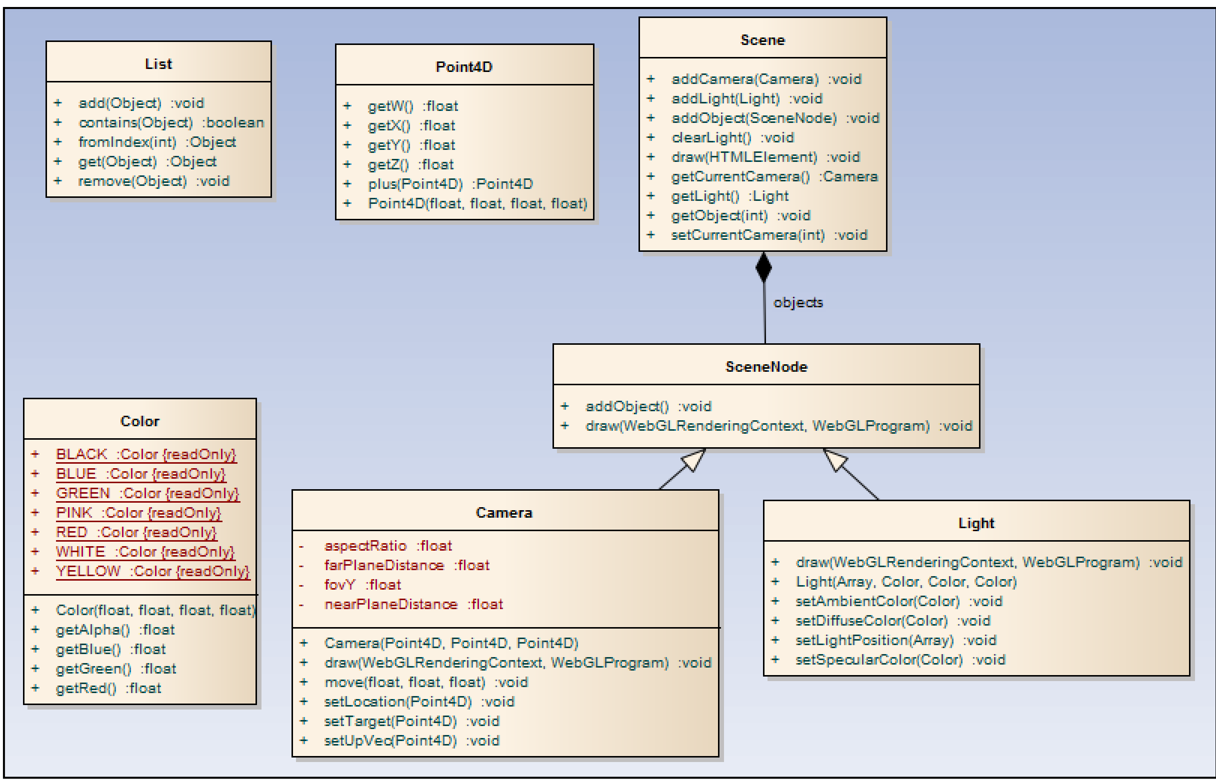


Figura 14 - Diagrama de classes do pacote `br.furb.fwgl.core`

A classe `SceneNode` representa um objeto que pode ser adicionado ao grafo de cena armazenado pela classe `Scene`. Um objeto desta classe possui dois atributos: `parentNode`, que armazena uma referência ao seu objeto-pai na hierarquia ou `null`, caso este seja o objeto raiz e o atributo `objects`, uma lista que contém todos os objetos que são filhos diretos dele. Apesar de ser possível instanciar um objeto desta classe devido as limitações impostas pelo JavaScript, isso não deve ser feito pois ela representa um conceito abstrato.

A classe `Camera` representa uma câmera na cena. Seu construtor recebe 3 parâmetros: `location`, um objeto da classe `Point4D` indicando a localização da câmera; `target`, um objeto `Point4D` que indica a localização para qual a câmera está direcionada e `upVec`, um objeto da classe `Point4D` que indica como a câmera está posicionada. Esta classe tem como base a projeção em perspectiva, sendo possível configurar quatro atributos relacionadas a essa projeção, sendo eles `nearPlaneDistance`, `farPlaneDistance`, `fovY` e `aspectRatio`.

O gerenciamento de luz é feito pela classe `Light`, baseado no modelo de reflexão de Phong. O construtor dessa classe recebe quatro parâmetros, responsáveis por definir a luz da cena. O parâmetro `lightPosition` representa a posição da luz. Os três parâmetros restantes configuram o esquema de cores a ser utilizado: `ambientColor`, `diffuseColor` e

specularColor definem a cor do ambiente, a cor difusa a cor especular, respectivamente.

3.2.2.2 Pacote br.furb.fwgl.objects

O pacote br.furb.fwgl.objects contém as implementações referentes aos objetos contidos em uma cena. Neste pacote estão as classes responsáveis pelas transformações na cena, assim como as classes que dão forma aos objetos na tela. A Figura 15 apresenta os métodos e atributos públicos das classes presentes neste pacote.

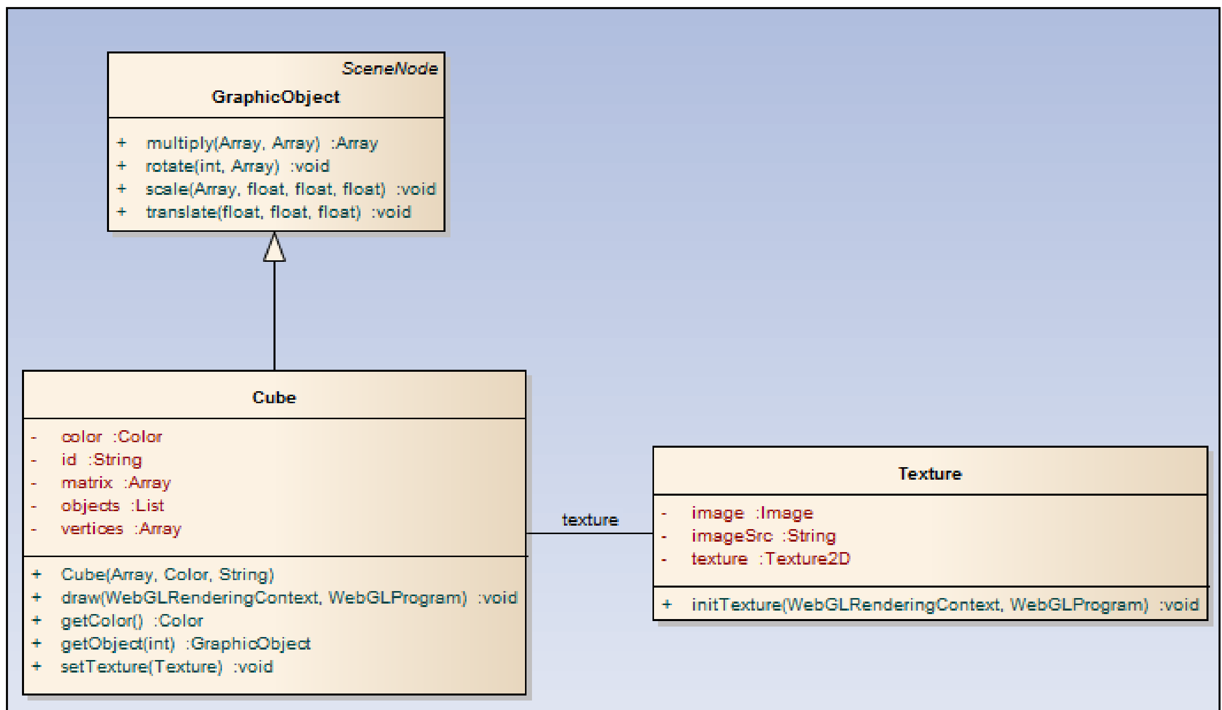


Figura 15 - Diagrama de classes do pacote br.furb.fwgl.objects

A classe GraphicObject é uma classe filha da SceneNode, cujo objetivo é representar um objeto gráfico que pode ser desenhado em uma cena. Ela possui a característica de ser transformável, permitindo que a posição do objeto possa ser alterada programaticamente pelo desenvolvedor. Isso pode ser feito chamando os métodos translate, que é responsável pela translação do objeto, rotate, que rotaciona o objeto para o ângulo determinado e scale, que faz a escala do objeto com base na porcentagem passada como parâmetro. Assim como a classe SceneNode, não deve ser criado um objeto da classe GraphicObject diretamente, pois ela tem o propósito de ser abstrata.

A classe Texture define uma textura a ser utilizada em um objeto da cena. Seu

construtor possui apenas o parâmetro `imageSrc`, uma String que armazena o caminho da imagem no servidor. O único método disponível na classe `Texture` chama-se `initTexture` e deve ser acessado apenas pelo objeto que a possui. Seu objetivo é renderizar a imagem, que se encontra no caminho informado pelo desenvolvedor, em uma textura de duas dimensões que será aplicada no objeto que a contém.

A classe `Cube` estende a classe `GraphicObject` e representa uma maneira simples de definir um cubo na tela. Seu construtor recebe três parâmetros: `vertices`, um `Array` de `float` contendo as coordenadas de onde o cubo deverá ser desenhado, `color`, um objeto da classe `Color` com a cor na qual o objeto deve ser desenhado e `id`, uma String que identifica o objeto que está sendo criado. Dos três parâmetros, apenas o primeiro é obrigatório, ficando a critério do desenvolvedor informar os outros dois.

3.2.2.3 Pacote `br.furb.fwgl.io`

O pacote `br.furb.fwgl.io` é responsável por gerenciar eventos de entrada do usuário, assim como ler os arquivos XML que armazenam uma cena seguindo os padrões estabelecidos. A Figura 16 mostra as classes definidas para este pacote.

