UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

MJ3L: MOTOR DE JOGOS 3D EM LUA PARA IOS

FABIANO GUIZELLINI MODOS

BLUMENAU
2011

2011/1-17

FABIANO GUIZELLINI MODOS

MJ3L: MOTOR DE JOGOS 3D EM LUA PARA IOS

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. Paulo César Rodacki Gomes, Dr. – Orientador

BLUMENAU
2011

2011/1-17

MJ3L: MOTOR DE JOGOS 3D EM LUA PARA IOS

Por

FABIANO GUIZELLINI MODOS

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

Presidente:	Prof. Paulo César Rodacki Gomes, Dr. - Orientador, FURB
Membro:	Prof. Dalton Solano dos Reis, M.Sc. - FURB
Membro:	Prof. Mauro Marcelo Mattos, Dr. - FURB

Blumenau, 30 de junho de 2011

Dedico este trabalho a todos os meus familiares e amigos, especialmente aqueles que me ajudaram diretamente na realização deste.

AGRADECIMENTOS

A Deus, pelo seu imenso amor e graça.

Aos meus pais que são exemplos de vida pra mim, por colocarem os filhos como a maior prioridade da vida deles, por me ensinarem o valor de uma boa educação, da importância dos estudos, e também por sempre me incentivarem a lutar pelos meus sonhos.

Ao meu irmão, por estar sempre me ajudando, e também pelas dicas de como utilizar o *MacBook* que foram muito úteis para o desenvolvimento deste trabalho.

Aos meus amigos Brian Fruman e Gustavo Leyendecker, por compreenderem a minha ausência no trabalho nos últimos dias para dedicação na conclusão deste trabalho.

Ao grupo de futebol de quinta feira, por ser grandes amigos e um dos únicos lugares durante esse semestre que eu conseguia não pensar neste trabalho.

Ao meu orientador, Paulo César Rodacki Gomes, por ter acreditado e me ajudado na conclusão deste trabalho.

Make all you can, save all you can, give all you can.

John Wesley

RESUMO

Este trabalho apresenta o desenvolvimento de um motor de jogos para a criação de jogos 3D utilizando a linguagem de programação Lua na plataforma iOS, denominado motor de jogos 3D em Lua (MJ3L) para iOS. Este trabalho apresenta também como foi feito o desenvolvimento para suportar a execução do MJ3L no iOS, renderizando funções OpenGL ES e tratando eventos de toque na tela em Lua. O motor fornece uma estrutura de grafo de cena e movimentação de câmera no espaço 3D.

Palavras-chave: Motor de jogos. OpenGL ES. Lua. iOS.

ABSTRACT

This paper presents the development of a game engine for creating 3D games using the Lua programming language in the iOS platform, named motor de jogos 3D em Lua (MJ3L) for iOS. This work also shows how was the development to support the execution of the MJ3L in the IOS, rendering OpenGL ES functions and handling touch events on the screen in Lua. The engine provides a scene graph structure and moving camera in 3D space.

Key-words: Game engine. OpenGL ES. Lua. iOS.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

3D – Três Dimensões

API – *Application Programming Interface*

ES – *Embedded System*

IDE – *Integrated Development Environment*

OpenAL - *Open Audio Library*

OpenGL – *Open Graphics Library*

RF – Requisito Funcional

RNF – Requisito Não Funcional

SDK – *System Development Kit*

RGB – *Red, Green* e *Blue*

SUMÁRIO

1 INTRODUÇÃO

O iPhone é um telefone celular com comunicação *wireless* que possui várias funcionalidades, entre elas um gravador de videos em alta definição, navegador *web*, tocador de música e vídeo. A interface com o usuário permite multi-toque e teclado virtual. Atualmente existem mais de 200 mil aplicações para o iPhone desenvolvidas por terceiros (APPLE, 2010).

O iOS (iPhone OS) é o sistema operacional do iPhone. Com ele, jogos podem ser desenvolvidos e gráficos podem ser renderizados utilizando a OpenGL ES. Conforme Khronos (2010), OpenGL ES é uma API de gráficos 3D para sistemas embarcados e dispositivos móveis.

As funções de renderização dos gráficos de um jogo devem ser invocadas por um motor de jogos. Segundo Ward (2008), o conceito de um motor de jogos é relativamente simples. Existe para abstrair detalhes de processos comuns dos jogos, como renderização, interação com o usuário, física e outros. Desta forma, os desenvolvedores podem dar mais foco aos detalhes específicos do jogo que estão desenvolvendo.

Atualmente somente o produto comercial *Corona Game Edition* (CORONA, 2010) fornece uma ferramenta com motor de jogo e acesso as funções do OpenGL ES puramente em Lua para as plataformas iPhone e Android. Lua destaca-se pela sua simplicidade, portabilidade, rapidez e pela facilidade com que se pode embutir um interpretador Lua em uma aplicação C. Além disso, Lua é a única linguagem criada em um país em desenvolvimento a ganhar relevância global (KEPLER, 2008, p. 4). A linguagem é inteiramente projetada, implementada e desenvolvida no Brasil, por uma equipe na PUC-Rio (LUA, 2010).

Baseado na quantidade de aplicações desenvolvidas por terceiros para o iPhone e de não existir nenhuma ferramenta *open source* em Lua para desenvolvimento de jogos para iPhone, o presente trabalho propõe-se a desenvolver um motor de jogos 3D em Lua para a plataforma iPhone. Desta forma, pretende-se oferecer aos desenvolvedores uma ferramenta para desenvolvimento de jogos puramente na linguagem Lua. A proposta também contempla a criação de uma biblioteca para desenvolvimento em OpenGL ES 1.1 utilizando Lua.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver um motor de jogos 3D para iPhone na linguagem de *script* Lua.

Os objetivos específicos deste trabalho podem ser sumarizados da seguinte maneira:

a) disponibilizar um *framework* para desenvolver aplicações para iPhone em OpenGL ES 1.1 utilizando a linguagem de *script* Lua;
b) disponibilizar um grafo de cena para gerenciamento de renderização dos objetos;
c) disponibilizar métodos para tratamento de eventos;
d) disponibilizar movimentação de câmera no ambiente 3D.

1.2 ESTRUTURA DO TRABALHO

Este trabalho é formado por quatro capítulos. O capítulo 2 apresenta as tecnologias envolvidas, as partes do motor de jogos e apresenta os trabalhos correlatos. No capítulo 3 é abordado a especificação do motor de jogos desenvolvido e apresenta sua operacionalidade para o desenvolvimento de aplicativos. O capítulo 4 foca-se em apresentar as conclusões gerais e possíveis futuras implementações.

2 FUNDAMENTAÇÃO TEÓRICA

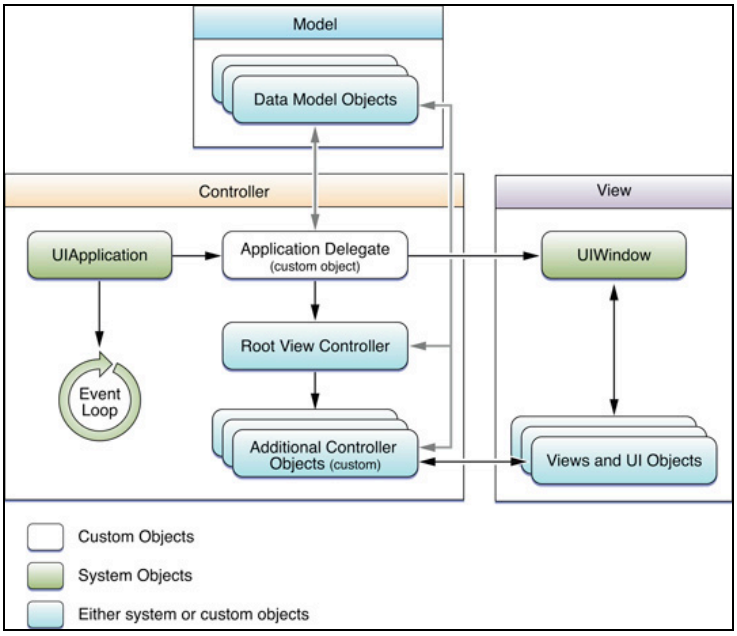
A seção 2.1 apresenta a plataforma iPhone OS. Na seção 2.2 é apresentado o OpenGL ES. Na seção 2.3 é apresentada a linguagem Lua. Na seção 2.4 é apresentado o conceito de um motor de jogos. Na seção 2.5 são apresentados trabalhos correlatos.

2.1 IPHONE OS

O iOS (iPhone OS) é o sistema operacional do iPhone. Toda aplicação do iOS é construída utilizando a biblioteca UIKIT, que fornece os objetos necessários para executar a aplicação, coordernar as entradas do usuário, e exibir o conteúdo na tela. Além disso é responsável por rastrear eventos de toque e movimento, e distribui-los para as outras partes da aplicação (IOS, 2010).

A renderização de objetos e gráficos no iOS pode ser feita utilizando as bibliotecas nativas, como a UIKIT. Outra forma de renderização é por meio da utilização da biblioteca OpenGL ES. Para essa é preciso criar uma superfície com bibliotecas nativas, onde os gráficos criados pelos OpenGL ES possam ser renderizados (IOS, 2010).

Segundo IOS (2010), desde o momento em que a aplicação é iniciada pelo usuário e enquato ela estiver em execução, o UIKit gerencia a maioria do comportamento da aplicação. Por exemplo, uma aplicação iOS recebe eventos continuamente do sistema e deve responder estes eventos. O recebimento de eventos é o serviço do objeto UIApplication, mas a resposta dos eventos é responsabilidade do código customizado. Relacionamento similar existe em outras partes da aplicação, com os objetos do sistema gerenciando os processos em geral e o código customizado responsável pela implementação do comportamento especifico da aplicação. A Figura 1 apresenta como os objetos da UIKit trabalham com o código customizado e o Quadro 1 descreve a função de cada objeto. Os objetos na cor branca são os objetos customizados, na cor verde os objetos de sistema e na cor azul os objetos de sistema ou customizados.



Fonte: IOS (2010).

Figura 1 – Objetos da UIKit e customizados

Objeto	**Descrição**
UIApplication	O objeto UIApplication gerencia o *event loop* e coordena outros comportamentos de alto nível da aplicação.
Application delegate	O objeto *application delegate* é um objeto customizado que é provido para a aplicação no começo da execução. O objetivo principal deste objeto é inicializar a aplicação e apresentar sua janela na tela do dispositivo.
Data model	Os objetos *Data model* armazenam o conteúdo específico da aplicação, por exemplo informações do banco de dados.
View controller	Os objetos *View controller* gerenciam a apresentação do conteúdo da aplicação.
UIWindow	O objeto UIWindow coordena a apresentação de uma ou mais visões na tela do dispositivo. A maioria das aplicações possuem apenas uma janela.
View control	Views e controls provém a representação visual do conteúdo da aplicação.

Fonte: IOS (2010).

Quadro 1 – Detalhamento dos objetos da UIKit e customizados

Todo evento enviado pelo iOS para uma aplicação é encapsulado em um único objeto de evento do tipo `UIEvent`. No caso de eventos relacionados a toque na tela, o evento contém um ou mais objetos do tipo `UITouch` (IOS, 2010).

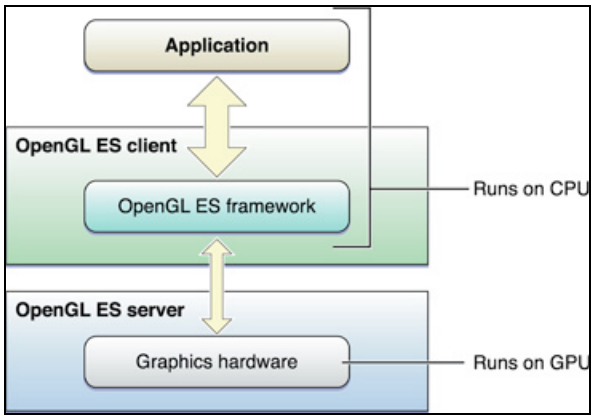
2.2 OPENGL ES

Conforme Wright (2000, p. 32), OpenGL é uma API que possui mais de 200 comandos e funções. Estes comandos são usados para desenhar gráficos primitivos como pontos, linhas, e polígonos em três dimensões. Além disso, OpenGL suporta iluminação, sombreamento, mapeamento de texturas, transparência e muitos outros efeitos especiais.

OpenGL ES é uma API multi-plataforma com funcionalidades para trabalhar com gráficos 2D e 3D em sistemas embarcados e dispositivos móveis. Ela é uma sub-coleção do OpenGL, fornecendo uma interface poderosa e flexível de baixo nível para software e aceleração gráfica (KHRONOS, 2010).

OpenGL ES 1.1 é definida em relação à especificação OpenGL 1.5 e salienta a aceleração de hardware da API. Ela oferece maior funcionalidade, qualidade de imagem melhorada e otimizações para aumentar o desempenho, reduzindo consumo de banda de memória para economizar energia (KHRONOS, 2010).

Segundo IOS OPENGLES (2010), OpenGL ES usa um modelo de cliente servidor, conforme demonstrado na Figura 2. Quando a aplicação invoca funções OpenGL ES, ela se comunica com o cliente OpenGL ES. O cliente processa a função chamada e quando necessário converte o comando para o OpenGL ES *server*. Este tipo de modelo permite que o processo de uma função seja divida entre o cliente e o servidor. A natureza do cliente, servidor, e o caminho de comunicação entre eles é especificado em cada implementação do OpenGL ES.



Fonte: IOS (2010).

Figura 2 – Modelo cliente servidor do OpenGL ES

A especificação do OpenGL ES define o funcionamento da API, mas não define funções para gerenciar a interação entre o OpenGL ES e a infraestrutura de hardware que da suporte ao sistema operacional. A especificação espera que cada implementação forneça

funções para alocarem um contexto de renderização e um sistema de *framebuffer*.

2.3 LINGUAGEM LUA

Lua é uma linguagem de programação poderosa, rápida e leve, projetada para estender aplicações. A linguagem combina sintaxe simples para programação procedural com poderosas construções para descrição de dados baseadas em tabelas associativas e semântica extensível. Lua é tipada dinamicamente, é interpretada a partir de *bytecodes* para uma máquina virtual baseada em registradores, e tem gerenciamento automático de memória com coleta de lixo incremental. Estas características fazem dela uma linguagem ideal para configuração e prototipagem rápida (LUA, 2011). No Quadro 2 é demonstrado um código em Lua que utiliza tabelas. O exemplo mostra uma tabela sendo usada para adicionar 1000 itens numéricos e posteriormente mostra a mesma tabela sendo indexada por caracteres.

```
a = {} -- tabela vazia
-- cria 1000 entradas novas
for i=1,1000 do a[i] = i*2 end
print(a[9]) -- > 18
a[“x”] = 10
print(a[“x”]) -- > 10
print(a[“y”]) -- > nil
```

Fonte: Ierusalimschy (2003, p. 14).

