UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

MJ3I-PA - AMBIENTE VIRTUAL 3D PARA VISUALIZAÇÃO DE BRAÇOS ARTICULADOS NO IOS

HEITOR AUGUSTO SCHMITT

BLUMENAU
2012

2012/2-13

HEITOR AUGUSTO SCHMITT

MJ3I-PA - AMBIENTE VIRTUAL 3D PARA VISUALIZAÇÃO DE BRAÇOS ARTICULADOS NO 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. Dalton Solano dos Reis, M. Sc. - Orientador

BLUMENAU
2012

2012/2-13

MJ3I-PA - AMBIENTE VIRTUAL 3D PARA VISUALIZAÇÃO DE BRAÇOS ARTICULADOS NO IOS

Por

HEITOR AUGUSTO SCHMITT

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. Mauro Marcelo Mattos, Dr. – FURB
Membro:	Prof. Aurélio Faustino Hoppe, M. Sc. – FURB

Blumenau, 12 de dezembro de 2012

Dedico este trabalho a todos que me ajudaram de alguma forma na realização deste. Especialmente meus pais, demais familiares e amigos.

AGRADECIMENTOS

À minha família, por todo o suporte, incentivo e cobranças.

Aos meus amigos, pelo companheirismo e ajuda durante o curso.

Ao meu orientador, Dalton Solano dos Reis, por toda a atenção, pelo comprometimento e por não deixar de mostrar apoio mesmo quando o trabalho não caminhava da forma esperada.

Aquele que tiver paciência terá o que deseja.

Benjamin Franklin

RESUMO

Esta monografia apresenta a construção de uma biblioteca gráfica, baseada na já existente biblioteca V-ART, que utiliza o conceito de personagens articulados para dispositivos móveis da plataforma iOS, juntamente com um aplicativo que utiliza a biblioteca para desenhar um braço mecânico articulado. A biblioteca disponibiliza ferramentas para a criação de personagens articulados simples. O desenho destes personagens foi realizado com a biblioteca OpenGL ES 2.0 disponível no iOS. A interação com o aplicativo ocorre através de toques na tela, sendo possível selecionar e rotacionar cada uma das articulações. Por fim foi demonstrado o desempenho prático com a aplicação rodando em um dispositivo da plataforma iOS.

Palavras-chave: Personagens articulados. iOS. Motor gráfico. OpenGL.

ABSTRACT

This monography describes a graphic library, based on a pre-existing library called V-ART, which uses the concept of articulated characters for mobile devices on iOS platform, along with an application that uses the library to draw an articulated mechanical arm. The library provides tools for creating simple articulated characters. The design of these characters was done with the OpenGL ES 2.0 library available in iOS. The interaction with the application occurs by tapping the screen, making it possible to select and rotate each of the articulations. Finally the practical performance was demonstrated with the application running on an iOS platform device.

Key Words: Articulated characters. IOS. Graphic engine. OpenGL.

LISTA DE ILUSTRAÇÕES

Figura 1 – Arquitetura iOS 17
Figura 2 – Principais classes da biblioteca V-ART 21
Figura 3 - Diagrama do pipeline de programação de sombras do OpenGL ES 2.0 24
Quadro 1 – Código de um `vertex shader` 25
Figura 4 – OpenGL ES 2.0 `vertex shader` 26
Quadro 2 – Código de um `fragment shader` 26
Figura 5 – OpenGL ES 2.0 `fragment shader` 27
Figura 6 – MJ3I em funcionamento 28
Figura 7 – Tela de edição do Unity 3 29
Figura 8 – Diagrama de casos de uso 32
Quadro 3 – Caso de uso `UC01` 32
Quadro 4 – Caso de uso `UC02` 33
Quadro 5 – Caso de uso `UC03` 33
Quadro 6 – Caso de uso `UC04` 34
Figura 9 – Principais classes da aplicação 35
Figura 10 – Diagrama de sequencia `Mover articulações` 36
Quadro 7 – Método `viewDidLoad` da classe `ViewController` 38
Quadro 8 – Método `setupGL` da classe `ViewController` 39
Quadro 9 – Método `loadShaders` da classe `ViewController` 39
Quadro 10 – Implementação `Fragment Shader` 40
Quadro 11 – Implementação `Vertex Shader` 40
Quadro 12 – Objetos utilizados na aplicação 40
Quadro 13 – Método `setupScene` da classe `ViewController` 41
Quadro 14 – Método `glkViewdrawInRect` da classe `ViewController` 42
Quadro 15 – Método `DrawInstanceOGL` da classe `MeshObject` 43
Figura 11 – Aplicação executando no simulador 44
Figura 12 – Tela do dispositivo dividida em segmentos 45
Figura 13 – Objetos após aplicação do caso de uso `UC01` 46
Figura 14 – Câmera rotacionada após aplicação do caso de uso `UC02` 46
Figura 15 – Aplicação de 30 braços executando no simulador 49

LISTA DE TABELAS

Tabela 1 – Resultados obtidos na aplicação com N nodos 49
Tabela 2 – Comparativo entre ferramentas 50

LISTA DE SIGLAS

2D – Duas Dimensões

3D – Três Dimensões

API – *Application Programming Interface*

CPU – *Central Processing Unit*

DNS – *Domain Name System*

DOF – *Degree Of Freedom*

DTD – *Document Type Definition*

FPS – *Frames Per Second*

GHz – *Giga Hertz*

GPLv2 – *General Public Licence Version 2*

GPS – *Global Positioning System*

GPU – *Graphics Processing Unit*

IK – *Inverse Kinematics*

KB – Kylo Bytes

MB – *Mega Bytes*

OpenGL – *Open Graphics Library*

OpenGL ES – *Open Graphics Library for Embedded Systems*

OS – *Operational System*

SDK – *Software Development Kit*

SO – Sistema Operacional

SQL – *Structured Query Language*

UML – *Unified Modeling Language*

XML – *eXtensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO 13
1.1 OBJETIVOS DO TRABALHO 14
1.2 ESTRUTURA DO TRABALHO 14
2 FUNDAMENTAÇÃO TEÓRICA 15
2.1 MOTORES PARA JOGOS 15
2.2 IOS 14
2.2.1 Frameworks 18
2.2.1.1 Foundation 18
2.2.1.2 UIKit 19
2.2.1.3 Core Graphics 19
2.2.1.4 Quartzcore 20
2.3 V-ART E PERSONAGENS ARTICULADOS 20
2.3.1 V-ART 20
2.3.1.1 Biblioteca V-ART 21
2.3.2 Personagens articulados 22
2.4 OPENGL ES 23
2.4.1 `Vertex shader` 25
2.4.2 `Vertex shader` 26
2.5 TRABALHOS CORRELATOS 27
2.5.1 Motor de jogos 3D para iPhone OS (TAKANO, 2009) 27
2.5.2 Unity 3 28
3 DESENVOLVIMENTO 30
3.1 DESENVOLVIMENTO IOS 30
3.2 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHO 30
3.3 ESPECIFICAÇÃO 31
3.3.1 Casos de uso 31
3.3.1.1 Mover articulações 32
3.3.1.2 Rotacionar câmera 33
3.3.1.3 Alterar material dos objetos 33
3.3.1.4 Adicionar e movimentar objetos 33
3.3.2 Diagrama de classes 34

3.3.2.1 Aplicação 35
3.3.3 Diagrama de sequencia 36
3.4 IMPLEMENTAÇÃO 47
3.4.1 Técnicas e ferramentas utilizadas 37
3.4.1.1 Interface da aplicação 37
3.4.1.2 Composição da cena 38
3.4.2 Operacionalidade da implementação 44
3.5 RESULTADOS E DISCUSSÃO 47
3.5.1 Testes de desempenho (FPS) e alocação de memória 47
3.5.1.1 Testes com aplicação de 3 nodos 48
3.5.1.2 Testes com aplicação de N nodos 48
3.5.2 Comparativo com trabalhos correlatos 50
4 CONCLUSÕES 51
4.1 EXTENSÕES 52
REFERÊNCIAS BIBLIOGRÁFICAS 53

1 INTRODUÇÃO

Inicialmente o entretenimento através de jogos foi popularizado pelos consoles, mais conhecidos como video-games. Eles continuam a proporcionar diversão, mas recentemente tiveram que se reinventar para não perderem espaço nas concorridas horas da rotina moderna. O renascimento foi possível graças à mudanças na forma como jogador interage com o jogo. Um exemplo, talvez o mais clássico e conhecido, é o Kinect, aparelho criado para utilização junto ao Xbox 360, que através de mapeamento das articulações do corpo humano, transforma o próprio jogador em um controle.

A revolução no mundo dos video-games foi impulsionada pela rápida disseminação de smartphones como forma de entretenimento. Os jogos para celulares são geralmente simples e viciantes, daqueles que se aprende a jogar em minutos, mas leva-se dias para largá-los e procurar um novo que proporcione a mesma diversão.