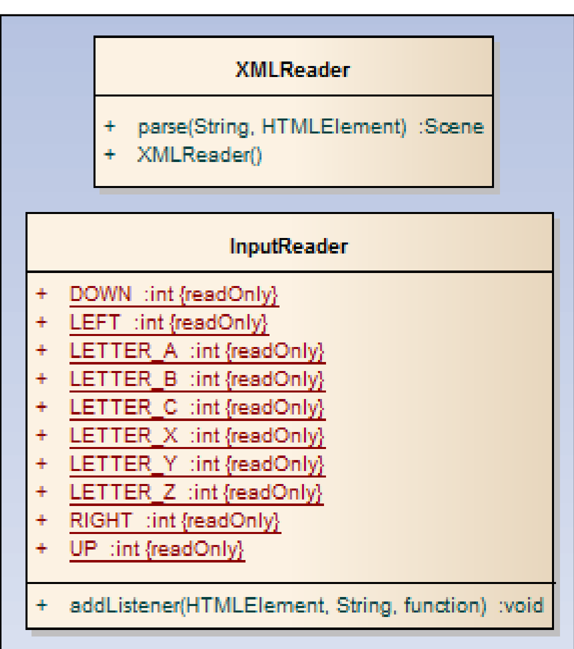


Figura 16 - Diagrama de classes do pacote `br.furb.fwgl.io`

A classe `XMLReader` é responsável pela leitura dos arquivos XML que armazenam uma cena do motor de jogos. Essa classe possui um único método disponível para acesso do

desenvolvedor, chamado `parse`. Esse método é responsável por ler o arquivo no servidor e transformá-lo em uma cena montada equivalente as configurações que foram lidas. O método `parse` recebe dois argumentos: o primeiro parâmetro, nomeado `file`, é uma `String` que contém o endereço do arquivo XML a ser convertido. Esse arquivo deve estar disponível no servidor em que o motor de jogos está sendo executado. O segundo parâmetro, nomeado `canvas`, é o elemento HTML que representa o *canvas* no qual a cena convertida deve ser desenha. Ao ser executado esse método retorna um objeto da classe `Scene`, que terá seus objetos, luzes e câmeras definidos conforme definido no arquivo XML.

Para o tratamento de entrada do usuário é possível utilizar a classe `InputReader`, cujo objetivo é facilitar a comunicação entre os eventos disparados pelo usuário. Ela possui diversas constantes que identificam qual tecla o usuário pressionou, aumentando a legibilidade do código. Através dessa classe também é possível adicionar um *listener* para um evento de determinado elemento, chamando o método `addListener`. Esse método recebe três parâmetros: `element`, que representa o elemento do qual se deseja receber atualizações; `event`, uma `String` que identifica o nome do evento no qual o *listener* será registrado e `fn`, que define a função a ser executada no momento em que o evento ocorrer.

3.2.3 Diagrama de sequência

Este seção apresenta um diagrama de sequência do motor de jogos, ilustrando a maneira em que ocorre a interação entre o desenvolvedor e o motor de jogos para desenhar um cubo em uma cena. A Figura 17 exibe este diagrama.

Para iniciar o desenho da cena, o `Desenvolvedor` chama o método `draw` da classe `Scene`. O método `draw` irá inicializar o contexto WebGL, através do método `initGL`. Logo após ocorre uma chamada ao método `initShaders`, responsável por carregar o *fragment shader* e o *vertex shader* e armazená-los dentro da classe `Scene`. Feito isto, ocorre a chamada ao método `draw` da classe `Camera`, que irá alterar a matriz de projeção para receber a projeção em perspectiva conforme as configurações estabelecidas pelo `Desenvolvedor`. Com a câmera configurada, é feita uma chamada a método `draw` da classe `Light`. Esse método irá atualizar os atributos referentes a luz pertences ao *vertex shader*.

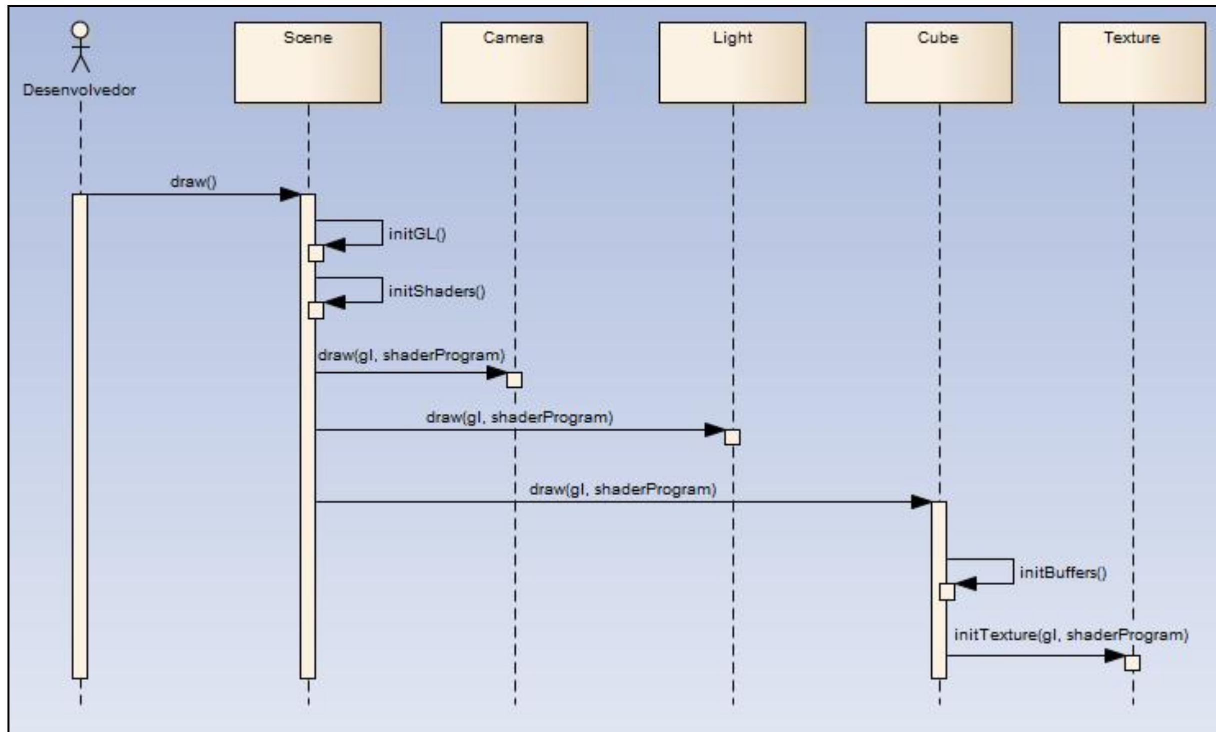


Figura 17 - Diagrama de sequência da ação de desenhar um cubo em uma cena

Após ter preparado o ambiente, configurando a câmera e luz, a classe `Scene` chama o método `draw` da classe Cube. Esse método inicializará os buffers, contendo os dados armazenados no atributo `vertices`. Após iniciar os buffers ele chama o método `initTexture` da classe `Texture`, que criará uma textura baseada na imagem que ela armazena. Por fim, esse método faz a relação entre os buffers criados e os atributos presentes no *vertex shader* e chama o método `drawElements` do objeto `gl` para efetuar o desenho na cena.

3.3 IMPLEMENTAÇÃO

A seguir são mostradas as técnicas e ferramentas utilizadas na implementação, assim com detalhes das principais classes e rotinas implementadas durante o desenvolvimento do motor de jogos. Em Pereira (2012), estão disponíveis os cenários utilizados neste trabalho.

3.3.1 Técnicas e ferramentas utilizadas

O desenvolvimento do motor de jogos foi feito na linguagem JavaScript, utilizando a

API WebGL na versão 1.0. O ambiente de desenvolvimento utilizado foi o Eclipse Juno, na versão 4.2 em conjunto com o *Web Tools Platform* (WTP), que possui uma série de recursos para criação e edição de arquivos HTML, Javascript e XML. Como servidor de aplicação foi utilizado o Apache Tomcat na versão 6.0.35. Os navegadores utilizados durante o desenvolvimento foram o Google Chrome 23.0.1271.64 m, Mozilla Firefox 15.0.1 e Opera 12.02.

Para execução e depuração do motor de jogos foi utilizado um notebook Acer modelo Aspire 4738-6666. Este notebook possui um processador Intel Core i5-480M, memória de 6 GB DDR3 e placa de vídeo Intel HD Graphics.

3.3.2 O motor de jogos

Essa seção descreve os conceitos utilizados no motor de jogos e sua arquitetura, assim como os passos seguidos para abstrair a implementação da API do WebGL em relação ao desenvolvedor.

3.3.2.1 Arquitetura do motor de jogos

O motor de jogos é constituído por um conjunto de cenas. Uma cena é uma área na qual é possível desenhar e que contém um conjunto de objetos, câmeras e luzes. A área de uma cena é delimitada pelo container no qual ela será adicionada. Uma cena é livre para ter quantos objetos achar necessário, sendo limitada apenas pela quantidade de memória disponível. O mesmo cenário dos objetos se aplica as câmeras: é possível ter várias câmeras em uma cena. Porém, apesar de ser permitido conter várias câmeras em uma mesma cena, apenas uma câmera pode ser ativada por vez. Ao contrário dos objetos e das câmeras, é possível ter apenas uma luz em uma cena.

Uma câmera é um objeto especial, que é posicionado em um lugar específico da cena para obter o desenho de um segundo lugar. A câmera não possui um lugar fixo na cena, podendo ter sua localização alterada conforme necessidade. A projeção utilizada pela câmera deste motor de jogos é a projeção em perspectiva, fazendo assim com que objetos mais distantes pareçam menores do que objetos mais próximos.

Um objeto é constituído por um conjunto de vértices, que definem sua localização na

cena. Um objeto pode ter outros objetos como filho, sendo que os objetos-filho são afetados pelas transformações aplicadas pelo objeto-pai. Isso significa que se um objeto-pai receber uma translação de 20 unidades para a direita em uma cena e um objeto-filho receber uma translação de 10 unidades também para a direita, o objeto-filho será deslocado 30 unidades para a direita.

Uma luz é constituída por uma cor que afetará os objetos. A luz afeta todos os objetos da cena e sua intensidade será definida pela localização da luz comparada a localização do objeto. Objetos mais distantes receberão menos impacto da cor da luz, enquanto objetos mais próximos serão altamente afetados por esta.

3.3.2.2 Estrutura do motor de jogos

O motor de jogos FWGL foi construído com o propósito de permitir que o desenvolvedor não utilize as APIs de baixo nível disponível na especificação do WebGL. O desenvolvedor pode desenvolver uma aplicação completa sem ter conhecimento de como funciona o WebGL, fazendo assim com que desenvolvedores iniciantes na área de computação gráfica consigam desenvolver aplicativos com uma curva de aprendizagem menor.

Essa abstração exige que o motor de jogos já tenha um *vertex shader* e um *fragment shader* pré-configurados, fazendo assim com que os valores definidos pelo usuário nas classes do motor de jogos cheguem até o WebGL de maneira transparente. O motor de jogos possui apenas um arquivo de *shaders* para cada um dos tipos, fazendo verificações condicionais para decidir quais recursos devem ser utilizados. O Quadro 7 mostra o código-fonte do *vertex shader* utilizador pelo motor de jogos.