Quadro 2 – Exemplo de código utilizando tabela em Lua

Aplicações que utilizam Lua normalmente possuem uma parte do código escrito em linguagem C e outra em Lua propriamente dita. De acordo com Ierusalimschy (2003, p. 275), Lua é uma linguagem interpretada e extensível que possui duas formas de comunicação com código escrito em C. No primeiro tipo, a parte escrita em linguagem C tem o controle da aplicação e o código Lua fica encapsulado em uma biblioteca. O código C neste tipo de interação é chamado de *application code*. No segundo tipo, Lua tem o controle da aplicação e o código C é encapsulado em uma ou mais bibliotecas. Neste caso, o código C é chamado de *library code*. Ambos casos utilizam a C API para fazer a comunicação entre o código C e Lua.

O Quadro 3 é apresenta um exemplo de uma função em C que pode ser invocado por um código em Lua. A função em questão retorna o seno do número passado por parâmetro, são apresentadas também as funções para empilhar a função e associar a função a uma variável global. Dessa forma para acessar esta função em Lua, basta invocar dentro do código

Lua a função `mysin(d)`.

```
static int l_sin (lua_State L) {
        double d = lua_tonumber(L, 1)/*pega o argumento*/
        lua_pushnumber(L, sind(d)); /*empilha resultado*/
        return 1;
}
/*Empilha o valor do tipo da função */
lua_pushcfunction(l, l_sin);
/*Associa isto a variavel global mysin */
lua_setglobal(l, "my_sin");
```

Fonte: Ierusalimschy (2003, p. 224).

Quadro 3 – Exemplo de código utilizando a C API

De acordo com TIOBE (2011), a popularidade global de Lua considerando todas as linguagens de programação aumentou 0.61% entre o período de maio de 2010 e maio de 2011, conquistando oito posições e subindo pra décima segunda colocação no *ranking* das linguagens mais populares. As linguagens C# e Objective-C com um crescimento de 2.76% e 2.65% foram as únicas linguagens que tiveram um crescimento maior que Lua neste período, conforme é apresentado na Figura 3.

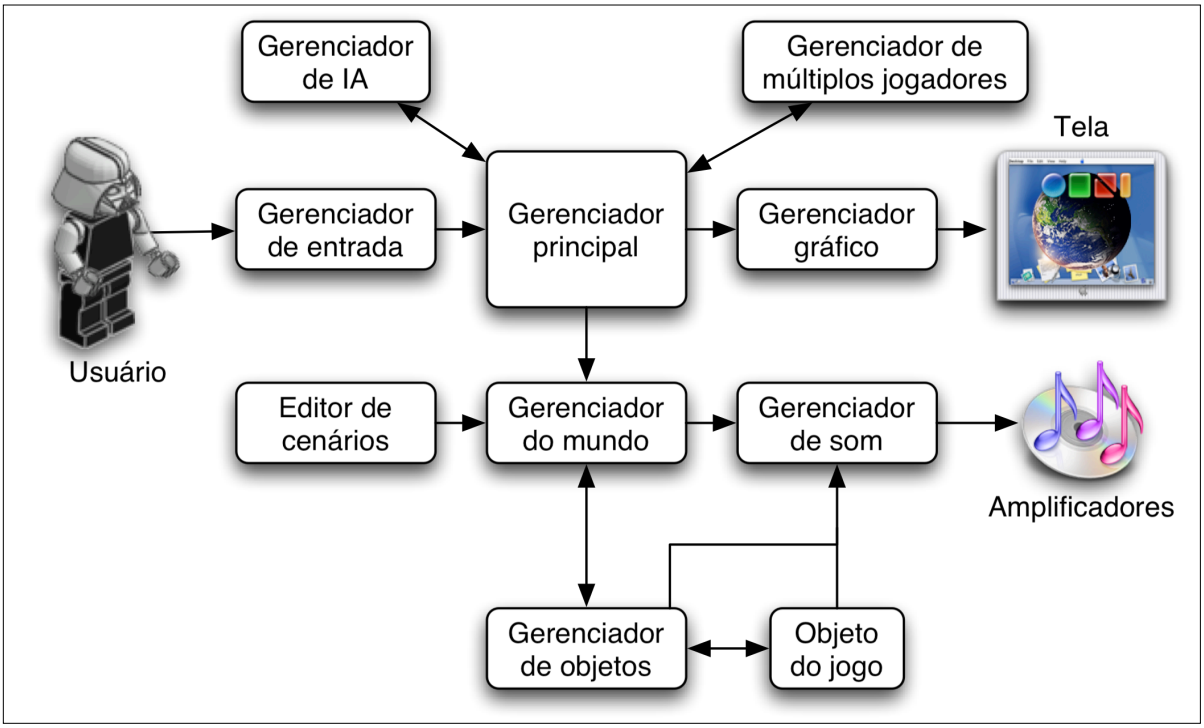
Position May 2011	Position May 2010	Delta in Position	Programming Language	Ratings May 2011	Delta May 2010	Status
1	2	↑	Java	18.160%	+0.20%	A
2	1	↓	C	16.170%	-2.02%	A
3	3	=	C++	9.146%	-1.23%	A
4	6	↑↑	C#	7.539%	+2.76%	A
5	4	↓	PHP	6.508%	-2.57%	A
6	10	↑↑↑↑	Objective-C	5.010%	+2.65%	A
7	7	=	Python	4.583%	+0.49%	A
8	5	↓↓↓	(Visual) Basic	4.496%	-1.16%	A
9	8	↓	Perl	2.231%	-1.05%	A
10	11	↑	Ruby	1.421%	-0.67%	A
11	12	↑	JavaScript	1.394%	-0.69%	A
12	20	↑↑↑↑↑↑↑↑	Lua	1.102%	+0.61%	A
13	9	↓↓↓↓	Delphi	1.073%	-1.49%	A
14	-	=	Assembly	1.042%	-	A
15	16	↑	Lisp	0.953%	+0.30%	A
16	23	↑↑↑↑↑↑↑	Ada	0.747%	+0.32%	A
17	15	↓↓	Pascal	0.709%	-0.02%	A
18	21	↑↑↑	Transact-SQL	0.697%	+0.21%	B
19	-	=	Scheme	0.580%	-	B
20	25	↑↑↑↑↑	RPG (OS/400)	0.503%	+0.09%	B

Fonte: TIOBE (2011).

Figura 3 – *Ranking* TIOBE de linguagens de programação

2.4 MOTOR DE JOGO

Segundo Gomes e Pamplona (2005, p. 2), motores de jogos são bibliotecas de desenvolvimento responsáveis pelo gerenciamento do jogo, das imagens, do processamento de entrada de dados e outras funções. A idéia é que os motores implementem funcionalidades e recursos comuns à maioria dos jogos de determinado tipo, permitindo que esses recursos sejam reutilizados a cada novo jogo criado. A Figura 4 exibe a arquitetura típica de um motor de jogos 3D.



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

Figura 4 – Arquitetura de um motor de jogos

O gerenciador de entrada recebe e identifica os eventos de entrada e os envia para o gerenciador principal. O gerenciador gráfico transforma o modelo que define o estado atual do jogo em uma visualização para o usuário. O gerenciador de inteligência artificial gerencia o comportamento dos objetos desenvolvidos pelo *designer* do jogo. O gerenciador de múltiplos jogadores trata a comunicação dos jogadores, independentemente do meio físico em que se encontram. O gerenciador de objetos carrega, controla o ciclo de vida, salva e destrói um grupo de objetos do jogo. O objeto do jogo possui dados relevantes para uma entidade que faça parte do jogo. Esta parte do motor controla a posição, velocidade, dimensão, detecção de colisão, entre outros. O gerenciador do mundo armazena o estado atual do jogo e para isso utiliza os gerenciadores de objetos. Por fim o gerenciador principal é responsável pela

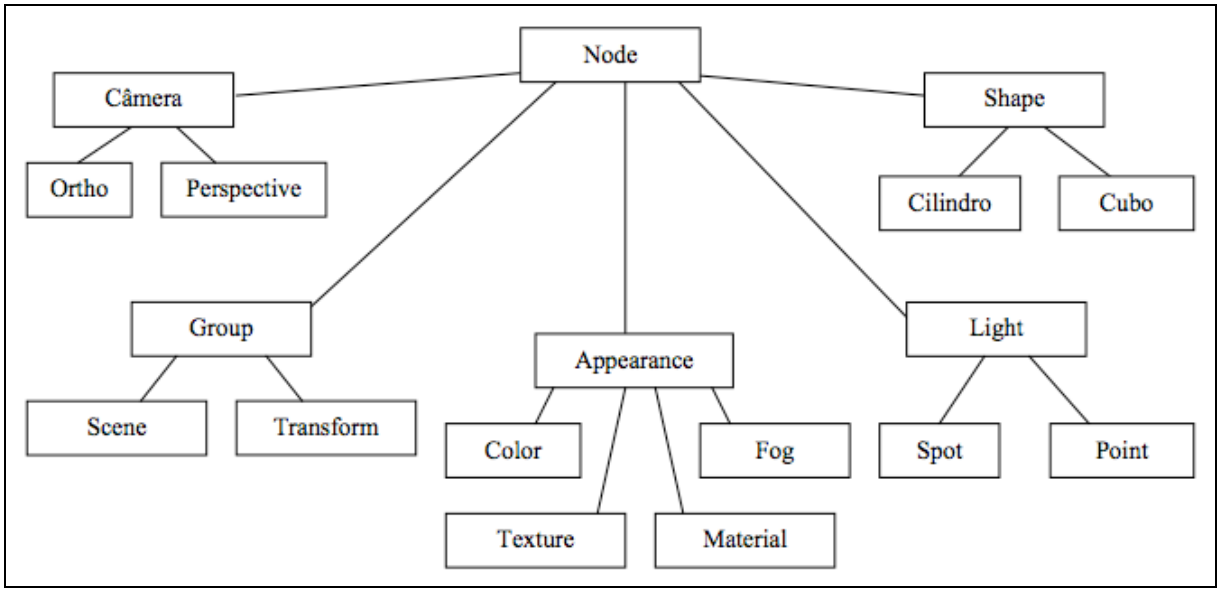
coordenação entre os demais componentes.

Motor de jogos oferece componentes reusáveis que podem ser manipulados para dar vida ao jogo. Carregando, mostrando, animando modelos, detectando colisão de objetos, física, inteligência artificial, todos podem ser componentes que formam um motor (WARD, 2008)

De acordo com Eberly (2001, p. xxvii) um motor de jogo é uma entidade complexa que consiste de mais que uma camada de renderização que desenha triângulos ou apenas uma coleção de técnicas desorganizadas. Um motor de jogo deve lidar com problemas do gerenciamento do grafo de cena, como o *front end* que disponibiliza as entradas de volta para o *renderer*, podendo ser um *renderer* baseado em software ou hardware.

2.4.1 Grafos de cena

Segundo Eberly (2001, p. 141), em um ambiente 3D com uma quantidade grande de objetos, o método mais simples para processar os objetos é agrupando-os em uma lista e iterando pelos itens para selecionar e renderizar. Apesar de ser um método simples, ele não é eficiente pois cada objeto do mundo vai ser verificado se pode ser renderizado. Um método mais eficiente para processar os objetos é agrupa-los hierarquicamente de acordo com sua localização espacial. 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, por exemplo agrupando entidades básicas, como caixas, esferas, cilindros, dentre outros, pode-se gerar objetos ou cenas mais complexos. (POZZER, 2007). Conforme demonstrado na Figura 5 a classe principal é a Node, de onde derivam todas as demais classes.



Fonte: Pozzer (2007).

Figura 5 – Estrutura hierárquica do grafo de cena

A estrutura de dados no formato árvore é a mais tradicional para agrupar os objetos. Os nós folha representam os objetos geométricos primitivos e os nós internos representam o agrupamento dos objetos relacionados.

2.5 TRABALHOS CORRELATOS

Em relação a trabalhos correlatos, devido ao fato de este trabalho apresentar um caráter pioneiro especialmente em relação às tecnologias empregadas, o *Corona Game Edition* (CORONA, 2010) foi o único projeto encontrado que oferece o mesmo tipo de ferramenta que o tema proposto. Foram selecionados também os projetos similares SIO2 (SIO2, 2011) e um motor de jogos 3D desenvolvido para iOS (TAKANO, 2009).

2.5.1 Corona game edition

O *Corona Game Edition* (CORONA, 2010) é uma ferramenta comercial para desenvolvimento de jogos baseada em linguagem Lua e em uma versão proprietária do OpenGL ES. Na data de elaboração do presente trabalho, o valor de sua licença de uso era de U$ 349,00 por ano. Trata-se de uma solução completa independente de plataforma para

desenvolvimento de aplicações para iPhone e Android. É possível escrever os códigos e executá-los em um simulador próprio da ferramenta.

Esta ferramenta possui um motor de jogos que pode executar em velocidade nativa nas plataformas iPhone e Android e faz uso das tecnologias de acelerômetros e interface multi-toque de ambas plataformas. Além disso, a ferramenta possui integração com o motor de física Box2D (BOX2D, 2010), onde é possível fazer simulação de física com qualquer objeto gráfico do Corona.

2.5.2 SIO2

O SIO2 (SIO2, 2011) é um motor de jogos comercial para iPad, iPhone e iPod Touch, podendo ser utilizado as linguagens de programação Objective-C, C++ e C para desenvolvimento. Este motor de jogos suporta a utilização da linguagem de *script* Lua para gerenciamento de cenas e tratamento de eventos.

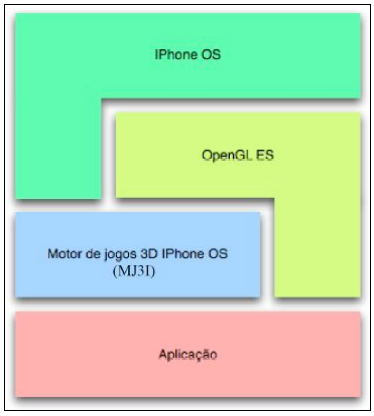
O motor é compatível com OpenGL ES, e possui um motor de física com dinâmica de corpos rígidos e não-rígidos, tratamento de colisão e sistemas de partículas. Além disso possui efeitos sonoros, mapeamento de texturas, iluminação e sombreamento, gerenciamento de cena, animação, e suporta também a importação de modelos 3D.