Smartphones são telefones celulares com acesso à Internet, serviços de voz, correio eletrônico, navegadores Web, câmera capaz de fotografar e filmar, tocador de música, visualizador de vídeos, vídeo-chamada, aplicativos, entre outras funcionalidades avançadas (PCMAG, 2012a).

Além dos smartphones, hoje tem-se o segmento de tablets, que cresce mais e mais a cada dia, proporcionando quase todas as utilidades de um celular, porém com um display maior, facilitando determinados trabalhos. Tablets são computadores de uso geral, contidos em um único painel. A sua característica marcante é o uso de telas sensíveis ao toque como forma de entrada de dados. Antes manuseadas por canetas especiais, hoje as tablets são operadas somente pelos dedos (PCMAG, 2012b).

É difícil hoje citar smartphones, tablets, diversão e entretenimento sem comentar a Apple. Através do iPhone e do iPad, com uma grande ajuda da sua loja de aplicativos e também do já fortalecido segmento de computadores pessoais, ela foi uma das responsáveis por revolucionar o conceito de diversão. Através destes produtos, ela introduziu no mercado o iOS, sistema operacional por eles utilizado.

Seguindo a tendência de crescimento e popularização dos dispositivos móveis avançados e a crescente demanda por jogos, desenvolveu-se um Motor para Jogos 3D na plataforma IOS (MJ3I). Segundo Eberly (2001, p. 3), os motores de jogos são bibliotecas de desenvolvimento responsáveis pelo gerenciamento do jogo, das imagens, do processamento de entrada de dados e cálculos relacionados aos fenômenos físicos.

A representação de braços articulados é realizada através de uma aplicação para a plataforma iOS que simula um ambiente virtual 3D, representando uma base e um braço mecânico articulado. Este tipo de braço é aquele que simula a existência de articulações, juntas que definem o grau de liberdade e movimentação do braço.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é representar personagens articulados 3D na plataforma iOS, criando para tal um ambiente virtual 3D.

Os objetivos específicos do trabalho são:

a) portar o aplicativo *Virtual Articulations for Virtual Reality* (V-ART) do Mac OS para o iOS;
b) disponibilizar um ambiente virtual 3D para visualização de personagens articulados.

1.2 ESTRUTURA DO TRABALHO

A estrutura deste trabalho está dividida em quatro capítulos, sendo o primeiro capítulo responsável pela apresentação geral do trabalho.

O segundo capítulo engloba a fundamentação teórica necessária para o entendimento deste trabalho.

O terceiro capítulo apresenta a etapa de desenvolvimento da aplicação na plataforma iOS, casos de uso, diagramas de classe e toda a especificação que define a aplicação. Ainda no terceiro capítulo são apresentados os principais trechos da implementação, os resultados obtidos e discussões geradas durante a etapa de desenvolvimento do trabalho.

Por fim, o quarto capítulo contém as conclusões do trabalho e sugestões para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

A seção 2.1 comenta sobre os motores para jogos, o que são e qual a sua utilização. Na seção 2.2 é apresentada a plataforma iOS, suas funcionalidades e sua arquitetura. A seção 2.3 apresenta o V-ART e personagens articulados. Por fim a seção 2.4 traz trabalhos correlatos ao proposto.

2.1 MOTORES PARA JOGOS

Segundo Eberly (2005, p. 149), em seu nível mais baixo, um motor para jogos tem a responsabilidade de desenhar objetos visíveis à um observador. O programador geralmente utiliza uma *Application Programming Interface* (API) gráfica, como *Open Graphics Library* (OpenGL) ou Direct3D para implementar um renderizador, que possui a tarefa de desenhar corretamente os objetos.

O Direct3D faz parte do pacote DirectX, da Microsoft. É um grupo de tecnologias designadas a tornar um computador rodando Windows em uma plataforma ideal para executar aplicações ricas em recursos multimídia, como gráficos coloridos, vídeos, animações 3D e áudio (MICROSOFT, 2012a).

OpenGL é a principal biblioteca para desenvolvimento de aplicações gráficas 2D e 3D. Desde a sua introdução, em 1992, tornou-se a API mais utilizada e suportada da indústria, trazendo milhares de aplicações para uma grande variedade de plataformas. O OpenGL fomenta inovação e velocidade no desenvolvimento, incorporando um largo conjunto de funções renderizadoras, efeitos especiais, mapeamento de texturas, entre outras funções de visualização (KHRONOS GROUP, 2012d).

O OpenGL *for Embedded Systems* (OpenGL ES) é uma subseção da biblioteca OpenGL, destinada ao desenvolvimento de aplicações gráficas para dispositivos móveis. Ela consiste de conjuntos de rotinas extraídas do OpenGL, criando uma flexível e poderosa interface de baixo nível entre software e aceleração gráfica. O OpenGL ES atualmente se encontra na versão 3.0 (KHRONOS GROUP, 2012b).

Ainda segundo Eberly (2005, p. 149), em determinadas plataformas, caso não exista aceleração por hardware, ou se uma API padrão estiver indisponível, o programador pode

implementar o sistema gráfico inteiro para executar na *Central Processing Unit* (CPU). O resultado é denominado renderização por software (*software renderer*). Ainda que os hardwares gráficos de hoje sejam poderosos a ponto de eliminarem a necessidade da renderização por software, esta característica permanece de suma importância, principalmente nos sistemas embarcados, com capacidades gráficas limitadas, como celulares e *tablets*.

Desta forma, pode-se afirmar que o conceito de um motor de jogos é relativamente simples. Ele deve abstrair os detalhes comuns a todos os jogos, como renderização, conceitos físicos e entrada de dados, além de tratar problemas como movimentação de objetos e gerenciamento de grafos de cena, para que os desenvolvedores possam focar-se nos detalhes que transformam um jogo comum em um jogo único (WARD, 2008).

2.2 IOS

O sistema operacional, desenvolvido pela Apple Inc., originalmente para utilização no iPhone, foi posteriormente portado para o iPod Touch e para o iPad. Desta forma, em junho de 2010, o inicialmente chamado iPhone OS foi renomeado e passou a ser chamado de iOS (PATEL, 2010).

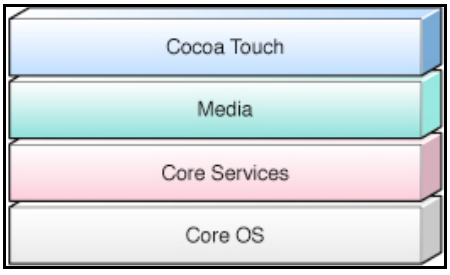
Do ponto de vista de um usuário do iOS, a palavra-chave é a simplicidade. Desde o primeiro contato o usuário sabe como utilizá-lo, graças ao modo intuitivo como o sistema aborda as funcionalidades e à interface com suporte a múltiplos toques. Até mesmo as tarefas mais complicadas, como edição de vídeo, inicializar uma vídeo-chamada ou alternar entre aplicativos, torna-se fácil com o iOS (APPLE, 2012a).

Analisando a visão de um programador, o iOS pode ser considerado como um ecossistema para desenvolvimento. O iOS *Software Development Kit* (SDK), combinado com as ferramentas oferecidas pelo Xcode facilita a tarefa de criação de aplicativos, enquanto a disponibilidade de publicá-los na loja de aplicativos da Apple faz com que seja fácil chegar ao consumidor final (APPLE, 2012b).

O sucesso alcançado pelo sistema operacional pode ser atribuído à estrutura sob a qual ele é construído. Derivado do núcleo do Mac OS, sistema operacional presente nos computadores da Apple, o iOS foi construído para ser compacto e eficiente, tirando máximo proveito do hardware oferecido pelo iPhone, iPod Touch e iPad. Algumas das tecnologias compartilhadas entre iOS e Mac OS incluem o Kernel, técnicas de compartilhamento e de

rede, além de compiladores C/C++ e Objective-C, para uma melhor performance (APPLE, 2012b).

A arquitetura presente no iOS é apresentada na Figura 1.



Fonte: Apple (2012c).

Figura 1 – Arquitetura iOS

A camada *Core OS* contém os recursos de baixo nível, que servem como base para a construção das outras tecnologias. Mesmo que estes recursos não sejam utilizados diretamente num determinado aplicativo, eles provavelmente estão sendo usados por outros *frameworks* (APPLE, 2012d).

Alguns dos recursos presentes nesta camada são a capacidade de efetuar cálculos, comunicar-se com acessórios plugados ao dispositivo ou conectados via *bluetooth*, segurança dos dados manuseados pelas aplicações, além dos recursos básicos de qualquer sistema como entrada e saída de dados, divisão por tarefas, compartilhamento e rede, acesso ao sistema de arquivos, alocação de memória, informações de localização e serviços *Domain Name System* (DNS) (APPLE, 2012d).

Na camada *Core Services* estão presentes os serviços de sistema que todas as aplicações fazem uso. Estes serviços são o controle de multitarefa, suporte à *Structured Query Language* (SQL) e à XML, implementação de agenda, *Global Positioning System* (GPS), entre outros (APPLE, 2012e).