O atributo `aVertexPosition` sempre deve ser informado pelo motor de jogos e indica em qual posição se encontra o objeto que está sendo desenhado. O mesmo vale para as variáveis `uUseLighthing` e `uUseTexture`, que indicam se há ou não a presença de luz e textura no objeto, respectivamente.

```
1.    attribute vec3 aVertexPosition;
2.    attribute vec3 aVertexNormal;
3.    attribute vec2 aTextureCoordinates;
4.    attribute vec4 aVertexColor;
5.
6.    uniform mat4 uMVMatrix;
7.    uniform mat4 uPMatrix;
8.    uniform mat3 uNMatrix;
9.    uniform vec3 uLightPosition;
10.   uniform vec3 uAmbientLightColor;
11.   uniform vec3 uDiffuseLightColor;
12.   uniform vec3 uSpecularLightColor;
13.
14.   uniform bool uUseLighting;
15.   uniform bool uUseTexture;
16.
17.   varying vec2 vTextureCoordinates;
18.   varying vec3 vLightWeighting;
19.
20.   varying vec4 vColor;
21.
22.   const float shininess = 32.0;
23.
24.   void main() {
25.       vec4 vertexPositionEye4 = uMVMatrix * vec4(aVertexPosition, 1.0);
26.       vec3 vertexPositionEye3 = vertexPositionEye4.xyz /
                                    vertexPositionEye4.w;
27.
28.   vec3 vectorToLightSource = normalize(uLightPosition -
                                           vertexPositionEye3);
29.       vec3 normalEye = normalize(uNMatrix * aVertexNormal);
30.
31.       float diffuseLightWeightning = max(dot(normalEye,
                                             vectorToLightSource),0.0);
32.
33.       vec3 reflectionVector = normalize(reflect(-vectorToLightSource,
                                                          normalEye));
34.       vec3 viewVectorEye = -normalize(vertexPositionEye3);
35.
36.       float rdotv = max(dot(reflectionVector, viewVectorEye), 0.0);
37.
38.       float specularLightWeightning = pow(rdotv, shininess);
39.       if(uUseLighting) {
40.           vLightWeighting = uAmbientLightColor +
                          uDiffuseLightColor * diffuseLightWeightning +
                          uSpecularLightColor * specularLightWeightning;
41.       } else {
42.           vLightWeighting = vec3(1.0, 1.0, 1.0);
43.       }
44.
45.       if(uUseTexture) {
46.           vTextureCoordinates = aTextureCoordinates;
47.       } else {
48.           vColor = aVertexColor;
49.       }
50.       gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
51.   }
```

Quadro 7 - *vertex shader* utilizado no motor de jogos

Caso a variável `uUseLighting` seja `true`, a variável `vLightWeighting` vai receber o

resultado do algoritmo de Phong, que é executado nas linhas acima. Caso contrário, a variável recebe um vetor de três posições, com a cor branca atribuída a ele. A variável `uUseTexture` determina a presença de textura no objeto. Caso exista uma textura, ela atribui o valor para a variável `vTextureCoordinates`, para ser lido posteriormente no *fragment shader*. Caso ela retorne `false`, é atribuído a cor do objeto para a variável `vColor`, que também será lida posteriormente no *fragment shader*. Por fim, a variável `gl_Position` recebe o resultado da multiplicação da matriz de projeção com a matriz de modelo e visão e a variável `aVertexPosition`, definindo o lugar em que o objeto deve aparecer na tela.

O *fragment shader* também precisa fazer verificações condicionais para decidir quais valores devem ser atribuídos às suas variáveis. O Quadro 8 exibe o seu código-fonte.

```
1. precision mediump float;
2.
3. varying vec2 vTextureCoordinates;
4. varying vec3 vLightWeighting;
5. varying vec4 vColor;
6.
7. uniform sampler2D uSampler;
8. uniform bool uUseTexture;
9.
10.
11.   void main() {
12.       if(uUseTexture) {
13.           vec4 textureColor = texture2D(uSampler, vTextureCoordinates);
14.           gl_FragColor = vec4(vLightWeighting.rgb * textureColor.rgb,
                                textureColor.a);
15.       } else {
16.           gl_FragColor = vColor * vec4(vLightWeighting, 1.0);
17.       }
18.   }
```

Quadro 8 - *fragment shader* utilizado no motor de jogos

Assim como o *vertex shader*, o *fragment shader* precisa verificar se o objeto possui textura, através da variável `uUseTexture`. Caso a variável contenha o valor `true`, ele recupera a cor da textura correspondente a sua coordenada, através do método texture2D. Caso não exista uma textura definida, ele utilizará a variável `vColor` passada como parâmetro pelo *vertex shader* quando essa condição for verdadeira. Em ambos os casos ele faz a multiplicação da cor com a variável `vLightWeighting`, que contém as informações a respeito da cor da luz. A variável `vLightWeighting` também é atribuída no *vertex shader*.

Além dos *shaders* configurados, é necessário abstrair as chamadas da API do WebGL. A classe `Scene` é a responsável por centralizar e orquestrar essas chamadas, através do método `draw`. Quando esse método é invocado, o objeto da classe `Scene` obtém o contexto WebGL através do método `initGL` e configura os *shaders* pré-configurados através do método `initShaders`. O Quadro 9 exibe o código-fonte do método `iniShaders`.

```
98.   initShaders : function() {
99.       var fragmentShader = this.getShader("fs");
100.      var vertexShader = this.getShader("vs");
101.      this.shaderProgram = this.gl.createProgram();
102.      var gl = this.gl;
103.      var program = this.shaderProgram;
104.
105.      gl.attachShader(program, vertexShader);
106.      gl.attachShader(program, fragmentShader);
107.      gl.linkProgram(program);
108.
109.      if (!gl.getProgramParameter(program, gl.LINK_STATUS)) {
110.          alert("Could not initialise shaders");
111.      }
112.
113.      gl.useProgram(program);
114.
115.      program.vertexPosition = gl.getAttribLocation(program,
                                                   "aVertexPosition");
116.      gl.enableVertexAttribArray(program.vertexPosition);
117.
118.      program.vertexNormal = gl.getAttribLocation(program,
                                                   "aVertexNormal");
119.      gl.enableVertexAttribArray(program.vertexNormal);
120.
121.      program.vertexTexture = gl.getAttribLocation(program,
                                               "aTextureCoordinates");
122.      gl.enableVertexAttribArray(program.vertexTexture);
123.
124.      program.uSampler = gl.getUniformLocation(program, "uSampler");
125.      program.uUseLighting = gl.getUniformLocation(program,
                                                   "uUseLighting");
126.      program.uPMatrix = gl.getUniformLocation(program, "uPMatrix");
127.      program.uNMatrix =  gl.getUniformLocation(program, "uNMatrix");
128.      program.uMVMatrix = gl.getUniformLocation(program, "uMVMatrix");
129.      program.uUseTexture = gl.getUniformLocation(program,
                                                   "uUseTexture");
130.  }
```

Quadro 9 - Código-fonte do método `initShaders`

O método inicia recuperando os *shaders* definidos através do método `getShader`. Esse método recebe o tipo do *shader* como parâmetro e recupera o arquivo do servidor, criando o tipo adequado através do método `gl.createShader`. Após isso, o método cria um programa e vincula os *shaders* carregados a esse programa, que será armazenado no atributo `shaderProgram`. Por fim, o método informa ao contexto WebGL que ele deve utilizar o programa criado e termina vinculando as variáveis presentes nos *shaders* aos atributos do objeto `shaderProgram`. Esses atributos fazem a comunicação do código JavaScript com o código GLSL, permitindo a troca de valor entre ambos.

Com o término da execução do método `initShaders`, o objeto `Scene` inicia a configuração de luz da cena. Inicialmente é feita uma verificação para descobrir se há alguma luz atribuída a cena. Caso não haja, é atribuído o valor `false` para a variável

`shaderProgram.useLightingUniform`, que irá comunicar ao *vertex shader* que ele pode utilizar a cor branca para trabalhar com a iluminação. Se existir uma luz na cena, o método `draw` desta luz é chamado. O Quadro 10 mostra a implementação deste método.

```
15.    draw : function(gl, shaderProgram) {
16.        var uLightPosition = gl.getUniformLocation(shaderProgram,
                                                    "uLightPosition");
17.        var uAmbientLightColor = gl.getUniformLocation(shaderProgram,
                                                    "uAmbientLightColor");
18.        var uDiffuseLightColor = gl.getUniformLocation(shaderProgram,
                                                    "uDiffuseLightColor");
19.        var uSpecularLightColor = gl.getUniformLocation(shaderProgram,
                                                    "uSpecularLightColor");
20.        gl.uniform3fv(uLightPosition, this.lightPosition);
21.        gl.uniform3fv(uAmbientLightColor, [this.ambientColor.getRed(),
22.                                            this.ambientColor.getGreen(),
23.                                            this.ambientColor.getBlue()]);
24.        gl.uniform3fv(uDiffuseLightColor, [this.diffuseColor.getRed(),
25.                                            this.diffuseColor.getGreen(),
26.                                            this.diffuseColor.getBlue()]);
27.        gl.uniform3fv(uSpecularLightColor, [this.specularColor.getRed(),
28.                                            this.specularColor.getGreen(),
29.                                           this.specularColor.getBlue()]);
30.    }
```

Quadro 10 - Método `draw` da classe `Light`

Quando esse método é invocado, o motor de jogos faz a associação entre as variáveis relacionadas à luz presentes no *vertex shader* com as variáveis criadas declaradas em JavaScript. Essa relação é feita através do método `gl.getUniformLocation`, que faz com que as variáveis em JavaScript apontem para a posição de memória que se encontram as variáveis do *vertex shader*. Com as variáveis relacionadas, ele atualiza os valores dessas variáveis, chamando o método `gl.uniform3fv` para cada uma delas.