2.5.3 MJ3I: Um motor de jogos 3D para o iPhone OS

O MJ3I é um motor de jogos para criação de jogos 3D na plataforma iPhone. O motor é baseado na biblioteca OpenGL ES, para construção das formas geométricas e nas bibliotecas da plataforma iPhone para realização da Interação Homem Computador (IHC), através da tela que permite múltiplos toques. O motor permite que sejam desenvolvido jogos, sem a necessidade de implementar todas as rotinas básicas e necessárias de um jogo 3D. O motor não implementa rotinas mais complexas como cálculo de física, sistema de partículas ou mesmo o áudio (TAKANO, 2009).

Segundo Takano (2009, p. 25) o MJ3I é uma camada intermediária que se localiza entre a aplicação e o OpenGL ES, conforme Figura 6. Apesar da biblioteca disponibilizar rotinas para o desenvolvimento do jogo ainda é possível acessar o OpenGL ES diretamente.

Além de se basear no OpenGL ES para realizar a renderização, o motor de jogos também utiliza a biblioteca UIKit, disponibilizada pela própria Apple, que integrada com o iPhone OS possibilita a captura de toques na tela.



Fonte: Takano (2009, p. 25).

Figura 6 – Visão da camada da biblioteca do motor

3 DESENVOLVIMENTO

Este capítulo detalha as etapas do desenvolvimento da biblioteca. São apresentados os requisitos, a especificação e a implementação do mesmo, mencionando as técnicas e ferramentas utilizadas. Também são comentadas questões referentes à operacionalidade da biblioteca e os resultados obtidos.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

Os requisitos apresentados encontram-se classificados em Requisito Funcional (RF) e Requisito Não-Funcional (RNF), os quais são:

a) fornecer um *framework* para desenvolvimento em OpenGL ES 1.1 utilizando a linguagem Lua (RF);
b) possuir um grafo de cena para gerenciar os objetos hierarquicamente (RF);
c) fazer movimentação de câmera no ambiente 3D (RF);
d) fornecer métodos para tratar eventos de toque no iPhone (RF);
e) ser implementado na linguagem de programação Lua, C e Objective-C (RNF);
f) executar na plataforma iPhone, no simulador e no dispositivo (RNF);
g) utilizar a biblioteca OpenGL ES (RNF);
h) utilizar o XCode como ambiente de desenvolvimento (RNF).

3.2 ESPECIFICAÇÃO

A especificação da biblioteca em questão foi desenvolvida seguindo a análise orientada a objetos, utilizando a notação *Unified Modeling Language* (OMG, 2005). A representação lógica é apresentada através dos diagramas de casos de uso, componentes, classes e sequência, os quais foram confeccionados utilizando a ferramenta *Enterprise Architect* (EA, 2011).

3.2.1 Visão da biblioteca

A Aplicação utiliza o MJ3L (Motor de Jogos 3D em Lua) e o LOGLES (Lua OpenGL ES). O M3JL utiliza o LOGLES e o iPhone WAX. O LOGLES é uma camada intermediária entre a aplicação e o OpenGL ES. O iPhone Wax é uma camada intermediária entre o motor de jogos e o iPhone OS (Figura 7).

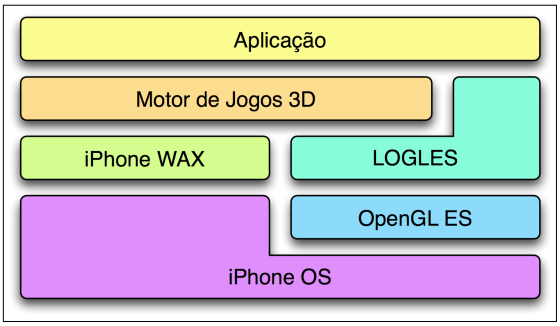


Figura 7 – Visão do motor e LOGLES na arquitetura

3.2.2 Diagrama de casos de uso

O diagrama na Figura 8 apresenta as principais ações que a aplicação terá acesso a biblioteca.

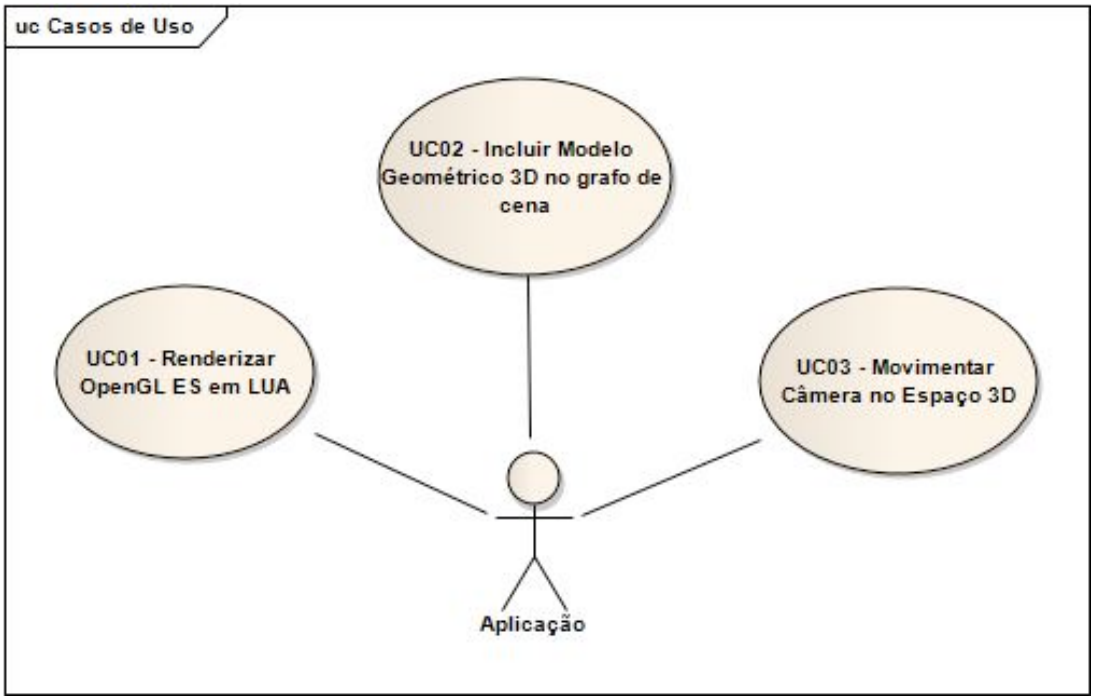


Figura 8 – Diagrama de casos de uso

No quadros 4, 5 e 6 são apresentados os detalhamentos de cada caso de uso da aplicação.

UC01 – Renderizar OpenGL ES em Lua	
Pré-condições	1) Instalar *framework* iPhone WAX 2) Adicionar API LOGLES a aplicação 3) Criar contexto de renderização do OpenGL ES e um *framebuffer*.
Cenário Principal	Invocar funções LOGLES no método de renderização
Pós condição	Aplicação renderiza funções invocadas

Quadro 4 – Detalhamento UC01 – Renderizar OpenGL ES em Lua

UC02 – Incluir Modelo Geométrico 3D no grafo de cena	
Pré-condições	Possuir acesso as classes GroupNode e Leaf
Cenário Principal	1) Procurar por um nó 2) Adicionar o modelo como folha do nó encontrado
Cenário Alternativo	No passo 1, não encontra nenhum nó 1) Criar um nó 2) Adicionar o modelo como folha do nó criado
Pós condição	Modelo adicionado ao grafo de cena

Quadro 5 – Detalhamento UC02 – Incluir Modelo Geométrico 3D no grafo de cena

UC03 – Movimentar Câmera no espaço 3D	
Pré-condições	Possuir acesso a classe Camera
Cenário Principal	Invocar método de visualização passando por parâmetro as coordenadas do ponto a ser vizualidado
Pós condição	Câmera foca visualização no ponto

Quadro 6 – Detalhamento UC03 – Movimentar Câmera no espaço 3D

3.2.3 Diagrama de componentes

O diagrama na Figura 9 apresenta os componentes externos utilizados pela biblioteca e como os objetos se relacionam entre os pacotes.

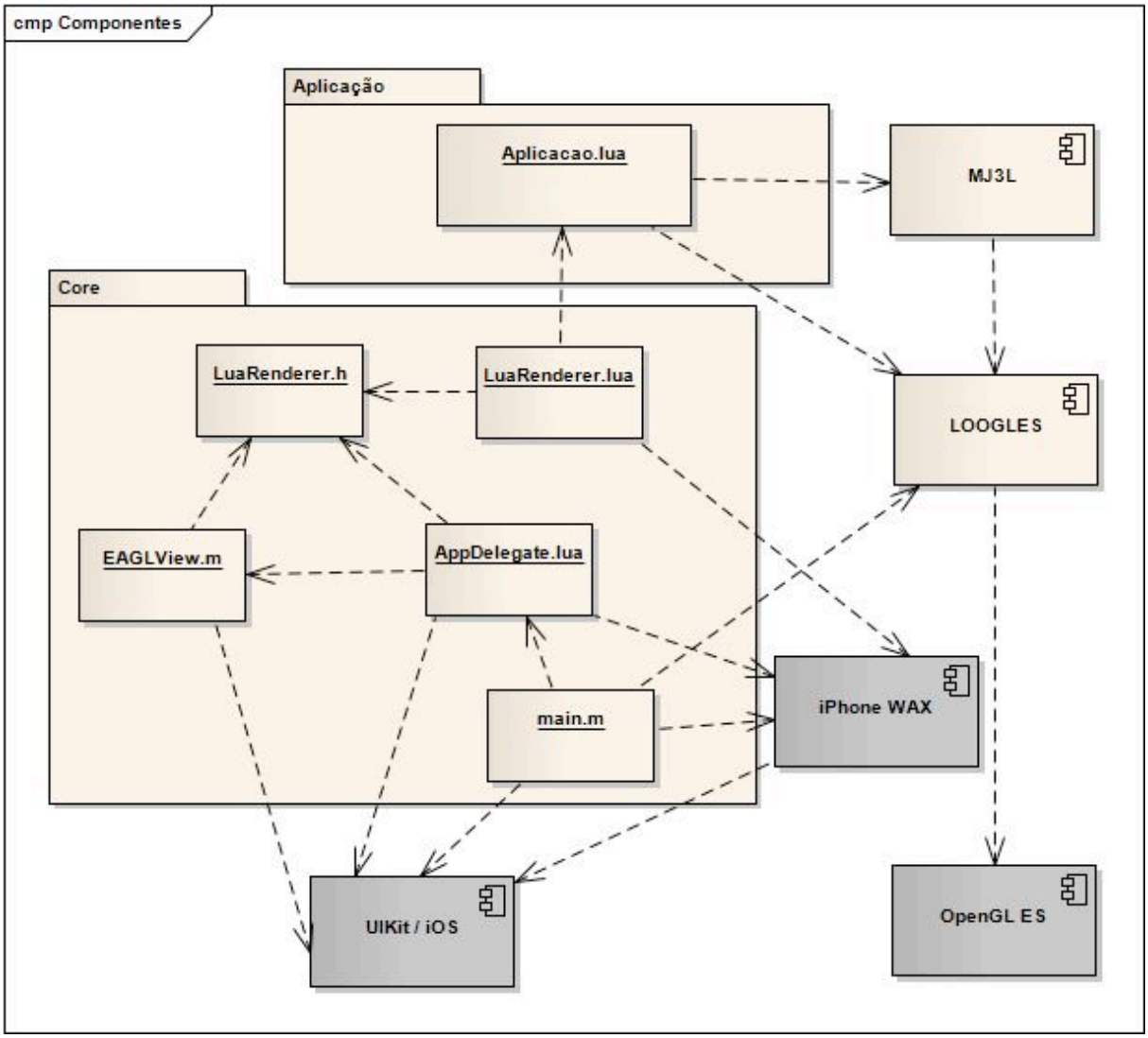


Figura 9 – Diagrama de componentes

3.2.3.1 Componentes Externos

Nesta seção são apresentados os componentes externos utilizados pela biblioteca, eles são os componentes iPhone Wax, UIKit/iOS e OpenGL ES, identificados pela cor cinza na Figura 5.

O *framework* UIKit fornece os objetos necessários para executar a aplicação, coordenar o tratamento de eventos de entrada de dados de toque na tela, e exibir o conteúdo na tela do iOS. O Wax iPhone é um *framework* para criar aplicações para o iPhone em Lua. Ele associa Objective-C e Lua usando o ambiente *runtime* do Objective-C, com esta ferramenta é possível fazer em Lua praticamente as mesmas coisas que são possíveis em Objective-C. O OpenGL ES é uma API na linguagem de programação C, e é utilizado para

renderização de gráficos 2D e 3D. A especificação do OpenGL ES não possui funções para criar um contexto de renderização e um *framebuffer* onde é exibido o resultado dos comandos de desenho. O iOS fornece o objeto `EAGLContext` para criar um contexto OpenGL ES e também um *framebuffer* para exibir os gráficos.

3.2.3.2 LOGLES

LOGLES é uma adaptação da API LuaGL (LUAGL, 2011) para suportar OpenGL ES 1.1. LuaGL é uma API *open source* desenvolvida na linguagem de programação C que fornece acesso às funcionalidades da API OpenGL em Lua.

Para o desenvolvimento do LOGLES foi preciso alterar os códigos originais do LuaGL, removendo as funções não implementadas pela versão OpenGL ES 1.1, adicionando as funções que não estavam implementadas e substituindo todas as variáveis do tipo double para float, porque o iOS não suporta o formato double. O apendice A fornece a listagem das funções OpenGL ES 1.1 do LOGLES.

3.2.3.3 Core e Aplicação

O pacote Core possui os objetos dependentes do UIKit e iPhone WAX que são responsáveis por iniciar a aplicação, criar o contexto de renderização e tratar eventos. O pacote Aplicação contém o *script* que representa a aplicação e tem acesso aos componentes MJ3L e LOGLES para desenvolver jogos e renderizar OpenGL ES. A Figura 10 apresenta o diagrama de classes desses pacotes.