Presentes na camada *Media* estão as tecnologias gráficas, de áudio e de vídeo, com o objetivo de proporcionar a melhor experiência multimídia para dispositivos móveis. As tecnologias presentes nesta camada são o suporte nativo à vetores 2D, animações, renderização 2D e 3D através do *Open Graphics Library for Embedded Systems* (OpenGL ES) e suporte à grande maioria dos formatos de imagens. Na questão do áudio, a camada oferece fácil acesso à biblioteca do usuário, permitindo a reprodução de músicas, gravação de áudio e suporte aos mais conhecidos formatos de arquivos de áudio. Por fim, a seção vídeo oferece fácil integração com a câmera presente no dispositivo, para captura de vídeos, manipulação de mídias e facilidade de reprodução de vídeos nas aplicações (APPLE, 2012f).

A camada *Cocoa Touch* contém os principais *frameworks* para a construção dos

aplicativos. Esta camada define a infraestrutura básica de uma aplicação e o suporte para as tecnologias chaves, como multitarefa, entrada de dados via toques na tela e sistema de notificações (APPLE, 2012g).

2.2.1 Frameworks

Um *framework* é um diretório hierárquico que encapsula recursos compartilhados, bibliotecas, arquivos de interface, imagens, cabeçalhos de classes e documentação em um único pacote. Múltiplas aplicações podem fazer uso desses recursos simultaneamente. O sistema carrega um *framework* na memória e compartilha essa única cópia com todas as aplicações (APPLE, 2012i).

Frameworks servem o mesmo propósito de bibliotecas compartilhadas estática ou dinamicamente, que é prover uma série de rotinas que podem ser chamadas por uma aplicação para execução de uma tarefa específica. *Frameworks* oferecem as seguintes vantagens perante as bibliotecas compartilhadas (APPLE, 2012i):

a) agrupam recursos relacionados, facilitando o processo de instalação, desinstalação e localização desses recursos;
b) incluem uma variedade maior de tipos de recursos, por exemplo cabeçalhos e documentação relevantes;
c) múltiplas versões de um *framework* podem ser incluídas dentro de um mesmo pacote, mantendo compatibilidade com programas mais antigos;
d) apenas uma cópia dos recursos de um *framework* residem fisicamente na memória em qualquer espaço de tempo, independente de quantos processos estejam utilizando esses recursos. Esse tipo de compartilhamento reduz o consumo de memória e melhora a performance da aplicação.

2.2.1.1 Foundation

O *framework* Foundation define uma camada base para as classes desenvolvidas em Objective-C, fornecendo as classes de objetos primitivos. Além disso, ele introduz vários paradigmas que agregam funcionalidade às classes desenvolvidas. Ele foi desenvolvido com

uma série de objetivos em mente (APPLE, 2012j):

a) oferecer uma pequena quantidade de classes com utilidades básicas;
b) facilitar o desenvolvimento de um software, introduzindo convenções consistentes para tarefas como liberação de memória;
c) oferecer suporte à strings *Unicode*, persistência e distribuição de objetos;
d) oferecer um nível de independência de SO, melhorando a portabilidade.

O Foundation inclui a classe objeto raíz (`NSObject`), classes que representam tipos básicos de dados, como strings e arrays de bytes, coleções para armazenamento de objetos, classes que representam datas e portas de comunicação (APPLE, 2012j).

2.2.1.2 UIKit

O *framework* UIKit é responsável por fornecer as classes necessárias para construção e gerenciamento de interfaces de usuário da aplicação. Ela fornece um objeto aplicação, classes para tratativas de eventos, um modelo para desenho de interfaces, janelas, visões e controles designados especificamente para o gerenciamento de uma interface de toques na tela (APPLE, 2012k).

2.2.1.3 Core Graphics

O *framework* Core Graphics é uma API baseada em C, que por sua vez se baseia no motor de desenhos Quartz. Ele fornece renderização 2D de baixo nível, com excelente performance e apresentação do resultado com grande fidelidade. Esse *framework* é utilizado para gerenciar desenho baseado em caminho, transformações, manipulação de cores, renderização fora da tela, padrões, gradientes, *shading*, criação de imagens e manipulação de documentos PDF (APPLE, 2012l).

2.2.1.4 Quartzcore

O *framework* Quartzcore é o responsável por oferecer suporte à processamento de imagens e manipulação de imagens de vídeo (APPLE, 2012m). Ele possui classes de camada, que disponibilizam conteúdo para visualização, classes de animação, layout e classes de transação que agrupa mudanças nas camadas para realização de uma única atualização (APPLE, 2012n).

2.3 V-ART E PERSONAGENS ARTICULADOS

A seguir serão apresentados o *framework* V-ART e personagens articulados.

2.3.1 V-ART

O V-ART é um *framework* desenvolvido em C++ para facilitar a criação de programas com ambientes 3D, em especial os que contém humanóides. O V-ART é um software livre, distribuído sob a *General Public License Version 2* (GPLv2). Ele distingue-se de outros *frameworks* semelhantes como Open Inventor (VSG, 2012), PLIB (PLIB, 2012) e VRJuggler (VRJUGGLER, 2012) por ser inteiramente orientado a objetos, possuir um sistema de suporte a animações e permitir a representação de articulações biologicamente corretas. O V-ART tem suas origens no projeto *Visualization and interaction with Virtual Patients* (VPAT) (VPAT, 2012) e era conhecido por este nome até maio de 2006, quando foi renomeado. Ele é projetado para ser multiplataforma (Linux, Windows e outras) e independente de API gráfica (OpenGL, Direct3D, entre outras) (V-ART, 2009a).

Sobre modelos 3D, o V-ART pode parcialmente importar modelos no formato `Wavefront OBJ`, além de oferecer recursos para importação de humanóides e modelos articulados, em geral, descritos em XML, via *Document Type Definition* (DTD) próprio (V-ART, 2009b).

2.3.1.1 Biblioteca V-ART

As classes da biblioteca são responsáveis pelas regras de negócio envolvendo a criação da cena e seus objetos, bem como por renderizar os mesmos. As principais classes da biblioteca estão representadas na Figura 2.

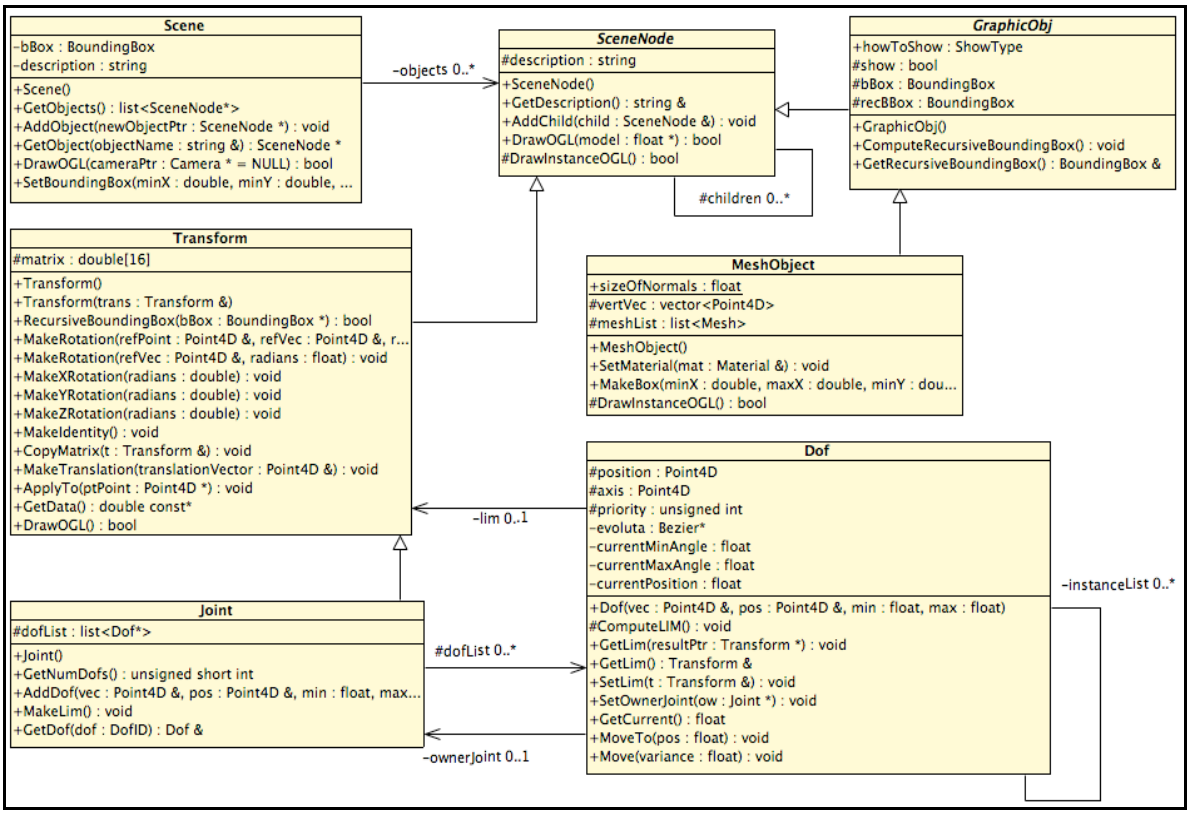


Figura 2 - Principais classes da biblioteca V-ART

A criação de um aplicativo inicia-se pela classe `Scene`, que armazena uma lista de `SceneNode`, essa lista representa os objetos a serem desenhados na cena. O método `drawOGL` é responsável por renderizar recursivamente os objetos que serão visualizados na tela do dispositivo.