Com os parâmetros de luz definidos, o motor de jogos inicia a configuração da câmera na cena. Primeiramente a câmera atual da cena é recuperada. Descoberta a câmera atual, a cena chama o método `draw` desta. O Quadro 11 exibe o código-fonte do método `draw` pertencente à classe `Camera`.

```
23.    draw : function(gl, shaderProgram) {
24.     gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
25.     mat4.perspective(this.fovY, this.aspectRatio,
                   this.nearPlaneDistance, this.farPlaneDistance, pMatrix);
26.     mat4.multiply(pMatrix, mat4.lookAt(
27.      [this.location.getX(), this.location.getY(), this.location.getZ()],
28.      [this.target.getX(), this.target.getY(), this.target.getZ()],
29.      [this.upVec.getX(), this.upVec.getY(), this.upVec.getZ()])
30.     );
31.    }
```

Quadro 11 - Método `draw` da classe `Camera`

A primeira instrução do método é garantir que o *viewport* do WebGL irá preencher toda a área do `canvas`. Isso é feito através da chamada ao método `gl.viewport`, que

especifica que a área a ser utilizada inicia em 0,0 e vai até `gl.viewportWidth`, `gl.viewportHeight`. Essas duas variáveis foram definidas no início do método `draw` da classe `Scene`, contendo a largura e a altura do elemento `canvas`, respectivamente. Com a área do *viewport* definida, é aplicada a projeção em perspectiva na matriz de projeção do motor de jogos, identificada pela variável `pMatrix`. Essa projeção é definida através do método `mat4.perspective`, presente no código da glMatrix. Por fim, a câmera é configurada para receber os parâmetros que identificam para onde ela está voltada. Isso é feito através do método `mat4.lookAt`, também disponível através da glMatrix.

Definidas as configurações de ambiente, a classe `Scene` começa a desenhar os objetos. Ela possui uma lista, chamada `objects`, que contém todos os seus filhos diretos. Ela percorre essa lista, chamando os filhos na ordem de inserção. Um dos tipos de objetos disponíveis para a criação é o cubo, representado pela classe Cube. O Quadro 12 exibe o código-fonte do método `draw`, presente na classe Cube.

O método inicia criando e configurando os *buffers* através do método `initBuffers`. A criação feita através da chamada ao método `gl.createBuffer` e `gl.bindBuffer`. Após criar o buffer são informados os dados que ele armazenará, através do método `gl.bufferData`. No caso da classe `Cube` podem ser criados três *buffers*: o *buffer* para as normais, para a textura e para a representação do cubo em si.

Após criar os buffers, o próximo passo verificar se a chamada ao método foi originada de um objeto-pai. É possível saber isso verificando se a variável matriz contém um valor diferente de `null`. Caso contenha, significa que a chamada originou-se de um objeto-pai, do contrário foi feita pela classe `Scene` e ela deve ser multiplicada pela matriz do objeto atual, para o grafo de cena funcionar corretamente. Essa técnica é necessária devido a falta dos métodos `pushMatrix` e `popMatrix` no WebGL, impedindo que o motor de jogos implemente uma pilha de matrizes.

Depois de garantir que a matriz recebeu as transformações aplicadas ao objeto-pai é feita a verificação da presença de textura no objeto. A variável `uUseTexture` do *vertex shader* é atualizada através do método gl.uniform1i. Caso haja textura, é feita a chamada para o método `initTexture` da classe `Texture`. Esse método mapeia as imagem contida no objeto da textura, permitindo que o WebGL a utilize para representar a cor dos objetos. Na sequência são o método atualiza a matriz de modelo e visão para receber os valores da matriz que foi atualizada anteriormente. Os valores dessa matriz são invertidos e o resultado dessa inversão é atribuído a matriz normal.

```
175.  draw : function(gl, shaderProgram, matrix) {
176.      this.initBuffers(gl);
177.
178.      var newMatrix;
179.
180.      if (matrix) {
181.          newMatrix = this.multiply(matrix, true);
182.      } else {
183.          newMatrix = this.matrix;
184.      }
185.
186.      gl.uniform1i(shaderProgram.uUseTexture, this.texture != null);
187.      if (this.texture) {
188.          this.texture.initTexture(gl, shaderProgram);
189.      }
190.
191.      gl.uniformMatrix4fv(shaderProgram.uMVMatrix, false, newMatrix);
192.      var normalMatrix = mat3.create();
193.      mat4.toInverseMat3(newMatrix, normalMatrix);
194.      mat3.transpose(normalMatrix);
195.      gl.uniformMatrix3fv(shaderProgram.uNMatrix, false, normalMatrix);
196.
197.      gl.bindBuffer(gl.ARRAY_BUFFER, this.cubeVertexPositionBuffer);
198.      gl.vertexAttribPointer(shaderProgram.vertexPosition, 3, gl.FLOAT,
                                 false, 0, 0);
199.
200.      gl.bindBuffer(gl.ARRAY_BUFFER, this.normalBuffer);
201.      gl.vertexAttribPointer(shaderProgram.vertexNormal, 3, gl.FLOAT,
                                 false, 0, 0);
202.
203.      if (this.texture) {
204.          gl.bindBuffer(gl.ARRAY_BUFFER,
                            this.cubeVertexTextureCoordinateBuffer);
205.          gl.vertexAttribPointer(shaderProgram.vertexTexture, 2,
                                       gl.FLOAT, false, 0, 0);
206.
207.          gl.activeTexture(gl.TEXTURE0);
208.          gl.bindTexture(gl.TEXTURE_2D, this.texture.texture);
209.      }
210.      gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER,
                        this.cubeVertexIndexBuffer);
211.      gl.drawElements(gl.TRIANGLES, 36, gl.UNSIGNED_SHORT, 0);
212.
213.      var objects = this.objects.size();
214.      for ( var i = 0; i < objects; i++) {
215.          this.objects.fromIndex(i).draw(gl, shaderProgram, newMatrix);
216.      }
217.  }
```

Quadro 12 - Método `draw` da classe `Cube`

O último passo é vincular o conteúdo dos *buffers* com as respectivas variáveis do *vertex shader*. Isso é feito através do método `gl.vertexAttribPointer`. O primeiro argumento a ser informado é a variável que será atualizada. Na sequência são informados o número de pontos por vértice e o tipo de dado em que eles se encontram. O quarto argumento especifica como devem ser tratados os valores que não foram do tipo `float`. Como o código da classe Cube determina que os vértices devam conter apenas valores do tipo `float`, é

passado o valor false. O quinto argumento informa o valor zero, informando que os dados estão armazenados em sequência e o último argumento especifica a posição em que os dados devem começar a ser lidos, que no caso da classe `Cube` sempre será a posição zero.

Após os buffers criados e configurados, referenciando as variáveis correspondentes do *vertex shader,* o método `draw` da classe `Cube` desenha a informação na cena, através da chamada ao método `gl.drawElements`. Por último, o método itera por todos os seus filhos, chamando o método `draw` de cada um deles. Os argumentos passados são os mesmos recebidos, com exceção da matriz, que passa a conter as transformações ocorridas no objeto atual.

3.3.3 Operacionalidade da aplicação

Esta seção apresenta um manual de uso do motor de jogos FWGL. Essas instruções permitem que o desenvolvedor inicie uma aplicação com os recursos disponíveis no motor de jogos. Para ilustrar as funcionalidades, foi desenvolvida uma aplicação exemplo, que possui um objeto cubo com textura, uma câmera e iluminação.

A primeira coisa que deve ser feita é preparar um arquivo HTML para receber as instruções do motor de jogos. Para isso é necessário importar todas as classes do FWGL no arquivo HTML em questão e declarar as *tags* HTML necessárias. O Quadro 13 exibe um arquivo HTML que importa as classes necessárias e cria o *canvas* utilizado pela cena O método inicia criando e configurando os *buffers* através do método `initBuffers`. A criação feita através da chamada ao método `gl.createBuffer` e `gl.bindBuffer`. Após criar o buffer são informados os dados que ele armazenará, através do método `gl.bufferData`. No caso da classe `Cube` podem ser criados três *buffers*: o *buffer* para as normais, para a textura e para a representação do cubo em si.

Após criar os buffers, o próximo passo verificar se a chamada ao método foi originada de um objeto-pai. É possível saber isso verificando se a variável matriz contém um valor diferente de `null`. Caso contenha, significa que a chamada originou-se de um objeto-pai, do contrário foi feita pela classe `Scene` e ela deve ser multiplicada pela matriz do objeto atual, para o grafo de cena funcionar corretamente. Essa técnica é necessária devido a falta dos métodos `pushMatrix` e `popMatrix` no WebGL, impedindo que o motor de jogos implemente uma pilha de matrizes.

Depois de garantir que a matriz recebeu as transformações aplicadas ao objeto-pai é feita a verificação da presença de textura no objeto. A variável `uUseTexture` do *vertex shader* é atualizada através do método gl.uniform1i. Caso haja textura, é feita a chamada para o método `initTexture` da classe `Texture`. Esse método mapeia as imagem contida no objeto da textura, permitindo que o WebGL a utilize para representar a cor dos objetos. Na sequência são o método atualiza a matriz de modelo e visão para receber os valores da matriz que foi atualizada anteriormente. Os valores dessa matriz são invertidos e o resultado dessa inversão é atribuído a matriz normal.

O método inicia criando e configurando os *buffers* através do método `initBuffers`. A criação feita através da chamada ao método `gl.createBuffer` e `gl.bindBuffer`. Após criar o buffer são informados os dados que ele armazenará, através do método `gl.bufferData`. No caso da classe `Cube` podem ser criados três *buffers*: o *buffer* para as normais, para a textura e para a representação do cubo em si.

Após criar os buffers, o próximo passo verificar se a chamada ao método foi originada de um objeto-pai. É possível saber isso verificando se a variável matriz contém um valor diferente de `null`. Caso contenha, significa que a chamada originou-se de um objeto-pai, do contrário foi feita pela classe `Scene` e ela deve ser multiplicada pela matriz do objeto atual, para o grafo de cena funcionar corretamente. Essa técnica é necessária devido a falta dos métodos `pushMatrix` e `popMatrix` no WebGL, impedindo que o motor de jogos implemente uma pilha de matrizes.