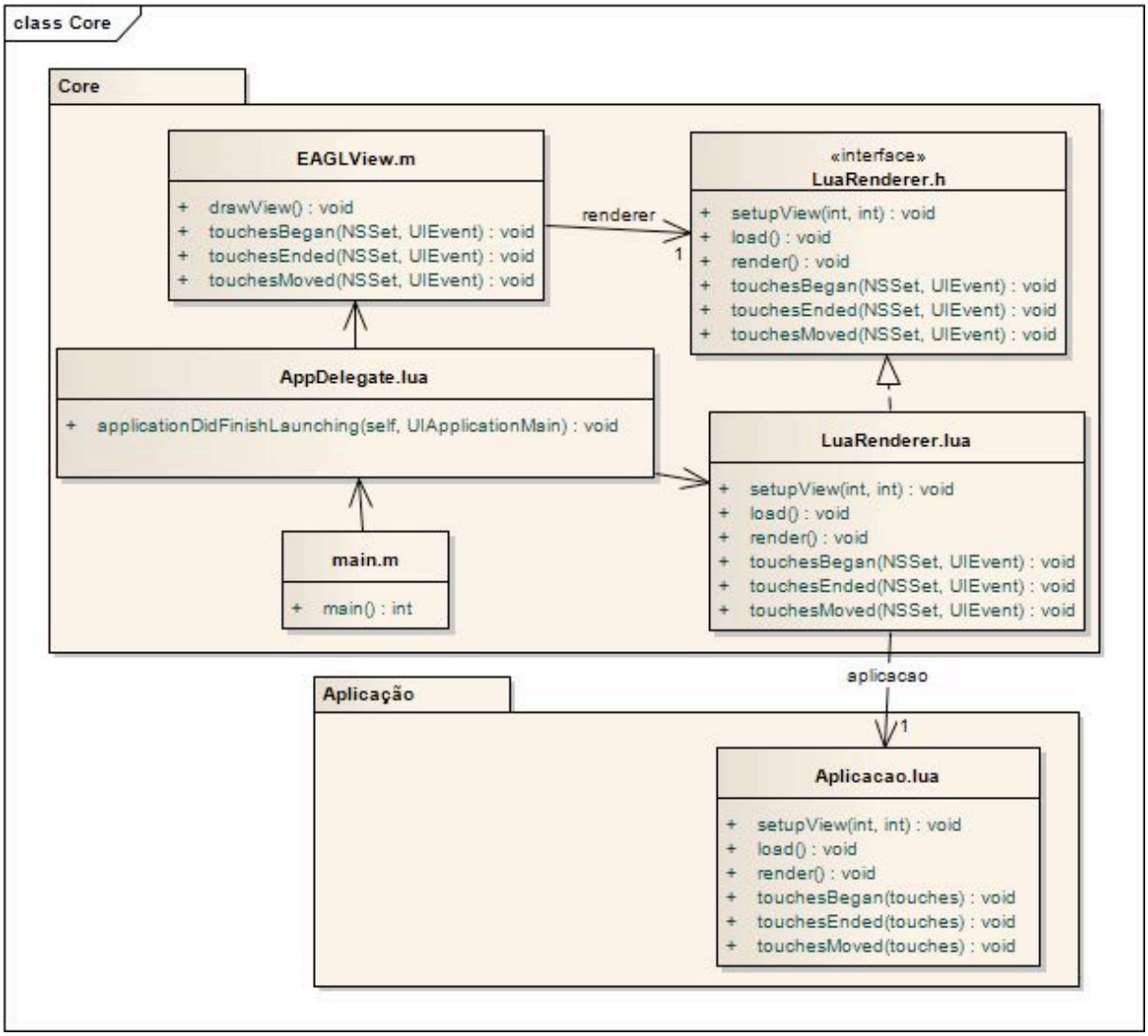


Figura 10 – Diagrama de classes do Core e Aplicação

O arquivo fonte Objective-C chamado `LuaRenderer.h` define os métodos necessários para a implementação da aplicação. O método `setupView` recebe dois parâmetros (*width*, *height*), representando a largura e a altura da tela do iPhone, este método é utilizado para configurar a visão da aplicação de acordo com as dimensões do dispositivo. O método `load` não recebe nenhum parâmetro, e deve ser utilizado para carregar os objetos a serem renderizados, no caso adicionar os objetos ao grafo de cena. O método `render` não recebe nenhum parâmetro e é responsável pelas chamadas para as funções de renderização do OpenGL ES. Os métodos `touchesBegan`, `touchesMoved` e `touchesEnded` são os métodos que recebem as chamadas dos eventos de toque na tela. Estes métodos recebem por parâmetro um *array* de objetos de toque. Cada objeto possui os atributos `tapCount`, `x` e `y`. O atributo `tapCount` é um número indicando a quantidade de vezes que o usuário tocou a tela em certo ponto por um período de tempo pré definido. Os atributos `x` e `y` representam a posição na tela que o toque ocorreu.

O *script* `LuaRenderer.lua` implementa os métodos definidos no arquivo `LuaRenderer.h`. Esta implementação é possível através da utilização do iPhone WAX. Este script é um *wrapper* para o script `Aplicacao.lua` repassando todas as chamadas de método para o `Aplicacao.lua`. Ele existe somente para que o `Aplicação.lua` não seja dependente do *framework* iPhone WAX. O *script* `Aplicação.lua` possui os métodos necessários para construir a aplicação e tratar os eventos. Este arquivo não é dependente de nenhuma biblioteca associada ao iOS, dessa forma é possível executar a aplicação em qualquer outra plataforma que tenha suporte a Lua e OpenGL ES.

O método `main` do arquivo `main.m` é invocado pelo iOS para iniciar a aplicação, nele é carregado o *framework* iPhone WAX e a API LOGLES, é criado também o `UIApplicationMain` do UIKit. O *script* `AppDelegate.lua` implementa os métodos definidos no arquivo `UIApplicationDelegate` do UIKit. Esta implementação é possível através da utilização do iPhone WAX.

A classe `EAGLView` é responsável por criar um contexto de renderização do OpenGL ES e o *framebuffer* para ser exibir os gráficos. Ela é instanciada pelo método `initWithFrame`. O método `setRenderer` recebe por parâmetro um objeto do tipo `LuaRenderer.h`. Este objeto é armazenado no atributo `renderer`. O método `drawView` é responsável por renderizar os gráficos. Os métodos `touchesBegan`, `touchesMoved`, `touchesEnded` são invocados pelo UIKit quando ocorre eventos de toque na tela.

3.2.3.4 MJ3L

Neste pacote estão os códigos desenvolvidos em Lua que fazem parte do motor de jogos. Os objetos deste pacote tem acesso a biblioteca LOGLES para utilizar as funções OpenGL ES. O desenvolvimento do MJ3L foi baseado no motor de jogos MJ3I desenvolvido por Takano (2009). A Figura 11 apresenta as classes e estruturas que formam a biblioteca.

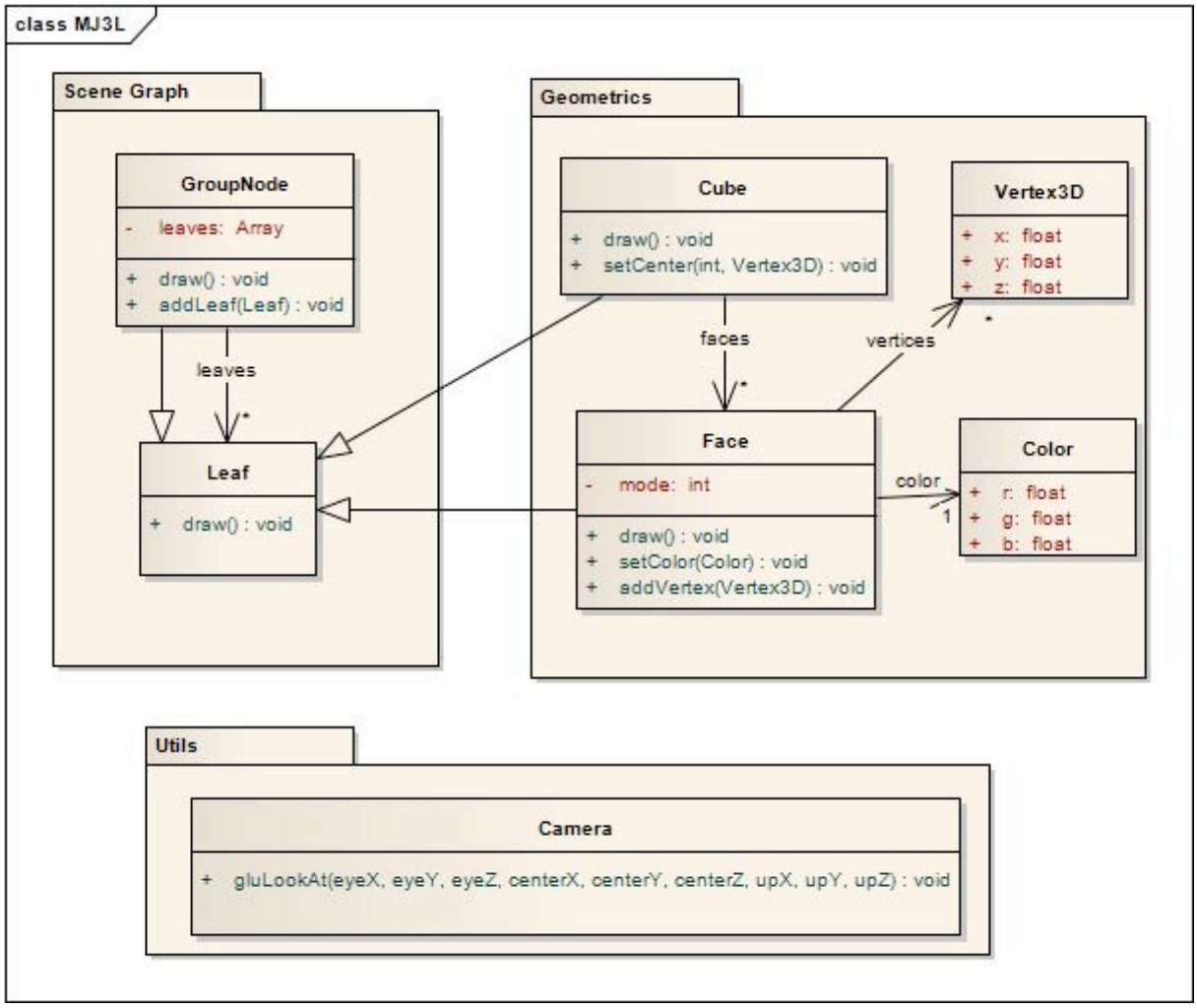


Figura 11 – Diagrama de classe do MJ3L

O projeto da biblioteca foi divido em três pacotes, *Scene Graph*, *Geometrics* e *Utils*.

3.2.3.4.1 *Scene Graph*

Neste pacote estão as classes que formam o grafo de cena, utilizado para o gerenciamento no ambiente 3D. O grafo de cena é organizado na forma de uma estrutura em árvore com nó e folhas. É composto por duas classes, `GroupNode` e `Leaf`.

A classe `Leaf` possui o método `draw`, as classes que herdam desta classe devem implementar este método para renderizar quando for um objeto geométrico ou fazer outro tipo de tratativa como no caso da classe `GroupNode`.

A classe `GroupNode` é uma herança da classe `Leaf`, e possui um atributo do tipo *array* com o nome de `leaves` que é utilizado para armazenar objetos que herdam da classe `Leaf`. O

método `addLeaf` é utilizado para adicionar novos objetos no *array* `leaves`. O parâmetro utilizado é um objeto do tipo `Leaf`. O método `draw` é responsável por iterar pelo *array* de `leaves` e invocar o método `draw` de cada item. Este método não possui nenhum parâmetro e o retorno é nulo.

3.2.3.4.2 *Geometrics*

A classe `Color` representa uma cor no modelo de cores RGB (*Red*, *Green*, *Blue*). Possui os atributos `r`, `g` e `b` do tipo float para armazenar o valor relativo a intensidade de cada cor (o valor pode variar entre o mínimo de zero e o máximo de 1). A classe `Vertex3d` representa a posição de um ponto em um espaço 3D. A coordenada do ponto é definida pelos atributo `x`, `y` e `z` do tipo float.

A classe `Face` representa o desenho de uma face na tela, para tal ela possui um *array* de vértices do tipo `Vertex3d`. O *array* não possui um limite de vértices, dessa forma pode-se aproveitar esta classe para gerar faces com três, quatro ou infinitas vértices. Para adicionar vértices deve-se utilizar o método `addVertex` e passar o `Vertex3d` por parâmetro. A cor da face é definida pelo método `setColor` passando a `Color` por parâmetro. A classe também possui o atributo `mode` do tipo inteiro, que é utilizado para definir o tipo de primitiva que será renderizado, possuindo o valor de uma constante conforme pré-definido pelo LOGLES (*gl.POINTS*, *gl.LINE_STRIP, gl.LINE_LOOP, gl.LINES, gl.TRIANGLE_STRIP, gl.TRIANGLE_FAN*, *gl.TRIANGLES*). O método draw renderiza as vértices adicionadas a face.

A classe `Cube` é utilizada para criar um objeto geométrico de um cubo no espaço 3D. O método `setCenter` possui dois parâmetros (`size`, `center`). O parâmetro `size` é do tipo inteiro e define o tamanho do cubo, e o parâmetro `center` é do tipo `Vertex3d` e define o ponto central do cubo. A partir desses parâmetros são geradas seis faces do tipo `Face`, cada uma com quatro vértices para formar o cubo. O método `draw` invoca o método `draw` das faces geradas no cubo.

3.2.3.4.3 *Utils*

Este pacote possui somente a classe `Camera`. O objetivo desta classe é facilitar a

movimentação e controle da janela de visualização do OpenGL ES no espaço 3D. De um ponto de vista mais abstrato a classe é como se fosse uma câmera no mundo real, onde o que define a imagem é o que está sendo filmado pela câmera e de onde está sendo filmado. São justamente estes os dados que são passados como parâmetros na chamada do método de movimentação de câmera. No método `gluLookAt`, os três primeiros parâmetros (`eyeX`, `eyeY` e `eyeZ`), são utilizados para definir no espaço 3D a posição do ponto onde a câmera estará visualizando. Os seguintes próximos três parâmetros (`centerX`, `centerY` e `centerZ`), definem onde fica a posição da janela de visualização e por final, os parâmetros (`upX`, `upY` e `upZ`) são utilizados para informar qual é a direção do vetor para cima.

3.2.4 Diagrama de sequência

O diagrama da Figura 12 apresenta o processo para inicializar a aplicação, demonstrando a criação das classes envolvidas no pacote Core e Aplicação.