A classe `SceneNode` é a base de todo objeto renderizado, como formas (`GraphicObj`). Cada nodo instanciado pode possuir filhos, formando um grafo de cena hierárquico, o conjunto de nodos compõem a cena. Estes nodos são desenhados através do método recursivo `drawOGL`, que executa o método `drawInstanceOGL` para se renderizar e em seguida chama o método `drawOGL` de cada um de seus filhos para renderizar os mesmos.

A classe `GraphicObj` é a base para objetos que possuem formas. Ela é responsável por informações como visibilidade do objeto gráfico, a *bounding box* do objeto, a *bounding box* recursiva (que envolve o objeto e seus filhos) e a forma que o objeto deve ser desenhado. O

método `computeRecursiveBoundingBox` define a *bounding box* recursiva e o método `getRecursiveBoundingBox` é utilizado para retornar a mesma.

Uma das especializações de `GraphicObj` é a classe `MeshObject`, a representação de um objeto gráfico composto por polígonos. O método `makeBox` é utilizado para a criação de objetos 3D retangulares e utilizado na construção dos objetos renderizados no aplicativo demonstrado no capítulo 3.3.2.2.

A classe `Joint` é responsável por representar as articulações. Ela é considerada um tipo especial de transformação, pois trabalha com a complexidade de articulações reais. `Joint` possui uma lista ordenada da classe `Dof`. As transformações executadas em um `Dof` influenciam os `Dofs` posteriores.

A classe `Dof` representa um grau de liberdade (do inglês *degree of freedom*) e é o componente básico de uma articulação (`Joint`). *Degree Of Freedom* (DOF) é um eixo de rotação que pode mover-se ao longo de uma curva 3D, possuindo esta rotação um alcance limitado que pode mudar de acordo com elementos externos (VART DOCS, 2012).

`Transform` é a classe que serve como base da classe `Joint` e representa as transformações geométricas. Ela instancia uma matriz de transformação que é aplicada também ao renderizar seus objetos filhos.

2.3.2 Personagens Articulados

Um personagem articulado é qualquer personagem dotado de articulações. No caso do V-ART, no qual são utilizados modelos humanóides, as articulações são pescoço, ombro, cotovelo, punho, joelho e tornozelo (MACIEL, 2012, p. 23).

A descrição formal do movimento realizado por estes personagens e suas articulações é chamada de cinemática. O objetivo é identificar as forças atuantes no objeto e aplicar a cinemática para determinar o movimento do mesmo. Idealmente, a posição do objeto pode ser determinada a qualquer instante, através dos seguintes passos: identificar as forças, desenvolver equações de movimento, determinar a posição do centro de massa e a orientação do objeto a todos os instantes (VEPA, 2009).

Em animação 3D, *Inverse Kinematics* (IK) é uma técnica que permite movimentação automática de um elemento, assim como de seus membros articulados, de uma maneira pré-definida e realista. A grande dificuldade no aprendizado da cinemática inversa está no fato de

que existem múltiplas soluções para um mesmo problema. Exemplificando, caso seja necessário movimentar uma mão em direção à maçaneta de uma porta, existem infinitos movimentos diferentes que podem ser realizados nas diversas articulações para alcançar o efeito desejado (D'SOUZA; SCHAAL; VIJAYAKUMAR, 2001).

A cinemática direta, por sua vez, estuda o posicionamento e orientação de coordenadas para definir a posição final de um modelo, levando em consideração os *Degrees Of Freedom* (DOFs) de cada membro articulado. O DOF nada mais é do que a própria articulação, um ponto onde é possível dobrar ou rotacionar o membro. Utilizando os DOFs mapeados é possível estabelecer uma parametrização, caracterizando cada ligação num sistema padrão de coordenadas. Para encontrar a localização final de cada membro/articulação são utilizados os DOFs e suas informações de orientação e coordenadas (VEPA, 2009).

2.4 OPENGL ES

O cenário da indústria atual é dominado por duas bibliotecas gráficas 3D principais, DirectX e OpenGL. DirectX é o padrão utilizado para desenvolvimento de aplicação gráficas no sistema operacional Windows (MICROSOFT, 2012b). Enquanto o OpenGL, por ser multi-plataforma, pode ser utilizado no desenvolvimento de aplicações gráficas para Linux, Mac OS X e Microsoft Windows (WRIGHT JR. et al., 2010).

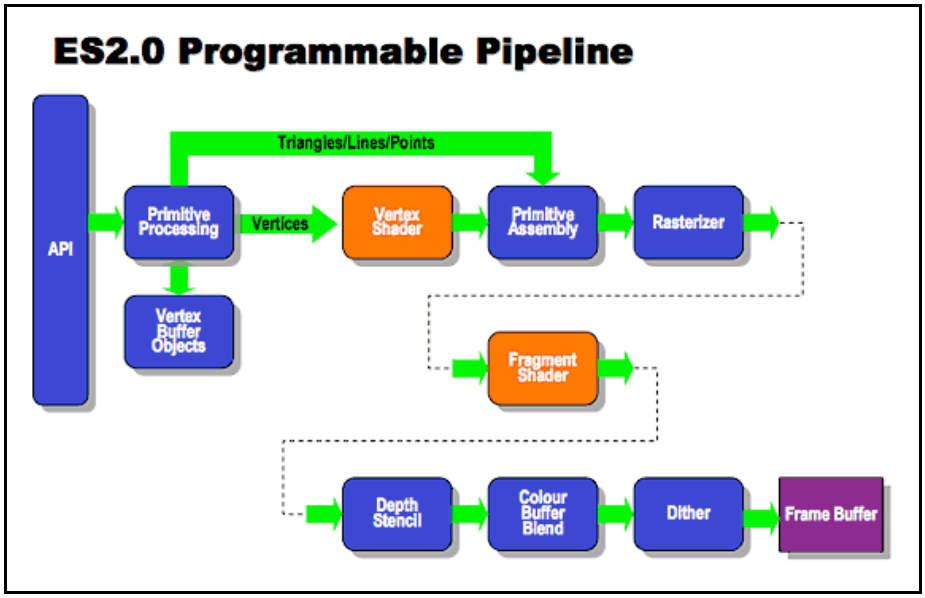
Devido à massiva adoção da biblioteca OpenGL para o desenvolvimento de aplicações *desktop* (MUNSHI; GINSBURG; SHREINER, 2009, p. 1), houve a necessidade de desenvolvimento de uma biblioteca semelhante, afim de atender a crescente demanda de aplicações gráficas para dispositivos portáteis e embarcados, a OpenGL ES.

Tais dispositivos possuem limitada capacidade de processamento e memória, além de preocupação com o consumo de energia, dessa forma, na criação da biblioteca OpenGL ES, foi dado um grande foco nessas limitações. A biblioteca removeu funções redundantes, limitou o consumo de energia e melhorou o desempenho através de comandos específicos de sombreamento, combinados com a utilização de `shaders`.

OpenGL ES é uma API multi-plataforma para funções gráficas de duas dimensões (2D) e 3D. Essas funções são utilizadas no desenvolvimento de aplicações para dispositivos móveis e sistemas embarcados. (KHRONOS GROUP, 2012b). Foi criada e é mantida pelo grupo Khronos, fundado em Janeiro de 2000 e focado na criação de padrões e APIs abertos

(KHRONOS GROUP, 2012a). A biblioteca OpenGL ES encontra-se na versão 3.0.

A Figura 3 demonstra as fases da OpenGL ES 2.0. `Vertex shader` e `fragment shader`, destacados em laranja, são os dois estágios de processamento do `shader`.



Fonte: Khronos Group (2011c).

Figura 3 – Diagrama do *pipeline* de programação de sombras do OpenGL ES 2.0

Esta versão, 2.0, da biblioteca, é baseada em `shader`, isto significa que nada é renderizado sem que um `vertex` e `fragment shader` válidos tenham sido carregados (MUNSHI; GINSBURG; SHREINER, 2009, p. 21/27).

"`Shaders` são pequenos pedaços de código executados na GPU e divididos em duas categorias: `vertex` e `frament shaders`" (RIDEOUT, 2010, p. 34). Os `shaders` são codificados em uma linguagem chamada OpenGL *Shading Language* (GLSL). Essa linguagem é semelhante à linguagem C, com a diferença de que o código gerado não é compilado juntamente com os demais fontes, mas sim em tempo de execução pela própria biblioteca (RIDEOUT, 2010, p. 34).