Depois de garantir que a matriz recebeu as transformações aplicadas ao objeto-pai é feita a verificação da presença de textura no objeto. A variável `uUseTexture` do *vertex shader* é atualizada através do método gl.uniform1i. Caso haja textura, é feita a chamada para o método `initTexture` da classe `Texture`. Esse método mapeia as imagem contida no objeto da textura, permitindo que o WebGL a utilize para representar a cor dos objetos. Na sequência são o método atualiza a matriz de modelo e visão para receber os valores da matriz que foi atualizada anteriormente. Os valores dessa matriz são invertidos e o resultado dessa inversão é atribuído a matriz normal.

```
<html>
	<head>
		<script src="../core/Setup.js"></script>
		<script src="../core/glMatrix-0.9.5.min.js"></script>
		<script src="../core/List.js"></script>
		<script src="../core/GraphicObject.js"></script>
		<script src="../core/Point4D.js"></script>
		<script src="../core/Scene.js"></script>
		<script src="../core/Light.js"></script>
		<script src="../core/Camera.js"></script>
		<script src="../core/Cube.js"></script>
		<script src="../core/Texture.js"></script>
		<script src="../core/XMLReader.js"></script>
		<script>
		    function startApp(){}
		</script>
	</head>
	<body onload="startApp()">
		<canvas id="myCanvas" height=500 width=500></canvas>
	</body>
</html>
```

Quadro 13 - Arquivo configurado para iniciar o desenvolvimento com o motor de jogos

O código a ser implementado deve ser inserido no método startApp, que será inicializado assim que o documento HTML estiver carregado no navegador web. No início do método é criado um objeto da classe `Scene`, representando a cena em que o desenvolvedor irá inserir seu cubo posteriormente. Também são criados os objetos que representam a luz e a câmera. O objeto luz irá receber quatro parâmetros, que indicam sua posição e cores, conforme modelo de Phong. Já a câmera irá receber três objetos da classe `Point4D` como parâmetro, informando sua localização e posição na cena, assim como o alvo para onde ela está voltada. Após os objetos referentes ao ambiente serem criados, eles precisam ser conectados à cena. Isso é feito chamando os método `addTexture` e `addLight`, presente na classe `Scene`. O Quadro 14 exibe o código que cria esses objetos e os adiciona à classe cena.

```
//cria a cena
var scene = new Scene();
//cria a luz
var lightPos = [0.0, 0.0, 1.0];
var ambientColor = new Color(0.5, 0.5, 0.5);
var diffuseColor = new Color(0.7, 0.7, 0.7);
var specColor = new Color(0.8, 0.8, 0.8);
var light = new Light(lightPos, ambientColor, diffuseColor, specColor);
//cria a câmera
var location = new Point4D(3.0, 3.0, 5.0);
var target = new Point4D(0.0, 0.0, 0.0);
var upVec = new Point4D(0.0, 1.0, 0.0);
var camera = new Camera(location, target, upVec);
//adiciona a luz e a câmera à cena
scene.addCamera(camera);
scene.addLight(light);
```

Quadro 14 - Criação dos objetos da câmera e luz e adição deles à cena

O cubo e sua textura também devem ser criados, através das classes `Cube` e `Texture`, respectivamente. O cubo pode receber três parâmetros: os vértices que o representam, sua cor e uma identificação, porém apenas o primeiro parâmetro é obrigatório. Neste exemplo é utilizado apenas o parâmetro obrigatório, pois ao utilizar uma textura no cubo, a cor passada como parâmetro seria automaticamente descartada. Como temos apenas um objeto no mundo, uma identificação não se faz necessária. O construtor da classe `Texture` requer apenas um parâmetro, que representa o caminho da imagem no servidor. Com ambos os objetos criados, são feitos dois vínculos: inicialmente, se adiciona a textura no cubo que foi criado. Concluída essa parte, é adicionado o cubo na cena. O Quadro 15 exibe o código-fonte responsável pelas ações descritas neste parágrafo.

```
//cria os vertices do cubo
var vertices = [new Point4D(1.0,  1.0,  1.0), new Point4D(-1.0, 1.0, 1.0),
new Point4D(-1.0, -1.0,  1.0), new Point4D(1.0, -1.0,  1.0),
new Point4D(1.0,  1.0, -1.0), new Point4D(-1.0,  1.0, -1.0),
new Point4D(-1.0, -1.0, -1.0), new Point4D(1.0, -1.0, -1.0),
new Point4D(-1.0,  1.0,  1.0), new Point4D(-1.0,  1.0, -1.0),
new Point4D(-1.0, -1.0, -1.0), new Point4D(-1.0, -1.0,  1.0),
new Point4D(1.0,  1.0,  1.0), new Point4D(1.0, -1.0,  1.0),
new Point4D(1.0, -1.0, -1.0), new Point4D(1.0,  1.0, -1.0),
new Point4D(1.0,  1.0,  1.0), new Point4D(1.0,  1.0, -1.0),
new Point4D(-1.0,  1.0, -1.0), new Point4D(-1.0,  1.0,  1.0),
new Point4D(1.0, -1.0,  1.0), new Point4D(1.0, -1.0, -1.0),
new Point4D(-1.0, -1.0, -1.0), new Point4D(-1.0, -1.0,  1.0)];
//cria o cubo e texture adiciona a texture ao cubo o cubo à cena
var cube = new Cube(vertices);
var texture = new Texture("crate.gif");
cube.setTexture(texture);
scene.addObject(cube);
```

Quadro 15 - Criação do cubo e textura

Finalmente, é necessário chamar o método `draw` da classe `Scene` para requisitar que o cubo seja desenhado na tela. O Quadro 16 exibe como recuperar o elemento *canvas* e passa-lo para a chamada do método `draw` da classe Scene.

```
var canvas = document.getElementById("myCanvas");
scene.draw(canvas);
```

Quadro 16 - Obtenção do elemento *canvas* e desenho da cena na tela

O código utilizado demonstra uma das duas formas possíveis de se criar uma cena. Ainda é possível importar um arquivo XML que armazena o conteúdo desta cena. Para fazer isso é necessário criar um objeto da classe `XMLReader`. Através desse objeto é feita a chamada para o método `parse`, que recebe como parâmetro uma `String` contendo o endereço do arquivo no servidor. O quadro demonstra o código necessário para fazer a importação da cena a partir do arquivo XML.

```
var canvas = document.getElementById("myCanvas");
var scene = new XMLReader().parse("example.xml", canvas);
scene.draw(canvas);
```

Quadro 17 - Importação do XML através da classe XMLReader.

O arquivo XML é composto do mesmo conjunto de elementos pertencentes à cena criada anteriormente. A cena contém uma câmera e uma luz definida, possuindo também um objeto cubo nela. Os parâmetros de cada um desses objetos são declarados como atributos e os objetos em si são declarados como *tags*. O Quadro 18 exibe o XML que estrutura a cena importada no arquivo HTML.

```
<?xml version="1.0"?>
<!DOCTYPE scene SYSTEM "fwgl.dtd">
<scene description="exemplo">
	<camera type="" description="camera">
		<position x="3.0" y="3.0" z="5.0" />
		<target x="0.0" y="0.0" z="0.0" />
		<up_vector x="0.0" y="1.0" z="0.0" />
	</camera>
	<light>
		<lightPosition x="0.0" y="0.0" z="1.0" />
		<ambientColor red="0.5" green="0.5" blue="0.5" />
		<diffuseColor red="0.7" green="0.7" blue="0.7" />
		<specularColor red="0.8" green="0.8" blue="0.8" />
	</light>
	<node>
		<cube texture="crate.gif">
			<position x="1.0" y="1.0" z="1.0" />
			<position x="-1.0" y="1.0" z="1.0" />
			<position x="-1.0" y="-1.0" z="1.0" />
			<position x="1.0" y="-1.0" z="1.0" />
			<position x="1.0" y="1.0" z="-1.0" />
			<position x="-1.0" y="1.0" z="-1.0" />
			<position x="-1.0" y="-1.0" z="-1.0" />
			<position x="1.0" y="-1.0" z="-1.0" />
			<position x="-1.0" y="1.0" z="1.0" />
			<position x="-1.0" y="1.0" z="-1.0" />
			<position x="-1.0" y="-1.0" z="-1.0" />
			<position x="-1.0" y="-1.0" z="1.0" />
			<position x="1.0" y="1.0" z="1.0" />
			<position x="1.0" y="-1.0" z="1.0" />
			<position x="1.0" y="-1.0" z="-1.0" />
			<position x="1.0" y="1.0" z="-1.0" />
			<position x="1.0" y="1.0" z="1.0" />
			<position x="1.0" y="1.0" z="-1.0" />
			<position x="-1.0" y="1.0" z="-1.0" />
			<position x="-1.0" y="1.0" z="1.0" />
			<position x="1.0" y="-1.0" z="1.0" />
			<position x="1.0" y="-1.0" z="-1.0" />
			<position x="-1.0" y="-1.0" z="-1.0" />
			<position x="-1.0" y="-1.0" z="1.0" />
		</cube>
	</node>
</scene>
```

Quadro 18 - Representação de uma cena em um arquivo XML

Ao final do processo descrito, um cubo será desenhado na tela. O cubo possui uma textura definida e está iluminado por uma luz, conforme os parâmetros definidos no arquivo XML ou através da codificação dos objetos manualmente. Também há uma câmera posicionada ligeiramente a direita do objeto, fazendo com que seja possível contemplar o efeito 3D apresentado pela cena. O resultado obtido pode ser observado através da Figura 18.

Figura 18 - Desenho exibido na tela após o código finalizado

3.3.4 Visualizador de grafo de cena

Com o objetivo de validar os recursos desenvolvidos no motor de jogos FWGL, foi criado um cenário especial, que representa um visualizador de grafo de cena. A motivação para criar esse cenário levou em conta dois aspectos: todas as funcionalidades disponíveis no motor de jogos seriam testadas e ele poderia ser utilizado para fins didáticos, visando facilitar o aprendizado de conceitos da computação gráfica.

O visualizador de grafo de cena carrega um arquivo do servidor previamente configurado e permite que o usuário edite seus parâmetros, modificando a cena de acordo com os novos valores atribuídos. Ele é representado através de um arquivo HTML e possui dois elementos *canvas,* sendo que ambos estão configurados para apresentar a mesma cena, porém através de duas câmeras diferentes. O usuário consegue alterar os parâmetros da primeira

câmera, sendo que a segunda tem seus valores fixados inicialmente e não pode ser alterada após o arquivo XML ser carregado. A Figura 19 exibe a aplicação sendo executada no navegador web.