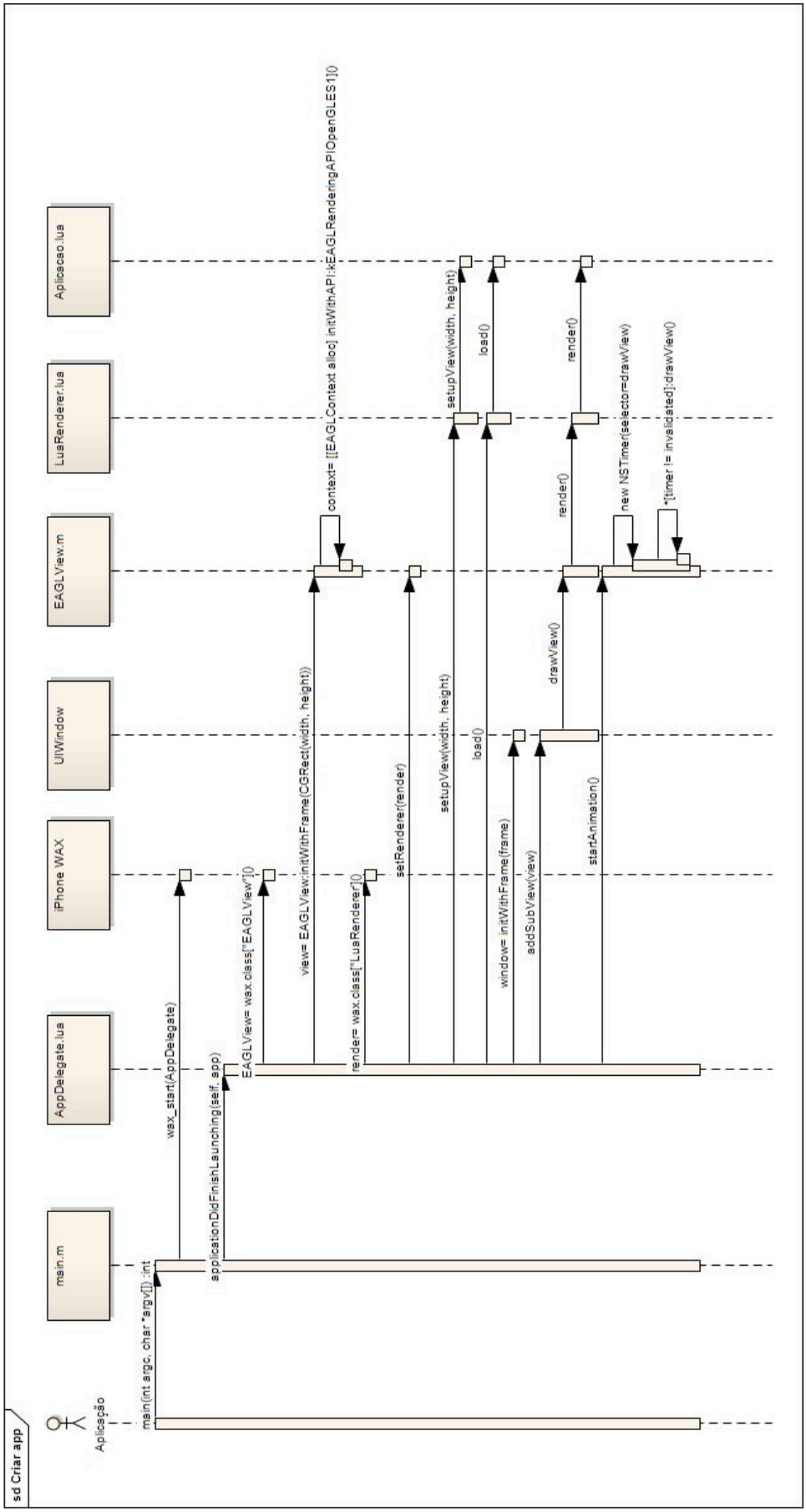


Figura 12 – Diagrama de sequência

3.3 IMPLEMENTAÇÃO

Esta seção apresenta as bibliotecas e ferramentas utilizadas para a execução de Lua na plataforma iOS e como foi feito o desenvolvimento do motor de jogos.

3.3.1 Técnicas e ferramentas utilizadas

Todo o desenvolvimento foi feito utilizando a IDE *xCode*, disponibilizado pela Apple no suíte de ferramentas de desenvolvimento com o mesmo nome. Para depuração foi utilizado o Iphone Simulator, que é integrado com a IDE do *xCode*.

Nesta seção serão apresentados x seções. Na seção 3.3.1.1 é apresentado a instalação do framework Wax iPhone e Criação da Aplicação. Na seção 3.3.1.2 é apresentado como foi criado a funcionalidade de renderização OpenGL ES em Lua no iPhone. Na seção 3.3.1.3 é apresentado o tratamento de eventos de toque em Lua. Na seção 3.3.1.4 é apresentado o desenvolvimento do motor de jogos.

3.3.1.1 Instalação do *framework* Wax iPhone e Criação da Aplicação

O Wax iPhone fornece um *template* "Wax iPhone App" para a IDE xCode, que é um *template* para criar aplicações utilizando este *framework*, conforme Figura 13. Para instalar o *template* deve-se seguir os seguintes passos:

a) baixar o projeto do portal GITHUB (2011).
b) acessar a pasta do wax utilizando o terminal de comandos, e executar o comando `rake install`. Isto irá instalar o *template* do projeto no xCode.

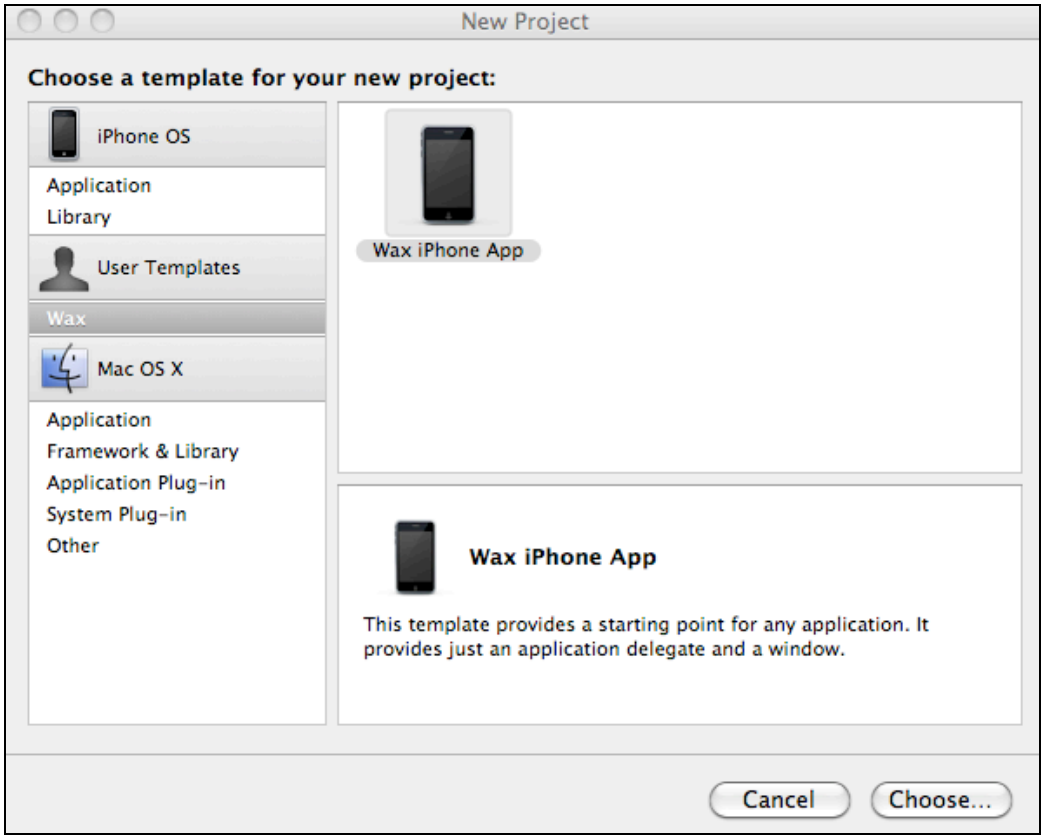


Figura 13 – Tela onde é apresentado o *template* Wax iPhone.

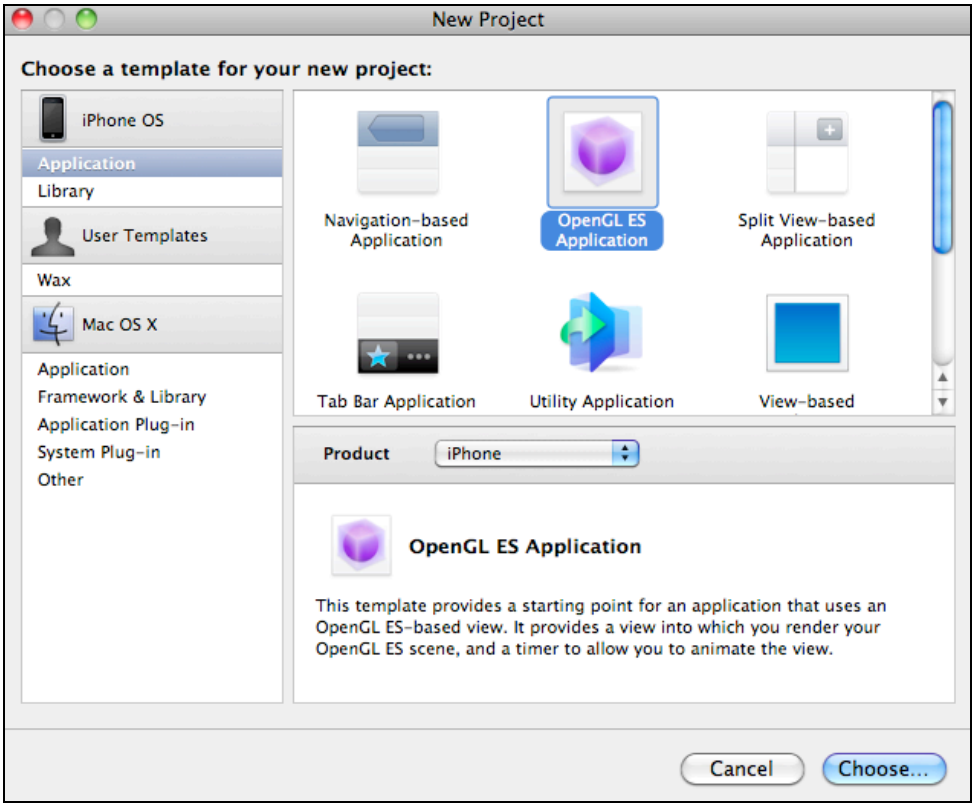
Este *template* gera um projeto com a capacidade de utilizar Lua no desenvolvimento de aplicativos para iOS. O arquivo `AppDelegate.lua` é gerado pelo *template*. O método `applicationDidFinishLaunching` deste arquivo é onde deve ser montado o layout da aplicação. Ele foi alterado para criar o `EAGLView.m` e o `LuaRenderer.lua`. O `LuaRenderer.lua` é setado como `renderer` do `EAGLView.m`, e o `EAGLView.m` é adicionado ao `UIWindow` do UIKit também criado neste método. Os métodos `setupView` e `load` do `LuaRenderer.lua`, e o método `startAnimation` do `EAGLView.m` são invocados neste método. Conforme é apresentado no Quadro 6. Na seção 3.3.1.2 é apresentado o `EAGLView.m`.

```
function applicationDidFinishLaunching(self, application)
  local f = UIScreen:mainScreen():bounds()
  --cria o UIWindow
  self.window = UIWindow:initWithFrame(f)
  -- cria um instância do LuaRenderer
  local lt = wax.class["LuaRenderer"];
  -- cria o EAGLView
  local EAGLView = wax.class["EAGLView"];
  local view = EAGLView:initWithFrame(CGRect(0,0,f.width,f.height))
  -- associa o renderer ao EAGLView
  view:setRenderer(lt);
  -- configura a visão e invoca o load pra carregar o grafo de cena
  lt:setupView(frame.width, frame.height)
  lt:load();
  --inicia o temporizador de animação
  view:startAnimation()
  -- adiciona o EAGLView na window
  self.window:addSubview(view)
  self.window:makeKeyAndVisible()
end
```

Quadro 6 – Apresenta a função `applicationDidFinishLaunching` do *script* `AppDelegate.lua`

3.3.1.2 Renderizar OpenGL ES em Lua no iPhone

A IDE xCode possui o *template* "OpenGL ES Application", que é um template padrão existente no System Development Kit (SDK) do iPhone, conforme Figura 14.



Fígura 14 – Tela onde é apresentado o *template* OpenGL ES Application

Este *template* gera os códigos para criar um contexto de renderização OpenGL ES, um *framebuffer* onde os gráficos serão renderizados e um temporizador para permitir a animação de cena. Usando este template como referência foi criado a classe `EAGLView.m` no projeto wax. No construtor `initWithFrame` desta classe foi adicionado o código para criar o contexto de renderização do OpenGL ES e criar um *framebuffer* e setar os valores *default* do controle de animação. O método `startAnimation` inicia o temporizador de animação que invoca o método `drawView` em um intervalo de tempo. A implementação desses métodos são demonstrados no Quadro 7.

```
- (id)initWithFrame:(CGRect)frame {
    if ((self = [super initWithFrame:frame])) {
  // Get the layer
        CAEAGLLayer *eaglLayer = (CAEAGLLayer *)self.layer;
  eaglLayer.opaque = TRUE;
        eaglLayer.drawableProperties = [NSDictionary
dictionaryWithObjectsAndKeys: [NSNumber numberWithBool:FALSE],
kEAGLDrawablePropertyRetainedBacking, kEAGLColorFormatRGBA8,
kEAGLDrawablePropertyColorFormat, nil];
// Create the OpenGL ES 1.1 context
          context = [[EAGLContext alloc]
initWithAPI:kEAGLRenderingAPIOpenGLES1];
          if (!context || ![EAGLContext setCurrentContext:context])
          {
                [self release];
                return nil;
// Create the OpenGL ES 1.1 context
          context = [[EAGLContext alloc]
initWithAPI:kEAGLRenderingAPIOpenGLES1];
          if (!context || ![EAGLContext setCurrentContext:context])
          {
                [self release];
                return nil;
}

          // Create default framebuffer object. The backing will be
allocated for the current layer in -resizeFromLayer
          glGenFramebuffersOES(1, &defaultFramebuffer);
          glGenRenderbuffersOES(1, &colorRenderbuffer);
          glBindFramebufferOES(GL_FRAMEBUFFER_OES, defaultFramebuffer);
          glBindRenderbufferOES(GL_RENDERBUFFER_OES, colorRenderbuffer);
          glFramebufferRenderbufferOES(GL_FRAMEBUFFER_OES,
GL_COLOR_ATTACHMENT0_OES, GL_RENDERBUFFER_OES, colorRenderbuffer);
animating = FALSE;
        displayLinkSupported = FALSE;
        animationFrameInterval = 5;
        displayLink = nil;
        animationTimer = nil;
// A system version of 3.1 or greater is required to use CADisplayLink.
The NSTimer
        // class is used as fallback when it isn't available.
        NSString *reqSysVer = @"3.1";
        NSString *currSysVer = [[UIDevice currentDevice] systemVersion];
        if ([currSysVer compare:reqSysVer options:NSNumericSearch] !=
NSOrderedAscending)
```

```
            displayLinkSupported = TRUE;
    }
    return self;
}

- (void)startAnimation
{
    if (!animating)
    {
        if (displayLinkSupported)
        {

            displayLink = [NSClassFromString(@"CADisplayLink")
displayLinkWithTarget:self selector:@selector(drawView)];
            [displayLink setFrameInterval:animationFrameInterval];
            [displayLink addToRunLoop:[NSRunLoop currentRunLoop]
forMode:NSDefaultRunLoopMode];
        }
        else
            animationTimer = [NSTimer
scheduledTimerWithTimeInterval:(NSTimeInterval)((1.0 / 60.0) *
animationFrameInterval) target:self selector:@selector(drawView)
userInfo:nil repeats:TRUE];
animating = TRUE;
    }
}
- (void)drawView
{
    [EAGLContext setCurrentContext:context];
// Invoca o render do LuaRenderer.lua
    [renderer render];
     glBindRenderbufferOES(GL_RENDERBUFFER_OES, colorRenderbuffer);
    [context presentRenderbuffer:GL_RENDERBUFFER_OES];
}
```