Estes `shaders` realizam os cálculos de sombra e luz, por exemplo, e fazem esse processamento na GPU dos dispositivos móveis, otimizando dessa forma o consumo de processador e energia (WRIGHT JR. et al., 2010). A GPU possui alta capacidade de processamento paralelo, permitindo que milhares de instâncias de `shaders` possam ser executadas simultaneamente (RIDEOUT, 2010, p. 34).

2.4.1 Vertex Shader

O `vertex shader` é utilizado para processamento de vértices, com o propósito de computar incidência de luz, transformar matrizes de pontos e gerar ou transformar coordenadas de texturas (MUNSHI; GINSBURG; SHREINER, 2009, p. 38). Esse processamento é realizado para todos os vértices da aplicação, um de cada vez (KHRONOS GROUP, 2012e).

O Quadro 1 demonstra um `vertex shader` utilizado para renderizar os vértices com a mesma cor de origem e com a posição resultante do produto entre as matrizes de projeção e visão.

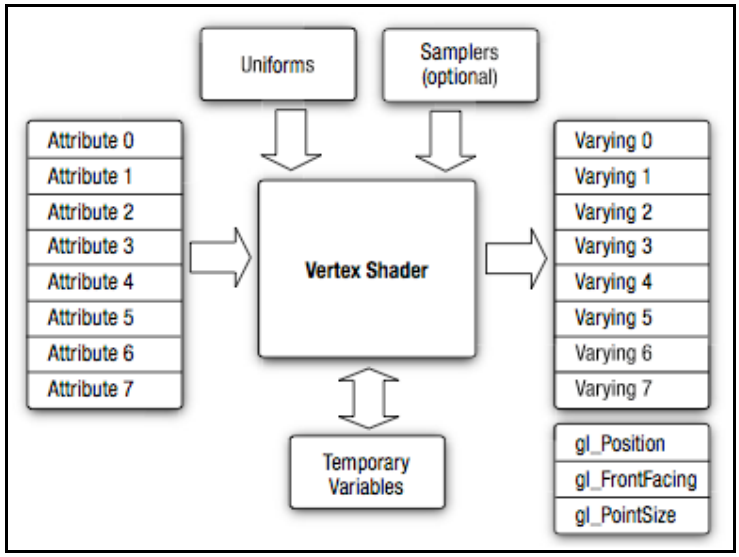
```
attribute vec4 Position;
attribute vec4 SourceColor;
varying vec4 DestinationColor;
uniform mat4 Projection;
uniform mat4 Modelview;

void main(void)
{
    DestinationColor = SourceColor;
    gl_Position = Projection * Modelview * Position;
}
```

Fonte: Rideout (2010, p. 35).

Quadro 1 – Código de um `vertex shader`

Os `attributes` são variáveis individuais, cada vértice possui valores específicos, já as `uniforms` possuem dados comuns à todos os vértices (MUNSHI; GINSBURG; SHREINER, 2009, p. 4). Tanto os atributos (`attribute`) quanto as constantes (`uniform`) são preenchidos através da função `glVertexAttribPointer` da OpenGL ES.

Os `varyings` representam as saídas do `vertex shader`. Nos estágios primitivo e de rasterização (*Primitive Assembly* e *Rasterizer* da Figura 1), os valores das saídas são processados para cada intervalo entre os vértices e enviados como entrada para o `fragment shader` (MUNSHI; GINSBURG; SHREINER, 2009, p. 5). As entradas e saídas do `vertex shader` são representadas na Figura 4.



Fonte: Munshi, Ginsburg e Shreiner (2009, p. 5).

Figura 4 – OpenGL ES 2.0 `vertex shader`

`Attributes` são os dados fornecidos para cada vértice a ser desenhado na cena, enquanto os `uniforms` são dados usados pelo `vertex shader` para todos os vértices. O recurso `Samplers` é um tipo específico de `uniforms` que representa as texturas usadas na cena gráfica, sua utilização é opcional.

2.4.2 Fragment Shader

O `fragment shader` é utilizado na produção de efeitos em texturas, iluminação por pixel e sombras (MUNSHI; GINSBURG; SHREINER, 2009, p. 181). Ele é processado para todos os *pixels* gerados através da interpolação dos vértices do `vertex shader`, o processamento ocorre pixel à pixel (KHRONOS GROUP, 2011e).

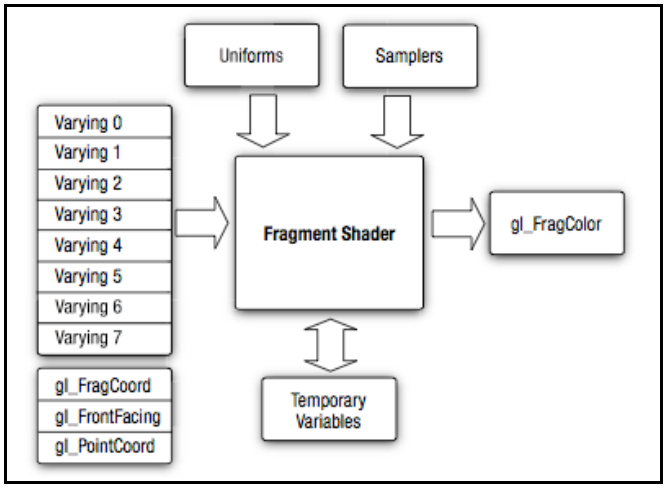
O Quadro 2 demonstra um `fragment shader` utilizado para renderizar os *pixels* com a cor interpolada entre as cores dos vértices do `vertex shader`.

```
varying lowp vec4 DestinationColor;

void main(void)
{
    gl_FragColor = DestinationColor;
}
```

Fonte: Rideout (2010, p. 36).

Quadro 2 – Código de um `fragment shader`

A Figura 5 representa as entradas e saídas do `fragment shader`.



Fonte: Munshi, Ginsburg e Shreiner (2009, p. 8).

Figura 5 – OpenGL ES 2.0 `fragment shader`

As `varyings` são os dados de entrada recebidos pelo `fragment shader`. Estes dados saem do `vertex shader` e são processados pelo estágio de rasterização para cada pixel intermediário dos vértices utilizando interpolação. Os `uniforms`, assim como acontece no `vertex shader` são dados constantes utilizados para todos os pixels, já o `samplers` é um tipo especial de `uniforms` usado para representar as texturas utilizadas na renderização.

Para utilização da biblioteca OpenGL ES 2.0 e a GLSL, pode-se escolher entre diferentes linguagens de programação, como por exemplo C, C++, Java e Objective-C.

2.5 TRABALHOS CORRELATOS

A seguir são apresentadas duas ferramentas correlatas ao motor desenvolvido. Ambos são motores de jogos 3D desenvolvidos para iOS. O MJ3I (TAKANO, 2009) é utilizado como base de extensão para o motor proposto. Será apresentado também o Unity 3 (UNITY, 2010a), conhecido motor de jogos para a plataforma iOS.

2.5.1 Motor de jogos 3D para o iPhone OS (TAKANO, 2009)

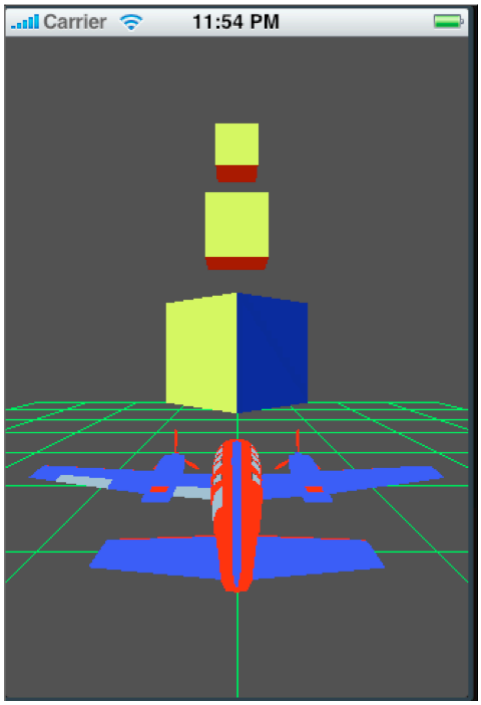
Takano (2009) projetou e implementou um motor de jogos para o iPhone OS, o MJ3I. O motor possui um grafo de cena para controle e gerenciamento da cena, construindo uma estrutura em forma de árvore, permitindo que cada objeto possua *n* filhos e a árvore possua *h*

níveis de hierarquia. Como resultado é possível gerenciar cenas com vários objetos geométricos e controlar as suas respectivas transformações (TAKANO, 2009, p. 46).

O motor implementado também realiza tratamento de toques na tela, controle de câmera, detecção de colisão, importação de modelos 3D e tratamento de eventos (TAKANO, 2009).

O motor foi todo programado em Objective-C, utilizando a ferramenta de desenvolvimento Xcode, que possui suporte oficial da Apple. Conforme relatado no trabalho, um dos pontos que facilitou o desenvolvimento foi a presença de um simulador integrado ao ambiente de desenvolvimento, permitindo a realização de testes com notável agilidade (TAKANO, 2009, p. 48).