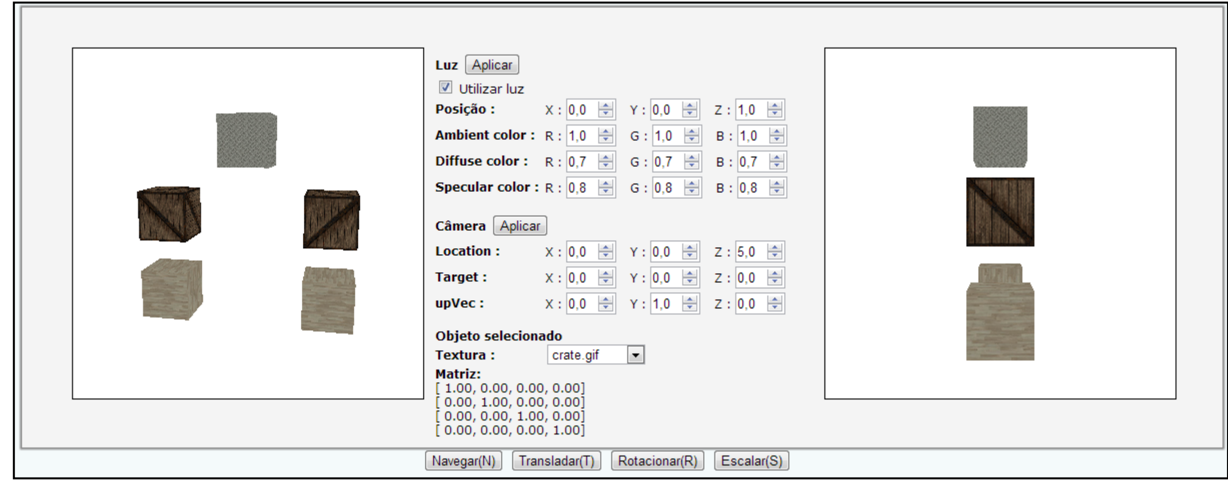


Figura 19 - Visualizador de grafo de cena carregado

Existem três tipos de objetos configuráveis no visualizador, sendo eles a luz, a câmera e o objeto selecionado. A luz pode ter quatro atributos alterados: o primeiro deles é a posição em que ela se encontra na cena e os três restantes definem a cor ambiente, difusa e especular, respectivamente. A câmera pode ser alterada através de três atributos diferentes: o primeiro, assim como no caso da luz, define a localização em que a câmera estará posicionada na cena. O segundo parâmetro indica para onde a câmera estará direcionada e, por final, é definida sua orientação na cena.

Por fim, é possível manipular os objetos da cena, sempre alterando os valores do objeto selecionado. Para definir o objeto selecionado é necessário estar no modo de navegação. Para entrar no modo de navegação, é necessário pressionar o botão Navegar, ou a tecla N. Com o modo de navegação ativado, o usuário deve pressionar a tecla direcional para baixo para navegar para o objeto-filho e a tecla direcional para cima para retornar ao objeto-pai. Para navegar entre os objeto-irmãos utilizam-se as teclas direcionais esquerda e direita.

Com o objeto selecionado, é possível aplicar quatro alterações sobre ele. A primeira alteração diz respeito à textura: é possível definir uma textura na lista disponível ou remover a textura atual, passando a exibir a cor que foi definida como padrão pelo desenvolvedor. Também é possível transladar o objeto, ativando o recurso através do botão Transladar ou pressionando a tecla T. As transformações geométricas referentes a rotação e escala também estão disponível através do botão Rotacionar e Escalar, respectivamente. Ambos também podem ser ativados pelas teclas de atalho, onde R faz a rotação e S faz a escala.

Para facilitar a visualização dos dados sendo transformados, também é exibido o

estado atual da matriz pertencente ao objeto atualmente selecionado. A cada transformação aplicada sobre ela, os valores são atualizados e o usuário pode verificar quais foram as mudanças que ocorreram na cena.

3.4 RESULTADOS E DISCUSSÃO

O presente trabalho teve como objetivo desenvolver um motor de jogos desenvolvido através da API do WebGL, no qual houvesse suporte a funcionalidades como grafo de cena, transformações geométricas, luz e importação de objetos de outros aplicativos, entre outras.

O desenvolvimento foi baseado na arquitetura utilizada pelo V-ART, que foi adaptada quando conveniente para se adequar a linguagem JavaScript. Também foram necessárias algumas adaptações no código devido ao fato de o V-ART implementar a renderização em OpenGL, enquanto o WebGL é baseado no OpengGL ES.

O modelo de desenvolvimento seguido na implementação do motor de jogos se assemelhou ao processo incremental da engenharia de software: cada objetivo específico representava um marco a ser atingido, recebendo prioridade os que eram dependência de outros objetivos. Primeiramente foram desenvolvidos o grafo de cena, as transformações geométricas e o gerenciador de objetos, seguidos pela implementação da câmera, iluminação e leitura de arquivos externos.

Houveram duas modificações relacionadas aos objetivos inicialmente definidos. Originalmente, a importação de arquivos deveria seguir um padrão pré-estabelecido. Pelo fato de o FWGL ser baseado na arquitetura do V-ART, optou-se por utilizar o modelo deste para fazer a importação dos arquivos. Seguindo o padrão estabelecido pelo V-ART, o requisito seria atendido, porém verificou-se necessário fazer alterações neste padrão para atender as adaptações feitas no FWGL, causando assim a necessidade de alterar o objetivo. Dos padrões adotados pelo V-ART, apenas foi implementada a importação dos arquivos XML que definem o grafo de cena, deixando pendente a parte referente a arquivos OBJ (Wavefront) definidos dentro dos arquivos XML do grafo de cena. Também ficou pendente a importação dos arquivo XML que define a animação.

Além da alteração no objetivo de importação de arquivos, também foi adicionado um novo objetivo, referente à adição de texturas no motor de jogos. Considerou-se relevante a adição de texturas como um objetivo por ser uma funcionalidade altamente utilizada no

mundo dos jogos e aplicações que envolvem computação gráfica em geral. As texturas foram implementadas conforme o esperado, porém possuem um problema na sua primeira execução: a imagem que representa a textura não é corretamente carregada, fazendo com que o objeto apareça com a cor preta definida (Figura 20). Caso o usuário recarregue a página, o problema desaparece. Outro fato a ser observado em relação às texturas é que cada objeto pode conter apenas uma textura.

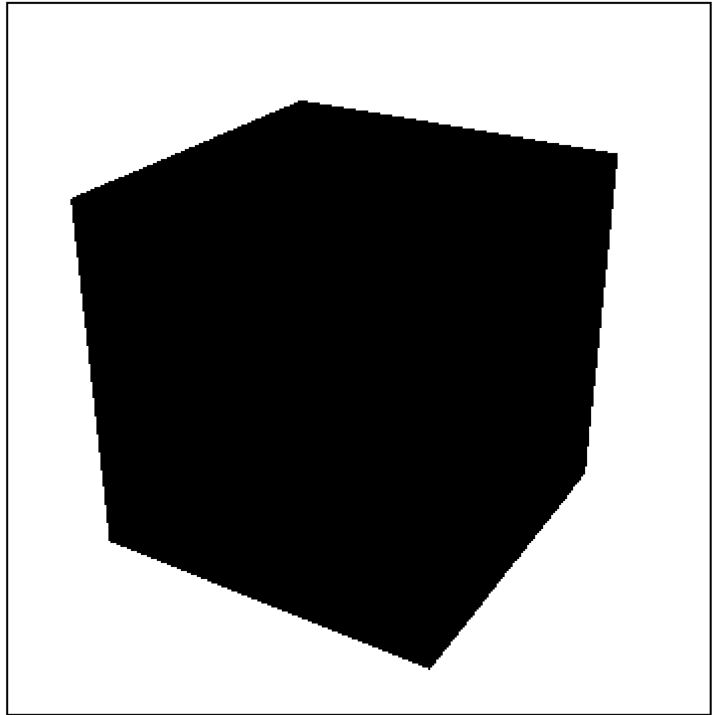


Figura 20 - Problema que ocorre nas texturas ao abrir a tela pela primeira vez

Outro objetivo que foi cumprido, porém de maneira limitada, foi o de iluminação. O requisito foi atendido conforme esperado, permitindo com que haja uma luz na cena, porém múltiplas luzes não são permitidas. A implementação atual do código contém apenas o modelo de reflexão de Phong, não disponibilizando outros modelos, limitando as possibilidades do desenvolvedor que utilizará o motor de jogos.

Também foram feitos testes visando medir o desempenho obtido pela aplicação, permitindo fazer uma análise dos pontos a serem melhorados com mais precisão. Estes testes são abordados na seção 3.4.1.

3.4.1 Testes de desempenho

Para a execução dos testes de desempenho foi criado um cenário (Figura 21) no qual é representado um grafo de cena. Este grafo de cena é constituído por objetos da classe `Cube`,

sendo estruturados como uma árvore binária, onde cada objeto pai possui dois objetos filhos.

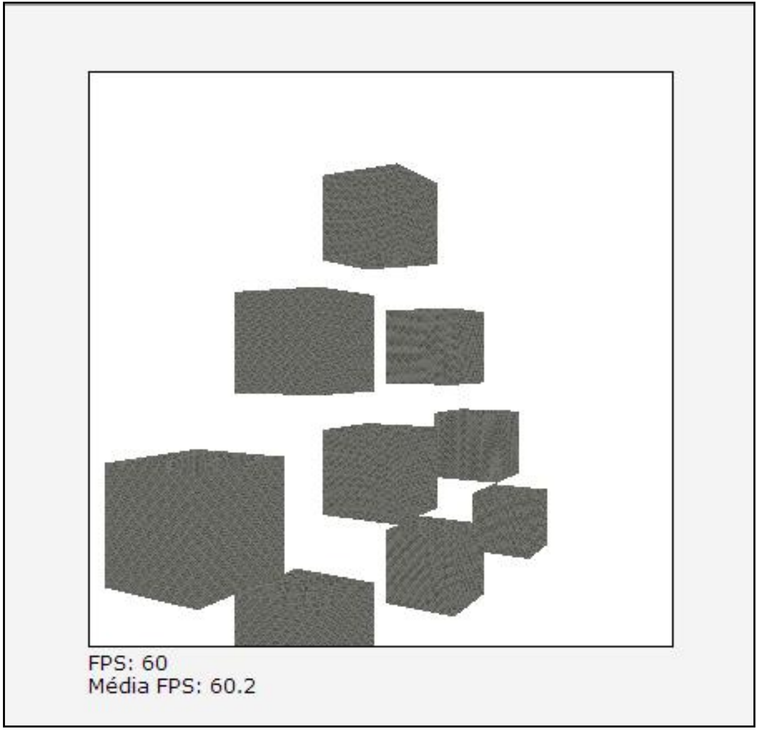


Figura 21 – Cenário utilizado nos testes de desempenho

Inicialmente foram feitos testes utilizado o navegador Google Chrome, iniciando com uma árvore de altura zero, representada por um objeto, até uma árvore com altura dez, representada por 2047 objetos. Os dados coletados são a média de *Frames Per Second* (FPS) dos 10 primeiros segundos de execução da cena. A Tabela **1** exibe os dados obtidos durante a análise.