Quadro 7 – Apresenta os métodos `initWithFrame`, `startAnimation` e `drawView` da classe `EAGLView.m`

No método `drawView` é onde deve-se invocar as funções OpenGL ES. Neste caso ele invoca o método render do `renderer` associado a ele. Este `renderer` é uma implementação em Lua. No Quadro 8 é apresentado um código exemplo de um `renderer` que usa a biblioteca LOGLES para renderizar OpenGL ES.

```
function Aplicacao:render()
    local squareVertices = { --cria o array de vertices
        {-0.5,  -0.33},
         {0.5,  -0.33},
        {-0.5,   0.33},
         {0.5,   0.33},
    };
    local squareColors = { --cria o array de cores
        {255, 255,   0, 255},
        {0,   255, 255, 255},
        {0,     0,   0,   0},
        {255,   0, 255, 255},
    };
    gl.MatrixMode(gl.PROJECTION);
    gl.LoadIdentity();
    gl.MatrixMode(gl.MODELVIEW);
    gl.LoadIdentity();
    gl.ClearColor(0.5, 0.5, 0.5, 1.0);
    gl.Clear(gl.COLOR_BUFFER_BIT);
    gl.VertexPointer(squareVertices);
    gl.EnableClientState(gl.VERTEX_ARRAY);
    gl.ColorPointer(squareColors);
    gl.EnableClientState(gl.COLOR_ARRAY);
    gl.DrawArrays(gl.TRIANGLE_STRIP, 0, 4);
end;
```

Quadro 8 – Código para renderizar OpenGL ES em Lua acessando a API LOGLES.

A Figura 15 mostra o resultado da execução do código exemplo utilizando o LOGLES.

Figura 15 – Face renderizada utilizando o LOGLES

3.3.1.3 Tratamento de Eventos de Toque em Lua

A classe `EAGLView.m` implementa os métodos `touchesBegan`, `touchesMoved`, `touchesEnded`. Nestes métodos as chamadas são repassadas para o `renderer`. São extraídos as propriedades `tapCount`. `x` e `y` da classe `UITouch` e passado para os métodos de tratamentos de evento do `renderer`. Conforme apresentado no Quadro 9.

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
     [renderer touchesBegan:touches]; //Invoca o touchesBegan do renderer
}

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
     [renderer touchesEnded:touches]; //Invoca o touchesEnded do renderer
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event{
     [renderer touchesMoved:touches]; //Invoca o touchesMoved do renderer
}
```

Quadro 9 – Métodos de tratamento de evento da classe `EAGLView.m`

No Quadro 10 é apresentado o código do `renderer` que implementa os métodos de tratamento de eventos, extraindo as propriedades `tapCount`, `x` e `y` da classe `UITouch`, e repassa para o `Aplicacao.lua`.

```
function touchesEnded(self,touches)
     app:touchesEnded(getTouchObjects(touches);--Invoca método do app
end

function touchesMoved(self,touches)
     app:touchesMoved(getTouchObjects(touches);
end

function touchesBegan(self,touches)
     app:touchesBegan(getTouchObjects(touches));
end

function getTouchObjects(touches)
     local t = {};
     local touchesArray = touches:allObjects()

     -- monta um array em lua, sem os objetos do iOS
     for i=1, table.getn(touchesArray) do
           local theTouch = touchesArray[i]
           touchPos = theTouch:locationInView(theTouch:view());

             --seta em t, ss propriedades do UITouch
             t[i] = {tapCount = theTouch:tapCount(), x = touchPos.x, y =
touchPos.y}
     end

     return t;
end
```

Quadro 10 – Métodos de tratamento de evento do *script* `LuaRenderer.lua`

3.3.1.4 Motor de Jogos

Nesta seção é apresentado o código do motor de jogos desenvolvido de acordo com a especificação deste trabalho. Na seção 3.3.1.4.1 é apresentado o grafo de cena. Na seção 3.3.1.4.2 são apresentados os objetos geométricos. Na seção 3.3.1.4.3 é apresentado o objeto que controla a movimentação de câmera.

3.3.1.4.1 Grafo de Cena

Conforme definido na seção 3.2.3.4.1, o grafo de cena é uma estrutura hierárquica com folhas (`Leaf`) e nós (`GroupNode`), onde o `GroupNode` estende `Leaf`.

Segundo LUA ORG (2011), Lua não é uma linguagem puramente orientada a objetos. Ela fornece meta-mecanismos para a implementação de classes e herança. Os meta-mecanismos de Lua trazem uma economia de conceitos e mantêm a linguagem pequena, ao mesmo tempo que permitem que a semântica seja estendida de maneiras não convencionais. O Quadro 11 apresenta uma implementação útil para fornecer em Lua a capacidade de criar e estender classes.

```
function inheritsFrom( baseClass )

    local new_class = {}
    local class_mt = { __index = new_class }

    function new_class:create()
        local newinst = {}
        setmetatable( newinst, class_mt )
        return newinst
    end

    if nil ~= baseClass then
        setmetatable( new_class, { __index = baseClass } )
    end

    -- Retorna a classe do objeto de instancia
    function new_class:class()
        return new_class
    end

    -- Returna a classe super do objeto de instancia
    function new_class:superClass()
        return baseClass
    end
```

```
    -- Returna true se o objeto for do tipo theClass
    function new_class:isa( theClass )
        local b_isa = false

        local cur_class = new_class

        while ( nil ~= cur_class ) and ( false == b_isa ) do
            if cur_class == theClass then
                b_isa = true
            else
                cur_class = cur_class:superClass()
            end
        end

        return b_isa
    end

    return new_class
end
```

Quadro 11 – Código útil para instanciar e estender classes em Lua

No Quadro 12 é apresentada a implementação da classes `Leaf` e `GrupoNode`.

```
--Classe Leaf
Leaf = {}
Leaf_mt = { __index = Leaf }

function Leaf:create() — Cria uma instancia de Leaf
   local new_inst = {}
   setmetatable( new_inst, Leaf_mt )
   return new_inst
end

function Leaf:draw()
 puts("Sub class need to override this method.")
end

--Classe GroupNode

GroupNode = inheritsFrom( Leaf )

function GroupNode:draw()
   -- itera pelos leaves, e invoca o draw deles
   local l = self.leaves
   while l do
     l.leaf:draw()
     l = l.next
    end
end

function GroupNode:addLeaf( l )
      --add na lista encadeada
   self.leaves = {next=self.leaves, leaf = l};
end
```

Quadro 12 – Implementação das classes `Leaf` e `GroupNode`

Como pode ser observado neste quadro, a implementação em Lua é muito simplificada devido a praticidade da linguagem. Por exemplo no método `addLeaf` em apenas uma linha de código foi possível montar a lista encadeada para armazenar as folhas.

3.3.1.4.2 Objetos Geométricos

O Quadro 13 apresenta as funções para criar os objetos `Vertex3D` e `Color`. Não foram criadas classes para estes objetos, pois eles são usados somente para armazenar os valores de seus atributos.

```
function create3dVertex(x, y , z)
 return {x = x,y = y,z = z} -- retorna uma struct com x, y e z
end
function createColor(r, g , b)
 return {r = r,g = g,b = b} -- retorna uma struct com r, g e b
end
```

Quadro 13 – Funções para criar `Vertex3D` e `Color`

A classe `Face` estende a classe `Leaf` e acessa a API LOGLES para renderizar os vértices associados a ela, conforme apresentado no Quadro 14.

```
Face = inheritsFrom( Leaf ) -- classe Face extende Leaf

function Face:setVertices( v )
     self.squareVertices = v;
end

function Face:setMode( mode )
     self.mode = mode
end

function Face:setColor( color )
     self.color = color
end

function Face:draw() -- função para renderizar a face
     if(self.mode == nil) then
           self.mode = gl.TRIANGLE_FAN -- gl.TRIANGLE_FAN é o padrão
     end
     if(self.color == nil) then
           self.color = createColor(0,0.5,0)
     end
     squareColors = {
        {self.color.r, self.color.g, self.color.b, 1},
        {self.color.r, self.color.g, self.color.b, 1},
        {self.color.r, self.color.g, self.color.b, 1},
        {self.color.r, self.color.g, self.color.b, 1}
    };

    gl.EnableClientState(gl.VERTEX_ARRAY);
    gl.EnableClientState(gl.COLOR_ARRAY);
    gl.ColorPointer(squareColors); -- função LOGLES que seta a cor
    gl.VertexPointer(self.squareVertices);
gl.DrawArrays(self.mode, 0, 4) -- função LOGLES que renderiza os arrays

    gl.DisableClientState(gl.VERTEX_ARRAY);
    gl.DisableClientState(gl.COLOR_ARRAY);

end
```

Quadro 14 – Implementação da classe `Face`

3.3.1.4.3 Movimentação Câmera

A implementação de movimentação de câmera foi baseada na especificação GLU LOOK AT (2011). O Quadro 15 apresenta a implementação da função `gluLookAt`.

```
function gluLookAt(eyex, eyey, eyez, centerx, centery, centerz, upx, upy,
upz)

 forward={}
 side={}
 up={}
 m={} --[4][4];

 forward[0] = centerx - eyex;
 forward[1] = centery - eyey;
 forward[2] = centerz - eyez;

 up[0] = upx;
 up[1] = upy;
 up[2] = upz;

 normalize(forward);

 -- Side = forward x up
 cross(forward, up, side);
 normalize(side);

 -- Recalcula up como: up = side x forward
 cross(side, forward, up);

 --preenche a matriz com os valores das posições calculadas
 m[1] = side[0];
 m[2] = side[1];
 m[3] = side[2];
 m[4] = 0
 m[5] = up[0];
 m[6] = up[1];
 m[7] = up[2];
 m[8] = 0
 m[9] = -forward[0];
 m[10] = -forward[1];
 m[11] = -forward[2];
 m[12] = 0
 m[13] = 0
 m[14] = 0
 m[15] = 0
 m[16] = 1

 gl.MultMatrix(m);
 -- função LOGLES para translação
 gl.Translate(-eyex, -eyey, -eyez);
 end
```

Quadro 15 – Função para focar a câmera em uma posição na tela

3.3.2 Operacionalidade da implementação

Para utilizar as funções do motor de jogos deve-se abrir no xCode o projeto exemplo oferecido neste trabalho, pois este já possui todas as dependências configuradas. O arquivo `Aplicacao.lua` é onde o jogo ou aplicação deve ser criada.

3.3.2.1 Configurar a área de visão

Ao trabalhar com OpenGL ES é preciso configurar a área de visão que o OpenGL ES vai ocupar do canvas. No LOGLES esta configuração é feita pela função `gl.ViewPort(x, y, width, height)`. O método `setupView` do `Aplicacao.lua` é invocado pelo *Core* e recebe por parâmetro as dimensões do *canvas* para que a aplicação possa configurar a área de visão. O Quadro 16 apresenta um exemplo de como pode ser configurado uma área de visão.

```
function Aplicacao:setupView(width, height)
 --define as posições de visualização da camera
 eye[0] = 0.0;
 eye[1] = 0.0;
 eye[2] = 2.0;

 center[0] = 0.0;
 center[1] = 0.0;
 center[2] = -10.0;

 zNear = 0.1
 zFar = 1000
 fieldOfView = 60.0;
   --define para trabalhar com profundidade
 gl.Enable(gl.DEPTH_TEST);
 gl.MatrixMode(gl.PROJECTION);
 size = zNear * math.tan(math.rad(fieldOfView) / 2.0);

 gl.Frustum(-size,size,-
size/(width/height),size/(width/height),zNear,zFar)
 --Configura a area de visão
 gl.Viewport(0, 0, width, height);

 gl.ClearColor(0, 0, 0, 0);
end
```

Quadro 16 – Apresenta um exemplo de implementação para a função `setupView` do `Aplicacao.lua`

3.3.2.2 Construção e renderização do grafo de cena

A construção do grafo de cena é feita através da criação de um nó raiz do tipo `GroupNode`. Neste nó serão incluídos os objetos geométricos e outros nós filhos. A adição de nós filhos é feito pelo método `addLeaf`. O método `load` do `Aplicacao.lua` é invocado quando aplicação é carregada, sendo aconselhável montar o grafo de cena nesse método. O quadro 17 apresenta um exemplo de como pode ser criado um grafo de cena.

```
function Aplicacao:load()
 root = GroupNode:create(); -- cria o nó root da árvore

 squareVertices2 = {
       {-0.5, 0.0, 0},
       {-0.5,  0.5, 0},
       {0.5,   0.5, 0},
       {0.5,  0.0, 0},
      };
 --cria uma instancia de Face
      local s2 = Face:create()
 s2:setColor(createColor(1,0,0)) -- seta a cor
 s2:setVertices(squareVertices2)
 root:addLeaf(s2) -- adiciona a Face no nó da árvore
end
```

Quadro 17 – Apresenta a criação de um grafo de cena na função `load` do Aplicacao.lua

Neste Quadro 17 é instanciado um nó raiz `GroupNode`, e adicionado a ele um objeto geométrico do tipo Face com quatro vértices. Neste processo ocorre somente a construção do grafo de cena, ou seja os objetos não são renderizados. Para renderizar um grafo de cena deve-se invocar o método `draw` do nó raiz dentro do método render do `Aplicacao.lua`, conforme demonstrado no Quadro 18.