A Figura 6 mostra o motor desenvolvido em funcionamento.



Fonte: Takano (2009).
Figura 6 – MJ3I em funcionamento

2.5.2 Unity 3

Unity 3 é uma ferramenta comercial voltada para o desenvolvimento de jogos 3D. Esta ferramenta permite trabalhar com DirectX ou OpenGL e suporta importação de modelos 3D, animações, texturas, *scripts* e sons criados em outros programas (UNITY, 2010a).

A *engine* destinada para os cálculos de física utilizada é a NVIDIA PhysX (UNITY, 2010d), considerada uma das *engines* mais poderosas atualmente. Ela permite a simulação do

movimento de peças de roupa ou de partes de um humanoide, como cabelo por exemplo. A NVIDIA PhysX permite também representar a deformação de uma bola ao ser chutada, articulações em objetos e sensação de tração em pneus, ideal para jogos de corrida (UNITY, 2010a).

Unity 3 suporta três linguagens de *script*: JavaScript, C# e um dialeto de Python chamado Boo. As 3 linguagens podem utilizar bibliotecas .NET, suportando interação com banco de dados, expressões regulares, XML e serviços de rede (UNITY, 2010b).

Multiplataforma, a ferramenta possui capacidade de publicar um projeto para Web, iOS, Android, Windows, Mac OS, Nintendo Wii, Playstation e Xbox (UNITY, 2010c).

A figura 7 apresenta a interface de edição de cenas da ferramenta Unity 3.

Fonte: Unity 3 (2010e).

Figura 7 – Tela de edição do Unity 3

3 DESENVOLVIMENTO

Neste capítulo são descritas as etapas realizadas para a construção do motor gráfico. São ilustradas as principais características da ferramenta, sua especificação, os requisitos funcionais e não-funcionais, a implementação e os resultados obtidos.

3.1 DESENVOLVIMENTO IOS

A aplicação proposta foi desenvolvida em 2 etapas. Uma das etapas consistiu em adaptar o código fonte do V-ART para que fosse compatível com o OpenGL ES 2.0 e para que fosse possível a utilização de bibliotecas e recursos da linguagem Objective-C. Na segunda etapa foi desenvolvida a aplicação para demonstração do motor gráfico implementado.

Ambas as etapas foram desenvolvidas utilizando o ambiente de desenvolvimento padrão do iOS, o Xcode.

3.2 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

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

a) a aplicação deverá gerenciar os objetos criados de forma hierárquica (RF);
b) a aplicação deverá implementar regras para movimentação de câmeras no ambiente (RF);
c) a aplicação deverá possuir um ambiente de visualização 3D, onde será possível constatar o funcionamento do motor proposto (RNF);
d) a aplicação deverá ser implementado na linguagem de programação Objective-C (RNF);
e) a aplicação deverá ser implementado com o auxílio das bibliotecas disponíveis no iOS SDK (RNF);

f) a aplicação deverá ser desenvolvido no ambiente Xcode 4 (RNF);
g) a aplicação deverá utilizar a biblioteca OpenGL ES para renderização dos modelos 3D (RNF);
h) a aplicação deverá ser documentado utilizando a ferramenta DoxyGen (DOXYGEN, 2012) (RNF);
i) a aplicação deverá ser testado no simulador integrado ao Xcode e também em um dispositivo iOS físico (iPhone, iPod Touch ou iPad) (RNF).

3.3 ESPECIFICAÇÃO

A especificação da ferramenta foi desenvolvida através da ferramenta Omnigraffle, utilizando os conceitos de orientação a objetos e baseando-se nos diagramas da *Unified Modeling Language* (UML), gerando os diagramas de caso de uso, classes e de sequência apresentados a seguir.

3.3.1 Casos de uso

Nesta seção são descritos os casos de uso de recursos da ferramenta. Os casos de uso são divididos entre dois atores. O Usuário faz uso da aplicação e interage com a mesma através de toques na tela. Já o Desenvolvedor faz uso das ferramentas disponibilizadas para configuração da cena a ser representada. Na figura 8 é apresentado o diagrama de casos de uso resultante.

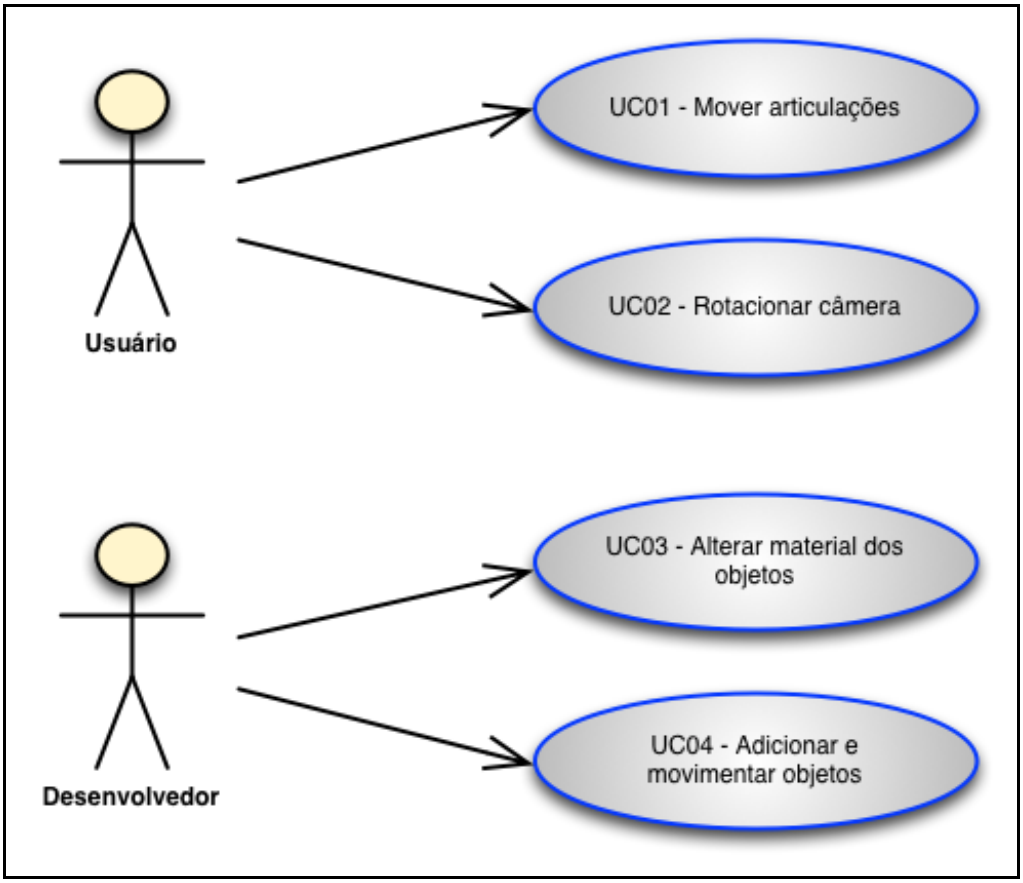


Figura 8 - Diagrama de casos de uso

3.3.1.1 Mover articulações

Este caso de uso descreve a interação do ator `Usuário` com o aplicativo, afim de mover as articulações do personagem representado. Detalhes deste caso de uso estão descritos no Quadro 3.

UC01 - Mover articulações	
Descrição	As articulações possuem um valor mínimo e um máximo que limitam o movimento da articulação.
Pré-Condição	O aplicativo de personagem articulado deve estar executando.
Cenário Principal	1. O `Usuário` toca o dispositivo na metade inferior da tela; 2. A aplicação move a articulação de acordo com a posição do toque na tela.
Exceção 1	1. Se a articulação estiver em seu limite mínimo ou máximo e o `Usuário` pressionar a tela na posição equivalente à mesma direção, nada acontece.
Pós-Condição	A articulação equivalente deve estar movida.

Quadro 3 - Caso de uso `UC01`

3.3.1.2 Rotacionar câmera

Este caso de uso descreve como o ator `Usuário` interage com o aplicativo para girar a câmera ao redor do personagem. Detalhes deste caso de uso estão descritos no Quadro .

UC02 – Rotacionar câmera	
Descrição	A câmera possui a funcionalidade de rotacionar ao redor do personagem.
Pré-Condição	O aplicativo de personagem articulado deve estar executando.
Cenário Principal	1. O `Usuário` arrasta o dedo sobre a tela horizontalmente; 2. A aplicação rotaciona a câmera ao redor do personagem de acordo com a direção que o `Usuário` arrastou.
Pós-Condição	A câmera dever estar rotacionada na direção que o `Usuário` arrastou o dedo.

Quadro 4 - Caso de uso `UC02`

3.3.1.3 Alterar material dos objetos

Este caso de uso descreve como o ator `Desenvolvedor` interage com o aplicativo para alterar o material dos objetos renderizados na cena. Detalhes deste caso de uso estão descritos no Quadro 5.