Tabela 1 - Número de quadros por segundo por número de objetos na cena

Quantidade de Objetos	Quadros por segundo (FPS)
1	60
3	59,1
7	58,2
15	51
31	26,3
63	13,6
127	7,4
255	4,2
511	2
1023	1,4
2047	1,1

Como pode ser observado, ao executar o cenário com 31 objetos, ocorre uma queda de aproximadamente cinquenta por cento do FPS. Para cenários com mais de sessenta e três objetos a cena se torna inviável de ser utilizada devido ao baixo número de FPS apresentado.

A Figura 22 exibe o gráfico com os resultados obtidos.

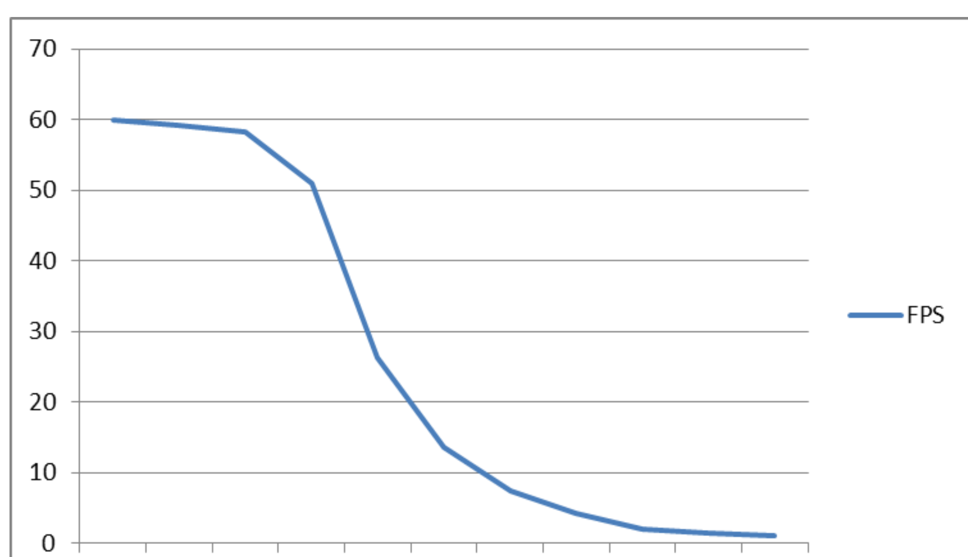


Figura 22 - Gráfico medindo o número de quadros por segundo obtidos pela aplicação

A partir das medições realizadas, percebeu-se que eram necessárias melhorias no desempenho do motor de jogos para que cenários complexos pudessem ser executados em tempo aceitável. Para fazer a análise dos métodos com pior desempenho, utilizou-se o Profiler, recurso disponível no Chrome Developer Tools, ferramenta disponibilizada em conjunto com o Google Chrome. A Figura 23 exibe o resultado obtido ao utilizar a ferramenta.

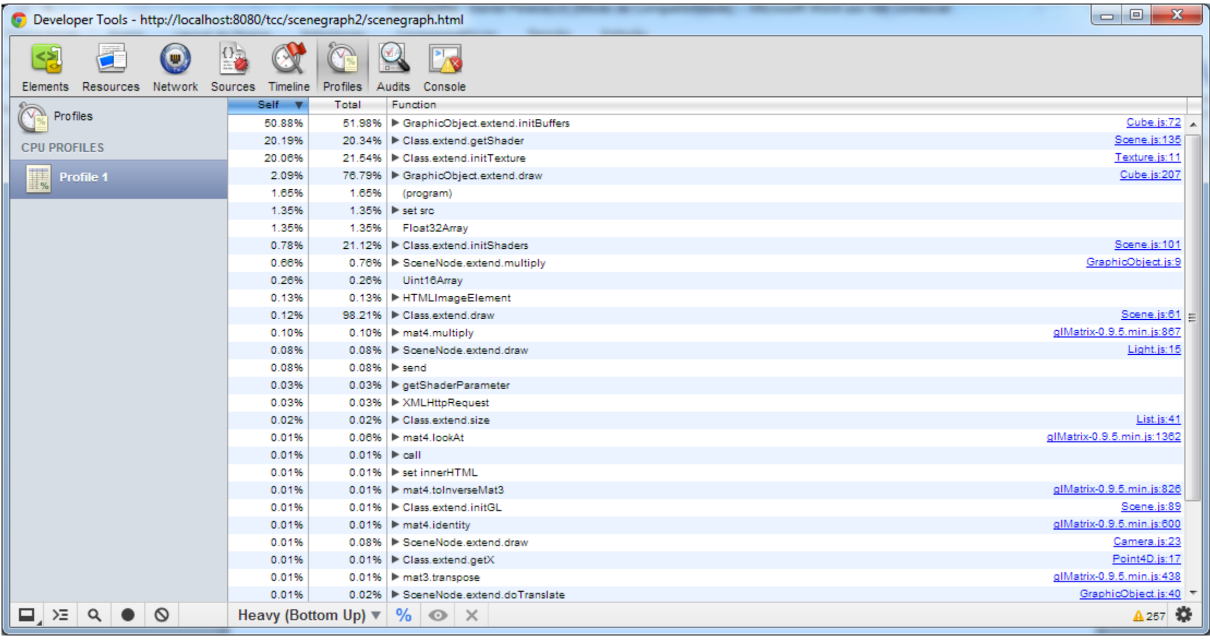


Figura 23 - Relatório da execução do *profile* da aplicação

Ao analisar o relatório gerado pelo *profiler*, percebeu-se que três métodos ocupavam 92,23 % do processamento, sendo eles os métodos `initBuffers`, `getShader` e `initTexture`.

Esses métodos sempre eram chamados a cada nova execução. Fazendo uma análise do código, verificou-se que os três métodos sempre irão gerar o mesmo conteúdo enquanto utilizarem elemento *canvas*. Constatado isso, o método `draw` da classe `Scene` foi alterado para verificar se o elemento *canvas* a ser utilizado na cena ainda é o mesmo utilizado anteriormente. Caso positivo, são utilizados os mesmos objetos já criados, caso contrário os objetos são recriados.

Com essa alteração foi feita uma nova coleta de dados. O cenário reproduzido foi o mesmo cenário do qual haviam sido extraídos os primeiros dados. Essa nova coleta representa o resultado final obtido com a aplicação, sendo feita em três diferentes navegadores web: Google Chrome, Mozilla Firefox e Opera. Os dados obtidos são apresentados na Tabela 2.

Tabela 2 - Número de quadros por segundo para os três navegadores testados

Quantidade de objetos	FPS obtido (Chrome)	FPS obtido (Firefox)	FPS obtido (Opera)
1	60,3	60,7	59,1
3	60,3	60,5	58,9
7	60,3	60,5	58,6
15	60,3	60,5	58,6
31	60,3	60,5	58,5
63	60,3	60,5	46,5
127	60,3	60,5	25,3
255	60,2	60,2	14,8
511	57,5	57,3	7
1023	31,6	32,6	4
2047	17,8	17,7	2,1

Ao analisar a tabela é possível perceber que houve um ganho significativo no desempenho da aplicação em relação à primeira versão. Todos os cenários passaram a executar com aproximadamente 60 FPS, com exceção os dois últimos. O resultado obtido no Chrome e no Firefox foi muito similar, porém ao executar a aplicação no Opera houve uma queda grande ao executar cenário com cento e vinte e sete objetos ou mais. A Figura 24 ilustra a diferença de resultado entre os três navegadores.

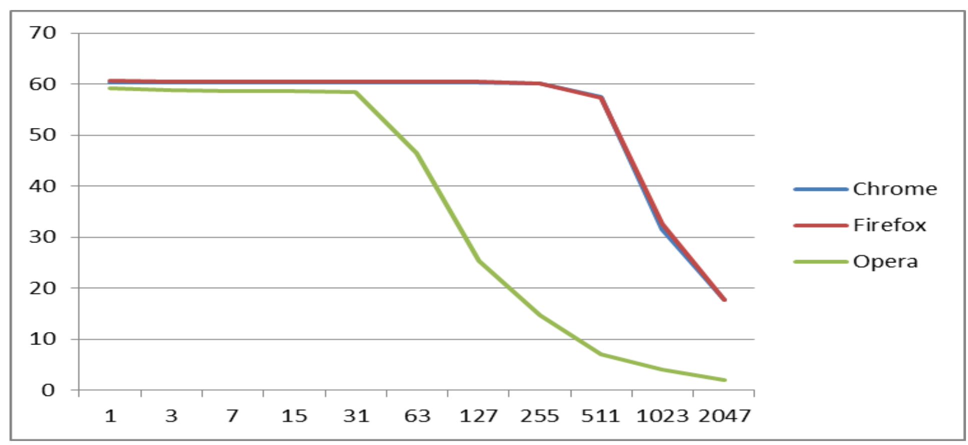


Figura 24 - Dados comparativos entre o resultado dos três navegadores testados

3.4.2 Comparativo entre o trabalho desenvolvido e os trabalhos correlatos

Nesta seção são apresentadas e discutidas as principais características deste trabalho, comparando-o com os trabalhos correlatos. Essa comparação está ilustrada no Quadro 19.

Características / trabalhos correlatos	CopperLicht	CubicVR	Three.js	FWGL
Gerenciador de objetos	x	x	x	x
Grafo de cena	x		x	x
Textura	x	x	x	x
Iluminação	x	x	x	x
Leitura de cena através de arquivos	x		x	x
Editor de mundo	x			
Abstração do WebGL			x	x
Suporte a vários renderizadores			x	

Quadro 19 - Características dos trabalhos correlatos e dos trabalhos desenvolvidos

O motor de jogos construído possui suporte as principais funcionalidades dos trabalhos correlatos, com exceção de duas: edição de mundo e suporte a vários renderizadores. A edição de mundo é uma funcionalidade atendida apenas pelo Copperlicht e possui um valor bastante significativo, tendo em vista que o FWGL tem como público-alvo desenvolvedores com pouca experiência em WebGL. O suporte aos renderizadores estava fora do escopo do trabalho, que tem seu foco em WebGL, porém também tem um grande valor na sua implementação.