```
function Aplicacao:render()
 gl.MatrixMode(gl.MODELVIEW);
 gl.ClearColor(0.2, 0.2, 0.2, 1.0);
 gl.Clear(gl.COLOR_BUFFER_BIT);
 gl.Clear(gl.DEPTH_BUFFER_BIT)
 gl.LoadIdentity();
 --configura o ponto de visao da camera
 gluLookAt(eye[0],eye[1],eye[2],center[0],center[1],center[2],0.0,1.0,0.0)
 -- invoca o draw no nó raiz
 root:draw()
end;
```

Quadro 18 – Apresenta a implementação para renderizar um grafo de cena no método `render` do `Aplicacao.lua`

O método `gluLookAt` é utilizado para configurar a câmera pra visualizar os objetos, nele é passado a posição do objeto geométrico adicionado ao grafo de cena.

3.3.2.3 Tratamento de eventos de toque

O tratamento de eventos de toque é feito através dos métodos `touchesBegan`, `touchesMoved` e `touchesEnded`, eles recebem por parâmetro uma lista de objetos de toque. O objeto de toque possui as propriedades `tapCount`, `x` e `y`. No Quadro 19 é apresentado uma implementação no método `touchesBegan` para movimentar o ponto de visão da câmera na horizontal ou vertical, de acordo com o ponto onde o toque ocorreu. No caso de toque simples a movimentação é na horizontal, e toque duplo é na vertical.

```
function Aplicacao:touchesBegan(touches)
      touchPos = touches[1];
      --Toque duplo, altera a vizualização pela posição y
      if(touchPos.tapCount>1) then
            if(touchPos.y <320) then
                  eye[1] = eye[1]-0.2;--eye, array utilizado pela Camera
            else
                  eye[1] = eye[1]+0.2;
            end
     else --Toque simples, altera a vizualização pela posição x
            if(touchPos.x <160) then
                  eye[0] = eye[0]+0.2;
            else
                  eye[0] = eye[0]-0.2;
            end
     end
end
```

Quadro 19 – Apresenta a implementação para tratar toque na tela no método `touchesBegan` do `Aplicacao.lua`

Na Figura 16 é apresentado a movimentação de câmera gerada a partir de toques simples no lados direito da tela.

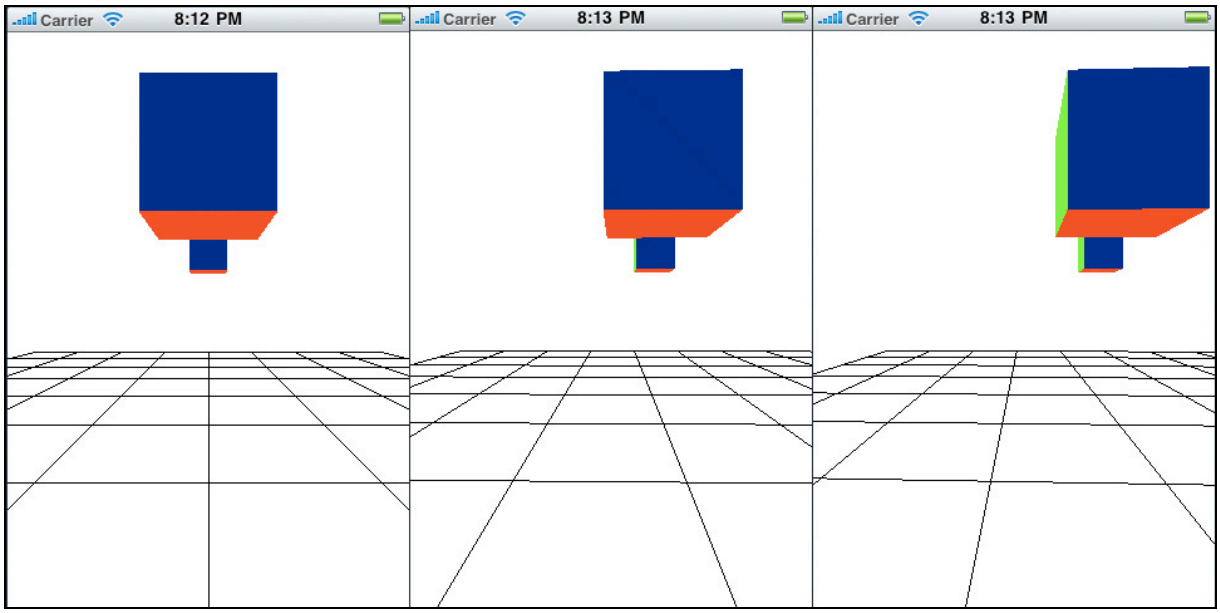


Figura 16 - Movimentação de câmera na horizontal

Na Figura 17 é apresentado a movimentação de câmera gerada a partir de toques duplos na parte superior da tela.

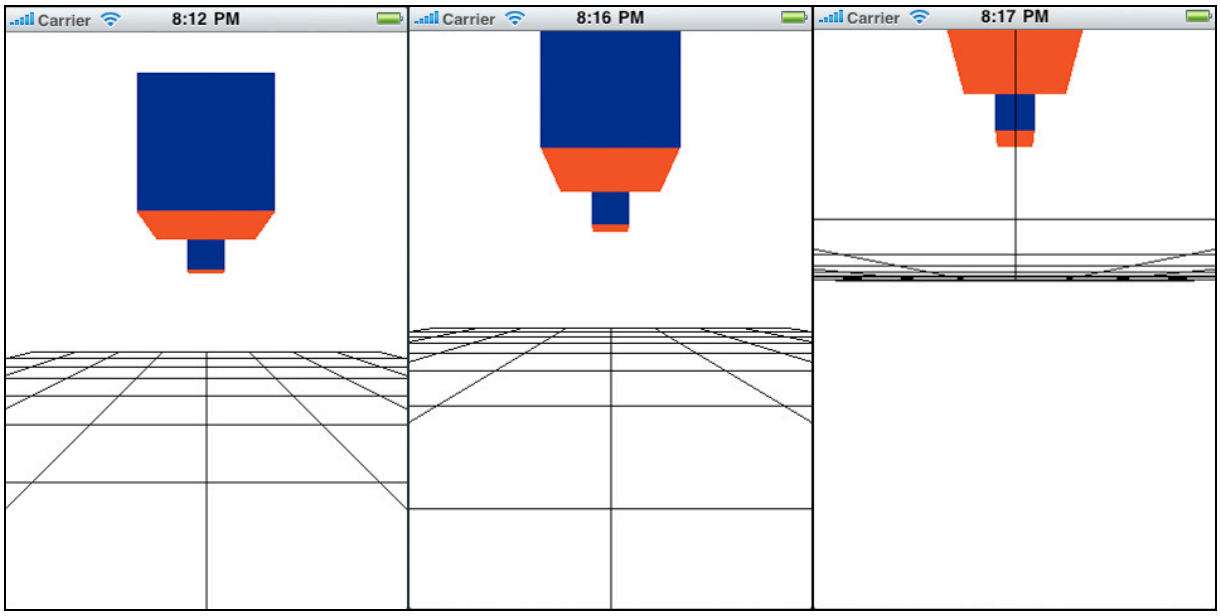


Figura 17 - Movimentação de câmera na vertical

3.4 RESULTADOS E DISCUSSÃO

Este trabalho apresentou o projeto e desenvolvimento de um motor de jogos na linguagem de *script* Lua, das funções para suportar a execução deste motor no iOS, e também da API para renderizar OpenGL ES em Lua. O motor de jogos oferece uma estrutura em forma de árvore para gerenciamento de grafo de cena. A estrutura permite que um nó possua *n* filhos do tipo nó interno ou folha. O objeto do tipo folha representa uma forma geométrica a ser renderizada. A Figura 18 apresenta um diagrama de um grafo de cena, onde objetos do tipo nó interno são representados por um quadrado e objetos do tipo folha são representados por um círculo.

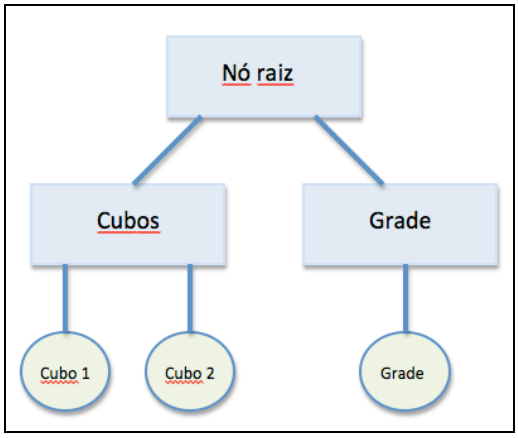


Figura 18 - Diagrama do grafo de cena

Na Figura 19 é apresentado o resultado da renderização do grafo de cena apresentado na Figura 18.

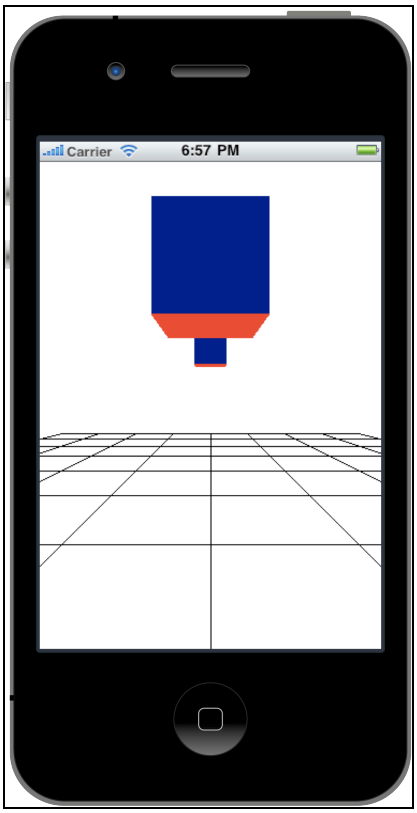


Figura 19 – Possui os objetos conforme o grafo de cena

O motor de jogos possui também uma rotina para movimentação de câmera no espaço 3D, que é utilizado para focar a vizualização dos objetos em um ponto desejado da tela. O motor não possui nenhuma rotina para tratamento de cálculos de física, áudio, detecção de colisão, entre outra funcionalidades importantes para um motor. Esta limitação torna este motor inferior aos motores dos trabalhos correlatos. Por outro lado, como vantagem este motor é o único *open source* onde pode ser desenvolvido um jogo e renderizar OpenGL ES

utilizando somente a linguagem Lua.

Para o desenvolvimento de aplicações, o uso da linguagem Lua pode representar uma vantagem em comparação com a linguagem Objective-C, pois ela possui sintaxe simples e gerenciador automático de memória para coleta de lixo. Nos quadros 20 e 21, são demonstrados um exemplo de código em Lua e Objetive-C que possuem a mesma finalidade.

```
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    NSMutableArray *arrayColors = [NSMutableArray arrayWithObjects:
@"Red", @"Green", @"Blue", @"Yellow"]; //cria array de cores

    int i;
    int count;
    for (i = 0, count = [arrayColors count]; i < count; i = i + 1)
    {
       NSString *element = [myArray objectAtIndex:i]; //pega a cor da pos
       NSLog(@"Color: %@", element); //imprimi o element
    }
    [pool release];
```

Quadro 20 – Código exemplo de *array* em Objective C