UC03 – Alterar material dos objetos	
Descrição	Cada objeto da cena possui um tipo de material atrelado, que pode ser alterado.
Pré-Condição	O `Desenvolvedor` deve possuir uma aplicação que utiliza a biblioteca de personagens articulados.
Cenário Principal	1. O `Desenvolvedor` instancia um novo objeto da classe `Material` passando como parâmetro o tipo de material a ser criado; 2. A biblioteca cria o novo objeto de acordo com o parâmetro informado; 3. O `Desenvolvedor` executa o método `SetMaterial` de um dos objetos da cena passando como parâmetro o objeto instanciado no passo 1.
Pós-Condição	O dispositivo deve renderizar o objeto com o novo tipo de material que foi atrelado.

Quadro 5 - Caso de uso `UC03`

3.3.1.4 Adicionar e movimentar objetos

Este caso de uso descreve a interação do ator `Desenvolvedor` com o aplicativo, afim

de adicionar e movimentar objetos e articulações. Detalhes deste caso de uso estão descritos no Quadro 6.

UC04 – Adicionar objetos à cena	
Descrição	Vários objetos e articulações podem ser adicionados a uma cena. As articulações são adicionadas entre objetos e fazem com que os objetos filhos do objeto base sejam movidos ao movimentar a articulação.
Pré-Condição	O `Desenvolvedor` deve possuir uma aplicação que utiliza a biblioteca de personagens articulados.
Cenário Principal	1. O `Desenvolvedor` instancia um novo objeto da classe `MeshObject` e executa o método `makeBox` informando as coordenadas de largura, altura e profundidade mínimas e máximas; 2. A biblioteca cria o novo objeto de acordo com as coordenadas informadas; 3. O `Desenvolvedor` instancia um novo objeto da classe `UniaxialJoint` e executa o método `addDof` armazenando o retorno deste método em um atributo do tipo `Dof`; 4. A biblioteca instancia uma nova articulação e adiciona a ela um grau de liberdade; 5. O `Desenvolvedor` executa o método `addChild` do objeto `MehsObject` criado no passo 1 passando como parâmetro a articulação criada no passo 3; 6. A biblioteca insere a articulação do passo 3 na lista de filhos do objeto do passo 1; 7. O `Desenvolvedor` repete os passos 1 e 2; 8. O `Desenvolvedor` executa o método `addChild` da articulação criada no passo 3 passando como parâmetro o objeto criado no passo 7; 9. A biblioteca insere o objeto criado no passo 7 na lista de objetos da articulação criada no passo 3; 10. O Desenvolvedor adapta o método de reconhecimento de toques na tela para executar o método move do objeto `Dof` criado no passo 3.
Pós-Condição	O dispositivo deve exibir dois objetos retangulares tridimensionais e habilitar o reconhecimento de toques na tela para movimentação da articulação criada entre esses objetos.

Quadro 6 - Caso de uso `UC04`

3.3.2 Diagramas de classes

Nesta seção são descritas as classes necessárias para a criação da aplicação que demonstra o motor gráfico em funcionamento. Na seção 3.3.2.1 serão exibidas as classes que compõem o aplicativo desenvolvido.

3.3.2.1 Aplicação

O diagrama da figura 9 representa as principais classes utilizadas pela aplicação que demonstra o funcionamento do motor gráfico desenvolvido.

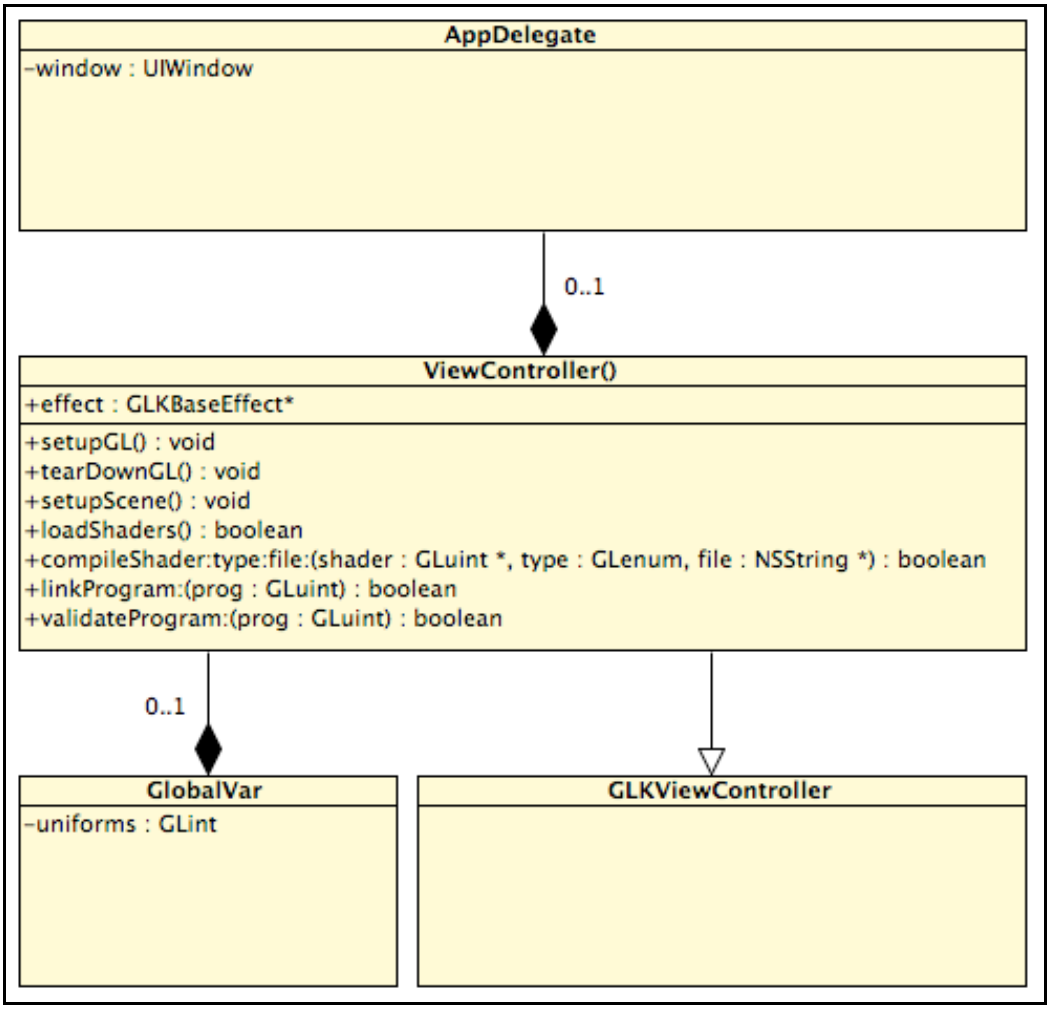


Figura 9 - Principais classes da biblioteca V-ART

A criação da aplicação foi feita com o *template* padrão do Xcode para desenvolvimento de aplicativos OpenGL ES. Através desse *template* o Xcode abastece a aplicação com 2 classes, `AppDelegate` e `ViewController`. A classe `AppDelegate` é responsável pela comunicação com a aplicação através de eventos disparados sempre que determinado estado é atingido, como por exemplo quando um aplicativo termina de carregar ou quando está prestes a ser encerrado. Já a classe `ViewController` fornece um modelo para controle de uma ou mais *Views,* responsáveis pela exibição na tela do dispositivo.

A classe `ViewController` utilizada na aplicação é um objeto da classe `GLKViewController` que implementa o *loop* de renderização do OpenGL ES. A classe `GlobalVar` foi criada para instanciação de variáveis globais. Essas variáveis são utilizadas

tanto nas classes da aplicação (classes Objective-c) como nas classes da biblioteca V-ART (classes c++).

3.3.3 Diagrama de sequencia

O diagrama de sequencia da figura 10 demonstra a interação do `Usuário` com a aplicação de personagens articulados, exemplificando o caso de uso `UC01`.