O Quadro 19 demonstra que o principal diferencial do FWGL é reunir todas as principais funcionalidades que os trabalhos correlatos, permitindo que o desenvolvedor não tenha conhecimento específico em WebGL.Essa diferença é importante pois o WebGL é uma tecnologia recente e ainda em construção, fazendo com que poucos desenvolvedores tenham o conhecimento necessário para utilizá-la.

Na comparação com os trabalhos, o Three.js é o trabalho que mais se assemelha em termos de funções implementadas, contemplando as mesmas funcionalidades que o FWGL. O Three.js é um motor de jogos criado antes da existência do WebGL, fazendo dele um motor de jogos mais amadurecido que os demais. Quando este trabalho foi iniciado, sua implementação para WebGL ainda está em fase de desenvolvimento, não tendo suporte a grande parte das funcionalidades mencionadas, porém estas foram finalizadas durante o desenvolvimento deste trabalho.

4 CONCLUSÕES

Este trabalho apresentou um motor de jogos desenvolvido em WebGL e uma aplicação que o utiliza através da representação visual de um grafo de cena.

O WebGL, principal API utilizada no desenvolvimento, ainda está sendo concebida e, devido a esse motivo, ainda há pouca bibliografia disponível. Os navegadores web que a suportam podem apresentar comportamentos diferentes em algumas ocasiões, fazendo com que a experiência do usuário seja diferente dependendo do navegador utilizado. Nas aplicações desenvolvidas para testar o motor de jogos, estas diferenças ficaram mais perceptíveis no desenvolvimento da iluminação do ambiente, porém com as novas versões dos navegadores o comportamento passou a ser similar.

A arquitetura adotada como solução pelo motor de jogos é simples e intuitiva, fazendo com que o desenvolvedor não necessite de conhecimentos em APIs de baixo nível da WebGL e GLSL para programar seu jogo ou aplicativo da área de computação gráfica. A arquitetura adotada foi baseada na arquitetura do V-ART, sendo necessárias algumas adaptações devido a diferenças entre a linguagem JavaScript e C++, porém a estrutura das classes principais foi mantida.

O motor de jogos mostrou um bom desempenho na execução dos testes, conseguindo desenhar mais de mil objetos a mais de 30 FPS em dois dos três navegadores testados, sendo esse número o suficiente para o usuário não perceber a troca de quadros na cena. Ainda foi obtido o resultado de 60 FPS em cenas com 255 objetos ou menos, o que permite com que o usuário veja a cena de maneira mais natural, especialmente em cenas onde ocorre uma interação muito grande entre a aplicação e o usuário.

4.1 EXTENSÕES

Durante o desenvolvimento deste trabalho, observaram-se algumas possibilidades de trabalhos de extensão. São elas:

a) importar arquivos OBJ a partir do arquivo XML, seguindo o padrão adotado no V-ART;
b) importar arquivos que contenham a descrição da movimentação de um objeto na

cena, seguindo o padrão adota no V-ART;

c) criar os demais tipos de iluminação além do modelo de Phong;
d) transcrever o código do V-ART referente aos personagens articulados;
e) permitir que um objeto possua mais de uma textura ao mesmo tempo;
f) permitir que uma cena possua mais de uma luz adiciona a ela;
g) permitir que o desenvolvedor utilize seus próprios arquivos de *shader*;
h) abstrair as chamadas WebGL, permitindo que o motor desenhe outros tipos de API sem necessidade de alterar o código.

REFERÊNCIAS BIBLIOGRÁFICAS

ACOSTA, Darien. **WebGL demo** - three.js reflection template. [S.l.], 2012. Disponível em: <http://www.webgl.com/2012/07/webgl-demo-three-js-reflection-template/>. Acesso em: 5 nov. 2012.

AMBIERA. **CopperLicht -** JavaScript 3D engine using WebGL. [S.l.], 2011. Disponível em: <http://www.ambiera.com/copperlicht/>. Acesso em: 13 nov. 2012.

AMOROSO, Danilo. **O que é pixel shader?** [S.l.], 2008. Disponível em: <http://www.tecmundo.com.br/811-o-que-e-pixel-shader-.htm>. Acesso em: 13 nov. 2012.

ANYURU, Andreas. **Profession WebGL programming:** developing 3D graphics for the web. Chichester: Wrox, 2012.

AULA DE ANATOMIA. **Sistema articular - diartroses (sinovias)**. [S.l.], 2012. Disponível em: <http://www.auladeanatomia.com/artrologia/sinovias.htm>. Acesso em: 12 dez. 2012.

CASTRO, Maria A. S. de. **O que é world-wide web**. São Carlos, 2003. Disponível em: <http://www.icmc.usp.br/ensino/material/html/www.html>. Acesso em: 13 nov. 2012.

CUBIC. **CubicVR 3d engine**. [S.l.], 2011. Disponível em: <http://www.cubicvr.org/>. Acesso em: 13 nov. 2012.

EBERLY, David H. **3D game engine design**: a pratical approach to real-time computer graphics. San Diego: Morgan Kaufmann Publishers, 2001.

ECMA. **TC39 - ECMAScript**. [S.l.], 2011. Disponível em: <http://www.ecma-international.org/memento/TC39.htm>. Acesso em: 13 nov. 2012.

PAMPLONA, Vitor F. **Um protótipo de motor de jogos 3D para dispositivos móveis com suporte a especificação mobile 3D gaphics api for J2ME**. 2005. 83 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

GROSSKURTH, Alan; GODFREY, Michael W. **Architecture and evolution of the modern web browser**. Waterloo, 2006. Disponível em: <http://grosskurth.ca/papers/browser-archevol-20060619.pdf >. Acesso em: 13 nov. 2012.

KHRONOS. **OpenGL ES -** the standard for embedded accelerated 3D graphics. [S.l.], 2011a. Disponível em: <http://www.khronos.org/opengles/>. Acesso em: 13 nov. 2012.

______. **WebGL -** OpenGL ES 2.0 for the Web. [S.l.], 2011b. Disponível em: <http://www.khronos.org/webgl/>. Acesso em: 13 nov. 2012.

KHRONOS. **OpenGL shading language**. [S.l.], 2011c. Disponível em: <http://www.opengl.org/documentation/glsl/>. Acesso em: 13 nov. 2012.

LOMBARDI, Victor. **Internet use and growth.** [S.l.], 2011. Disponível em: <http://www.noisebetweenstations.com/personal/essays/audio_on_the_internet/InternetUse.html>. Acesso em: 13 nov. 2012.

MUNSHI, Aaftab; GINSBURG, Dan; SHREINER, Dave. **OpenGL ES 2.0 programming guide**. Stoughton: Addison-Wesley Professional, 2008.

PEREIRA, Daniel. **Desenvolvimento de um motor de jogos 3D, utilizando WebGL.** Blumenau, 2012. Disponível em: <http://www.inf.furb.br/gcg/projetos_pub/fwgl/>. Acesso em: 13 nov. 2012.

POZZER, Cesar T. **Grafo de cena**. [S.l.], 2007. Disponível em: <http://www-usr.inf.ufsm.br/~pozzer/disciplinas/cga_2_grafo_cena.pdf>. Acesso em: 13 nov. 2012.

PRADA, Rodrigo. **WebGL, Flash ou Unity:** quem dominará os jogos online em 3D? [S.l.], 2010. Disponível em: <http://www.tecmundo.com.br/6832-webgl-flash-ou-unity-quem-dominara-os-jogos-online-em-3d-.htm>. Acesso em: 13 nov. 2012.

RESIG, John. **Simple JavaScript inheritance.** [S.l.], 2008. Disponível em: <http://www.ejohn.org/blog/simple-javascript-inheritance/>. Acesso em: 5 nov. 2012.

SCHNEIDER, Bruno. **framework V-ART**. [S.l.], 2012. Disponível em: <http://algol.dcc.ufla.br/~bruno/v-art/>. Acesso em: 5 nov. 2012.

STEFANOV, Stoyan. **JavaScript patterns**. Sebastopol: O'Reilly Media, 2010.

VART. **V-ART:** virtual articulations for reality. [S.l.], 2007. Disponível em: <http://vart.codeplex.com/>. Acesso em: 5 nov. 2012.

W3C. **HTML5**. [S.l], 2011a. Disponível em: <http://www.w3.org/TR/html5/>. Acesso em: 13 nov. 2012.

______. **Document Object Model (DOM)**. [S.l.], 2011b. Disponível em: <http://www.w3.org/DOM/>. Acesso em: 13 nov. 2012.

W3SCHOOLS. **HTML5 introduction**. [S.l.], 2011. Disponível em: <http://www.w3schools.com/html5/html5_intro.asp>. Acesso em: 13 nov. 2012.

WALSH, Aaron E. **Understanding scene graphs**. [S.l.], 2002. Disponível em: <http:/drdobbs.com/java/184405094/>. Acesso em: 13 nov. 2012.

WARD, Jeff. **What is a game engine?** [S.l.], 2008. Disponível em: <http://www.gamecareerguide.com/features/529/what_is_a_game_.php?page=1>. Acesso em: 13 nov. 2012.

WILLIAMS, James L. **Learning HTML5 game programming:** a hands-on guide to building online games using Canvas, SVG, and WebGL. Chichester: Wrox, 2011.

ANEXO A - Modelo simples de herança de classes em JavaScript

O Quadro 20 demonstra o código desenvolvido por John Resig para simular o comportamento de classes em JavaScript. Entre as funcionalidades apresentadas, encontra-se a sobrecarga de métodos e a utilização de construtores, através do uso da função `init`.

```
/* Simple JavaScript Inheritance
 * By John Resig http://ejohn.org/
 * MIT Licensed.
 */
(function() {
	fnTest = /xyz/.test(function() {
		xyz;
	}) ? /\b_super\b/ : /.*/;
	this.Class = function() {
	};
	Class.extend = function(prop) {
		var _super = this.prototype;
		initializing = true;
		var prototype = new this();
		initializing = false;
		for ( var name in prop) {
			prototype[name] = typeof prop[name] == "function"
				&& typeof _super[name] == "function"
				&& fnTest.test(prop[name]) ? (function(name, fn) {
				return function() {
					var tmp = this._super;
					this._super = _super[name];
					var ret = fn.apply(this, arguments);
					this._super = tmp;
					return ret;
				};
			})(name, prop[name]) : prop[name];
		}
		function Class() {
			if (!initializing && this.init)
				this.init.apply(this, arguments);
		}
		Class.prototype = prototype;
		Class.prototype.constructor = Class;
		Class.extend = arguments.callee;
		return Class;
	};
})();
```

Quadro 20 - Código fonte do modelo simples de herança