```
  array = { "Red", "Green", "Blue", "Yellow" } --cria array de cores
  for i=1, table.getn(array) do
     print("Color: " .. array[i]) --imprimi a cor da posição i do array
  end
```

Quadro 21 – Código exemplo de *array* em Lua

Uma desvantagem na utilização da linguagem Lua no desenvolvimento para iPhone, é que a IDE do xCode que é integrada com o simulador do iPhone não possui um analisador de sintaxe para Lua, também não *highlight* das variáveis. Dessa forma os erros no código são indicados somente em tempo de execução.

4 CONCLUSÕES

O presente trabalho apresentou o projeto e desenvolvimento de um motor de jogos 3D para o iOS em Lua. Foi desenvolvida uma API em Lua para renderizar OpenGL ES, um grafo de cena para controle e gerenciamento de cena, tratamento de toques na tela e movimentação de câmera no ambiente 3D.

Os resultados finais podem ser considerados satisfatórios, pois com a ferramenta desenvolvida foi possível implementar uma aplicação renderizando OpenGL ES e realizando tratamento de eventos de toque na tela puramente em Lua. Também foi possível fazer uso das funções oferecidas pelo motor MJ3L para auxiliar no desenvolvimento de jogos.

Em relação as tecnologias utilizadas, a IDE de desenvolvimento xCode integrada com o simulador do iPhone foi muito eficiente para testar a aplicação, mas não foi tão eficiente para o desenvolvimento do motor, pois a IDE não possui analisador de sintaxe para a linguagem Lua, então os erros são indicados somente em tempo de execução.

Como limitação do presente trabalho, pode-se indicar que o MJ3L implemente um pequeno conjunto de funções. Como vantagem o MJ3L foi desenvolvido em Lua, que é uma linguagem de *script open source* e independente de plataforma, podendo executar em qualquer ambiente que tenha suporte a linguagem de programação C.

4.1 EXTENSÕES

Como sugestão de melhoria do motor implementado, pode-se otimizar o grafo de cena atual para cada nó armazenar a sua posição no espaço. Desta forma, passa a ser possível fazer ordenação espacial para otimizar a busca na árvore, otimizando assim o processo de renderização. Pode-se também implementar rotinas de translação, rotação no nó do grafo de cena.

Como sugestão para trabalhos futuros, pode-se adicionar um conjunto maior de funcionalidades ao motor de jogos. Um motor de física pode ser adicionado através do uso da biblioteca *Bullet* (BULLET, 2010), pois ela é *open source* e desenvolvida na linguagem de programação C. Neste caso, seria preciso implementar *wrappers* das funções do *Bullet* em

Lua. Para tal, pode-se usar a API LuaGL como referencia para a implementação. A biblioteca *LuaOpenAL* (LUAOPENAL, 2006) pode ser adicionada para o tratamento de áudio. A vantagem dessa biblioteca é que ela já esta em Lua, e acessa as funções da biblioteca *OpenAL*.

Outra sugestão seria desenvolver uma biblioteca para executar a aplicação de forma transparente em outras plataformas. Dessa forma, a implementação da biblioteca deverá invocar os métodos do `Aplicacao.lua` de acordo com a especificação. Uma sugestão de plataforma é o Android, pois já possui suporte a Lua e OpenGL ES.

REFERÊNCIAS BIBLIOGRÁFICAS

APPLE. **iPhone**. [S.l.], 2010. Disponível em: <http://www.apple.com/iphone/specs.html>. Acesso em: 15 set. 2010.

BOX2D. **Box2D physics engine**. [S.l.], 2010. Disponível em: <http://www.box2d.org/>. Acesso em: 8 nov. 2010.

BULLET. **Bullet collision detection & physics library**. [S.l.], 2010. Disponível em: <http://bulletphysics.com/Bullet/BulletFull/>. Acesso em: 6 jun. 2011.

CORONA. **Corona game edition for iOS and Android game development.** [S.l.] 2010. Disponível em: <http://anscamobile.com/corona/games/index.html>. Acesso em: 17 set. 2010.

EA. **UML tools for software development and modelling – enterprise architect UML modeling tool.** [S.l.] 2011. Disponível em: <http://www.sparxsystems.com.au/>. Acesso em: 6 jun. 2011.

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

GITHUB. **Secure source code hosting and collaborative development.** [S.l.], 2011. Disponível em: < https://github.com/probablycorey/wax/archives/master>. Acesso em: 13 jul. 2011.

GLU LOOK AT. **gluLookAt.** [S.l.], 2011. Disponível em: <http://pyopengl.sourceforge.net/documentation/manual/gluLookAt.3G.html>. Acesso em: 5 jun. 2011.

GOMES, Paulo R.; PAMPLONA, Vitor F. M3GE: um motor de jogos 3D para dispositivos móveis com suporte a mobile 3D graphics API**.** In: SIMPÓSIO BRASILEIRO DE GAMES, 1., 2005, São Paulo. **Anais…** São Paulo: USP, 2005. p. 2.

IERUSALIMSCHY, Roberto. **Programming in Lua**. Rio de Janeiro, 2003.

IOS. **iOS application pogramming guide:** the core application design**.** [S.l.], 2010. Disponível em: <http://developer.apple.com/library/ios/#documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/CoreApplication/CoreApplication.html#//apple_ref/doc/uid/TP40007072-CH3-SW14>. Acesso em: 30 maio. 2011.

IOS OPENGLES. **OpenGL ES programming guide for iOS:** OpenGL ES on iOS**.** [S.l.], 2010. Disponível em: <http://developer.apple.com/library/ios/#documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/OpenGLESontheiPhone/OpenGLESontheiPhone.html%23//apple_ref/doc/uid/TP40008793-CH101-SW1>. Acesso em: 27 maio. 2011.

KEPLER. **Lua:** conceitos básicos e API C. [S.l.], 2008. Disponível em: <http://www.keplerproject.org/docs/apostila_lua_2008.pdf>. Acesso em: 14 set. 2010.

KHRONOS. **OpenGL ES**. [S.l.], 2010. Disponível em: <http://www.khronos.org/opengles>. Acesso em: 15 set. 2010.

LUA ORG. **A linguagem de programação Lua**. [Rio de Janeiro], 2011. Disponível em: <http://www.lua.org/portugues.html>. Acesso em: 6 jun. 2011.

LUAGL. **LuaGL:** OpenGL binding for Lua 5. [S.l.], 2011. Disponível em: <http://luagl.sourceforge.net/>. Acesso em: 6 jun. 2011.

LUAOPENAL. **OpenAL binding:** project info. [S.l.], 2006. Disponível em: <http://luaforge.net/projects/luaopenal>. Acesso em: 6 jun. 2011.

OMG. **UML – Unified Modeling Language specification.** Version 2.0. [S.l.], 2005. Disponível em: <http://www.uml.org/>. Acesso em: 6 jun. 2011.

OPENGLES. **OpenGL ES 1.1 reference pages**. [S.l.], 2011. Disponível em: <http://www.khronos.org/opengles/sdk/1.1/docs/man/>. Acesso em: 6 jun. 2011.

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: 6 jun. 2011.

SIO2. **SIO2 interactive**. [S.l.], 2011. Disponível em: <http://sio2interactive.com/>. Acesso em: 6 jun. 2010.

TAKANO, Rafael H. **MJ3I: Um motor de jogos 3D para o iPhone OS.** 2009. 52 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

TIOBE. **TIOBE software:** Tiobe Index. [S.l.], 2011. Disponível em: <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html/>. Acesso em: 6 jun. 2011.

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: 18 set. 2010.

WAX. **Wax iPhone**. [S.l.], 2009. Disponível em: <http://github.com/probablycorey/wax>. Acesso em: 15 set. 2010.

WRIGHT, Richard S. **OpenGL superBible**. Indianopolis: Waite Group Press, 2000.

APÊNDICE A – Listagem dos comandos OpenGL ES 1.1 em LOGLES

No Quadro 22 é apresentado a lista de comandos com as funções do OpenGL ES 1.1 na linguagem de programação C e as mesmas funções do LOGLES na linguagem Lua.

OpenGL ES 1.1 (OPENGLES, 2011)	LOGLES (Lua OpenGL ES)
`void glActiveTexture(GLenum texture);`	`gl.ActiveTexture(texture);`
`void glAlphaFunc(GLenum func, GLclampf ref);`	`gl.AlphaFunc( func,  ref);`
`void glBindBuffer(GLenum target, GLuint buffer);`	`gl.BindBuffer(  target, buffer);`
`void glBindTexture(  GLenum target, GLuint texture);`	`gl.BindTexture( target, texture);`
`void glBlendFunc( GLenum sfactor, GLenum dfactor);`	`gl.BlendFunc(  sfactor, dfactor);`
`void glBufferData(  GLenum target, GLsizeiptr size, const GLvoid * data, GLenum usage);`	`gl.BufferData(  target, size, data, usage);`
`void glBufferSubData(  GLenum target, GLintptr offset, GLsizeiptr size, const GLvoid * data);`	`gl.BufferSubData(  target, ptr offset, ptr size, data);`
`void glClear(  GLbitfield mask);`	`gl.Clear( mask);`
`void glClearColor(  GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha);`	`gl.ClearColor(  red, green, blue, alpha);`
`void glClearDepthf(  GLclampf depth);`	`gl.ClearDepth(depth);`
`void glClearStencil(  GLint s);`	`gl.ClearStencil(s);`
`void glClientActiveTexture(  GLenum texture);`	`gl.ClientActiveTexture(texture);`
`void glClipPlanef(  GLenum plane, const GLfloat *equation);`	`gl.ClipPlane(  plane, equationArray);`
`void glColor4f(  GLfloat red, GLfloat green, GLfloat blue, GLfloat alpha);`	`gl.Color( red, green, blue, alpha);`
`void glColorMask( GLboolean red, GLboolean green, GLboolean blue, GLboolean alpha);`	`gl.ColorMask(red, green, blue, alpha);`
`void glColorPointer(  GLint size, GLenum type, GLsizei stride, const GLvoid * pointer);`	`gl.ColorPointer(colorArray);`
`void glCompressedTexImage2D(GLenum target, GLint level,`	`gl.CompressedTexImage(target, level, internalformat,`

GLenum internalformat, GLsizei width, GLsizei height, GLint border, GLsizei imageSize, const GLvoid * data);	width, height, border, imageSize, data);
void glCompressedTexSubImage2D(GLenum target, GLint level, GLint xoffset, GLint yoffset, GLsizei width, GLsizei height, GLenum format, GLsizei imageSize, const GLvoid * data);	gl.CompressedTexSubImage(target, level, xoffset, yoffset, width, height, format, imageSize, const GLvoid * data);
void glCopyTexImage2D(GLenum target, GLint level, GLenum internalformat, GLint x, GLint y, GLsizei width, GLsizei height, GLint border);	gl.CopyTexImage(level, internalformat, border, x, y, width, height);
void glCopyTexSubImage2D(GLenum target, GLint level, GLint xoffset, GLint yoffset, GLint x, GLint y, GLsizei width, GLsizei height)	gl.CopyTexSubImage(level, x, y, xoffset, width, yoffset, height)
void glCullFace(GLenum mode);	gl.CullFace(mode);
void glDeleteBuffers(GLsizei n, const GLuint * buffers);	gl.DeleteBuffers(buffersArray);
void glDeleteTextures(GLsizei n, const GLuint * textures);	gl.DeleteTextures(texturesArray) ;
void glDepthFunc(GLenum func);	gl.DepthFunc(func);
void glDepthMask(GLboolean flag);	gl.DepthMask(flag);
void glDepthRangef(GLclampf near, GLclampf far);	gl.DepthRange(znear, zfar);
void glDisable(GLenum cap);	gl.Disable(cap);
void glDisableClientState(GLenum array);	gl.DisableClientState(array);
void glDrawArrays(GLenum mode, GLint first, GLsizei count);	gl.DrawArrays(mode, first, count);
void glDrawElements(GLenum mode, GLsizei count, GLenum type, const GLvoid * indices);	gl.DrawElements(mode, indicesArray);
void glEnable(GLenum cap)	gl.Enable(cap)
void glEnableClientState(GLenum	gl.EnableClientState(array)

array)	
void glFinish(void);	gl.Finish();
void glFlush(void);	gl.Flush();
void glFogf(GLenum pname, GLfloat param);	gl.Fog(pname, param);
void glFrontFace(GLenum mode);	gl.FrontFace(mode);
void glFrustumf(GLfloat left, GLfloat right, GLfloat bottom, GLfloat top, GLfloat near, GLfloat far);	gl.Frustum(left, right, bottom, top, zNear, zFar);
void glGenBuffers(GLsizei n, GLuint * buffers);	gl.GenBuffers(range); → num
void glGenTextures(GLsizei n, GLuint * textures);	gl.GenTextures(n); → texturesArray
void glGetBooleanv(GLenum pname, GLboolean * params);	gl.GetBoolean(pname); → paramsArray
void glGetBufferParameteriv(GLenum target, GLenum pname, GLint * params);	gl.GetBufferParameteriv(target, pname); → paramsArray
void glGetClipPlanef(GLenum plane, GLfloat *equation);	gl.GetClipPlane(plane); → equationArray
GLenum glGetError(void);	gl.GetError(); → error
void glGetLightfv(GLenum light, GLenum pname, GLfloat * params);	gl.GetLight(light, pname); → paramsArray
void glGetMaterialfv(GLenum face, GLenum pname, GLfloat * params);	gl.GetMaterial(face, pname); → paramsArray
void glGetPointerv(GLenum pname, GLvoid ** params);	gl.GetPointer(pname, n); → valuesArray
const GLubyte * glGetString(GLenum name);	gl.GetString(name); → string
void glGetTexEnvfv(GLenum target, GLenum pname, GLfloat * params);	gl.GetTexEnv(pname); → paramsArray
void glGetTexParameterfv(GLenum target, GLenum pname, GLfloat * params);	gl.GetTexParameter(target, pname); → paramsArray
void glHint(GLenum target, GLenum mode);	gl.Hint(target, mode);
GLboolean glIsBuffer(GLuint buffer);	gl.IsBuffer(buffer); → true/false
GLboolean glIsEnabled(GLenum cap);	gl.IsEnabled(cap); → true/false
GLboolean glIsTexture(GLuint texture);	gl.IsTexture(texture); → true/false
void glLightf(GLenum light,	gl.Light(light,

GLenum pname, GLfloat param);	pname, param);
void glLightModelf(GLenum pname, GLfloat param)	gl.LightModel(pname, param)
void glLineWidth(GLfloat width);	gl.LineWidth(width);
void glLoadIdentity(void);	gl.LoadIdentity();
void glLoadMatrixf(const GLfloat * m);	gl.LoadMatrix(mArray);
void glLogicOp(GLenum opcode);	gl.LogicOp(opcode);
void glMaterialf(GLenum face, GLenum pname, GLfloat param);	gl.Material(face, pname, param);
void glMatrixMode(GLenum mode);	gl.MatrixMode(mode);
void glMultMatrixf(const GLfloat * m);	gl.MultMatrix(mArray);
void glMultiTexCoord4f(GLenum target, GLfloat s, GLfloat t, GLfloat r, GLfloat q);	gl.MultiTexCoord(target, s, t, r, q);
void glNormal3f(GLfloat nx, GLfloat ny, GLfloat nz);	gl.Normal(nx, ny, nz);
void glNormalPointer(GLenum type, GLsizei stride, const GLvoid * pointer);	gl.NormalPointer(normalArray2);
void glOrthof(GLfloat left, GLfloat right, GLfloat bottom, GLfloat top, GLfloat near, GLfloat far);	gl.Ortho(left, right, bottom, top, zNear, zFar);
void glPixelStorei(GLenum pname, GLint param);	gl.PixelStore(pname, param);
void glPointParameterf(GLenum pname, GLfloat param);	gl.PointParameter(pname, param);
void glPointSize(GLfloat size);	gl.PointSize(size);
void glPointSizePointerOES(GLenum type, GLsizei stride, const GLvoid * pointer);	gl.PointSizePointerOES(type, stride, pointer);
void glPolygonOffset(GLfloat factor, GLfloat units);	gl.PolygonOffset(factor, units);
void glPopMatrix(void);	gl.PopMatrix();
void glPushMatrix(void);	gl.PushMatrix();
void glReadPixels(GLint y, GLint y, GLsizei width, GLsizei height, GLenum format, GLenum type,	gl.ReadPixels(y, y, width, height, format); → pixelsArray

GLvoid * pixels);	
void glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z);	gl.Rotate(angle, x, y, z);
void glSampleCoverage(GLclampf value, GLboolean invert);	gl.SampleCoverage(value, invert);
void glScalef(GLfloat x, GLfloat y, GLfloat z);	gl.Scale(x, y, z);
void glScissor(GLint x, GLint y, GLsizei width, GLsizei height);	gl.Scissor(x, y, width, height);
void glShadeModel(GLenum mode);	gl.ShadeModel(mode);
void glStencilFunc(GLenum func, GLint ref, GLuint mask);	gl.StencilFunc(func, ref, mask);
void glStencilMask(GLuint mask);	gl.StencilMask(mask);
void glStencilOp(GLenum fail, GLenum zfail, GLenum zpass);	gl.StencilOp(fail, zfail, zpass);
void glTexCoordPointer(GLint size, GLenum type, GLsizei stride, const GLvoid * pointer);	gl.TexCoordPointer(vArray2);
void glTexEnvf(GLenum target, GLenum pname, GLfloat param);	gl.TexEnv(pname, param);
void glTexImage2D(GLenum target, GLint level, GLint internalformat, GLsizei width, GLsizei height, GLint border, GLenum format, GLenum type, const GLvoid * pixels);	gl.TexImage2D(level, internalformat, width, height, border, format, type, pixelsUserData);
void glTexParameterf(GLenum target, GLenum pname, GLfloat param);	gl.TexParameter(target, pname, param);
void glTexSubImage2D(GLenum target, GLint level, GLint xoffset, GLint yoffset, GLsizei width, GLsizei height, GLenum format, GLenum type, const GLvoid * pixels);	gl.TexSubImage2D(target, level, xoffset, yoffset, width, height, format, type, pixelsUserData);
void glTranslatef(GLfloat x, GLfloat y, GLfloat z);	gl.Translate(x, y, z);
void glVertexPointer(GLint size,	gl.VertexPointer(vertexArray2);

GLenum type, GLsizei stride, const GLvoid * pointer);	
void glViewport(GLint x, GLint y, GLsizei width, GLsizei height);	gl.Viewport(x, y, width, height);

Quadro 22 – Listagem de comando OpenGL ES e LOGLES