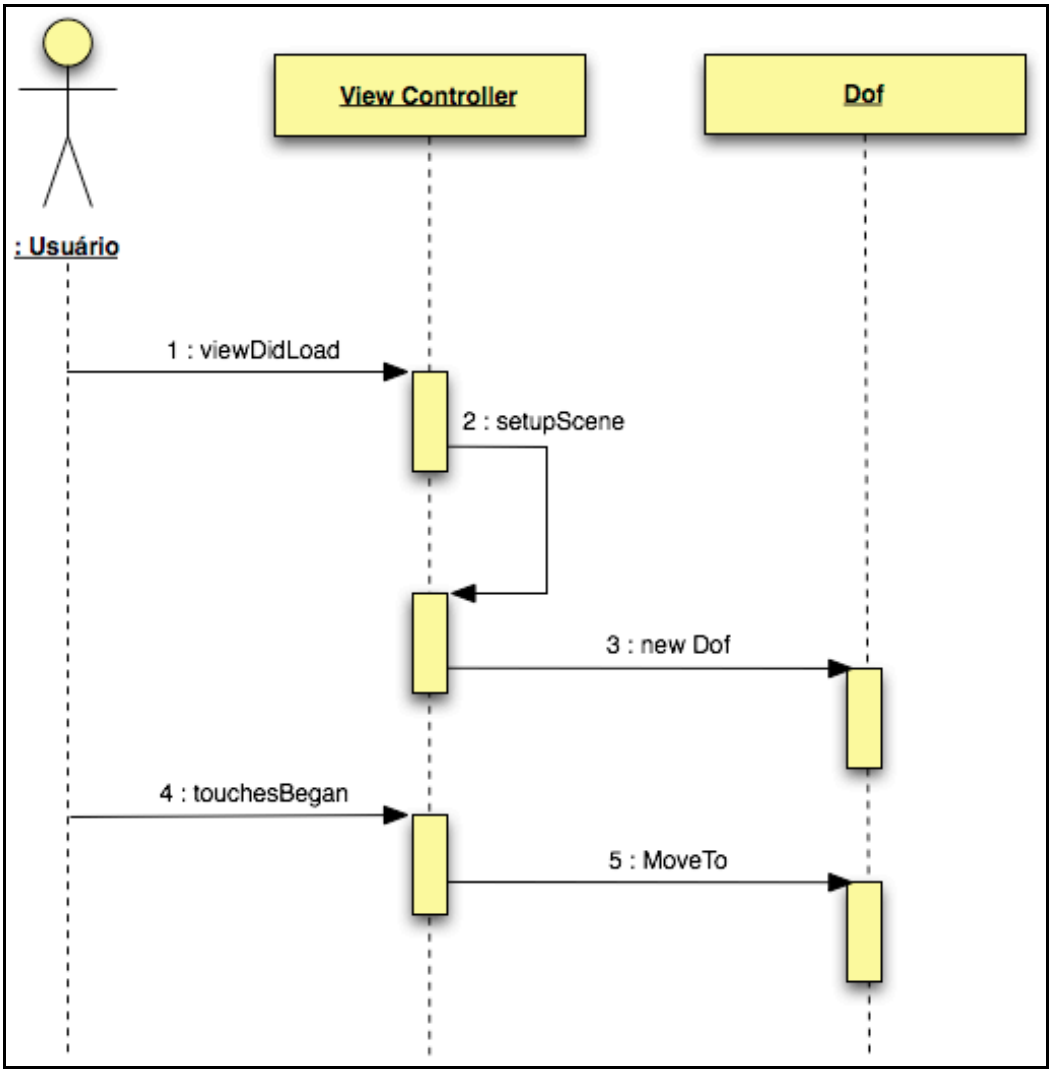


Figura 10 – Diagrama de sequencia `Mover articulações`

Ao acessar o aplicativo, o Usuário dispara o método `viewDidLoad` da classe `ViewController`, inicializando o conteúdo da aplicação e instanciando os objetos que compõem a cena através da chamada do método `setupScene`.

Após a inicialização da aplicação e de seus objetos, o usuário pode tocar na tela do dispositivo, disparando o método `touchesBegan`. Se o toque foi executado na metade inferior

da tela, a articulação será rotacionada através da chamada do método `MoveTo`.

3.4 IMPLEMENTAÇÃO

A seguir são mostradas as técnicas e ferramentas utilizadas e a operacionalidade da implementação, bem como as principais classes e métodos.

3.4.1 Técnicas e ferramentas utilizadas

O desenvolvimento do motor gráfico foi feito na ferramenta de desenvolvimento padrão do iOS, o Xcode, atualmente na versão 4.5.2. As linguagens de programação utilizadas foram Objective-C e C++. Na linguagem Objective-C foram desenvolvidas as classes responsáveis pela criação do ambiente OpenGL ES, pelo tratamento dos toques no dispositivo e criação da cena e de seus elementos. Essas classes foram desenvolvidas com o auxílio dos *frameworks* `Foundation`, `UIKit`, `CoreGraphics`, `QuartzCore` e `OpenGLES`. A versão da biblioteca OpenGL ES utilizada é a 2.0.

As classes da biblioteca V-ART foram mantidas na linguagem C++, sendo necessárias adaptações somente nos métodos responsáveis pelo desenho da cena para utilização da biblioteca OpenGL ES 2.0. Para realização de testes na aplicação, foram utilizados um simulador integrado ao Xcode, além de um dispositivo físico iPhone 4S, atualizado com a sexta versão do iOS.

O iPhone 4S possui processador Cortex-A9 Dual-core com 1 *Giga Hertz* (GHz), GPU PowerVR SGX543MP2, com memória de 512 *Mega Bytes* (MB) de RAM e display de 3.5 polegadas com resolução de 640x960 (GSM ARENA, 2012).

3.4.1.1 Interface da aplicação

O Xcode possui um editor visual de representações de interfaces de usuário, chamadas *storyboards*. Uma *storyboard* é composta por uma sequencia de cenas, cada uma contendo um

`view controller` e sua respectiva `view`.

Apenas uma `view controller` foi utilizada na aplicação desenvolvida, a classe ViewController, que é a tela principal da aplicação.

3.4.1.2 Composição da cena

Ao inicializar o aplicativo, a cena `View Controller` é carregada, instanciando um objeto da classe `ViewController`, classe responsável por controlar a `view`, e executando o método `viewDidLoad`, responsável pela inicialização do ambiente OpenGL ES e dos objetos que compõem a cena, conforme demonstra o Quadro 7.

```
102 - (void)viewDidLoad
103 {
104     [super viewDidLoad];
105
106     self.context = [[EAGLContext alloc] initWithAPI:kEAGLRenderingAPIOpenGLES2];
107
108     if (!self.context) {
109         NSLog(@"Failed to create ES context");
110     }
111
112     GLKView *view = (GLKView *)self.view;
113     view.context = self.context;
114     view.drawableDepthFormat = GLKViewDrawableDepthFormat24;
115
116     [self setupGL];
117     [self setupScene];
118     _lastRotation = 0.0f;
119 }
```

Quadro 7 – Método `viewDidLoad` da classe `ViewController`

Durante a execução do método viewDidLoad, o contexto da aplicação é inicializado com a versão 2.0 do OpenGL ES. A `view` é criada utilizando como base a classe `GLKView`, que implementa o loop de renderização do OpenGL.

Após a configuração do ambiente OpenGL ES, o método `viewDidLoad` executa os métodos `setupGL` e `setupScene`, responsáveis por carregar os `Shaders` utilizados na aplicação e instanciar os objetos que compõem a cena a ser exibida, respectivamente.

O método `setupGL` instancia o atributo `effect`, que mantém matrizes de visualização e projeção do modelo e executa o método `loadShaders`, como demonstrado no Quadro 8.

```
148  - (void)setupGL
149  {
150      [EAGLContext setCurrentContext:self.context];
151
152      [self loadShaders];
153
154      self.effect = [[GLKBaseEffect alloc] init];
155
156      glEnable(GL_DEPTH_TEST);
157  }
```

Quadro 8 – Método `setupGL` da classe `ViewController`

A principal funcionalidade da classe `setupGL` é a execução do método `loadShaders`, que carregará o `Fragment` e o `Vertex Shader` para utilização na aplicação, nos Quadros 9, 10 e 11 é possível visualizar mais detalhes sobre estes métodos além da implementação dos `Shaders`.

```
270  - (BOOL)loadShaders
271  {
272      GLuint vertShader, fragShader;
273      NSString *vertShaderPathname, *fragShaderPathname;
274
275      // Create shader program.
276      _program = glCreateProgram();
277
278      // Create and compile vertex shader.
279      vertShaderPathname = [[NSBundle mainBundle] pathForResource:@"Shader" ofType:@"vsh"];
280      if (![self compileShader:&vertShader type:GL_VERTEX_SHADER file:vertShaderPathname])
             {...}
284
285      // Create and compile fragment shader.
286      fragShaderPathname = [[NSBundle mainBundle] pathForResource:@"Shader" ofType:@"fsh"];
287      if (![self compileShader:&fragShader type:GL_FRAGMENT_SHADER file:fragShaderPathname])
             {...}
291
292      // Attach vertex shader to program.
293      glAttachShader(_program, vertShader);
294
295      // Attach fragment shader to program.
296      glAttachShader(_program, fragShader);
297
298      // Bind attribute locations.
299      // This needs to be done prior to linking.
300      glBindAttribLocation(_program, GLKVertexAttribPosition, "position");
301      glBindAttribLocation(_program, GLKVertexAttribNormal, "normal");
302
303      // Link program.
304      if (![self linkProgram:_program]) {...}
322
323      // Get uniform locations.
324      uniforms[UNIFORM_MODELVIEWPROJECTION_MATRIX] = glGetUniformLocation(_program,
             "modelViewProjectionMatrix");
325
326      uniforms[UNIFORM_COLOR_MATRIX] = glGetUniformLocation(_program, "colorMatrix");
327
328      // Release vertex and fragment shaders.
329      if (vertShader) {...}
333
334      if (fragShader) {...}
338
339      return YES;
340  }
```

Quadro 9 – Método `loadShaders` da classe `ViewController`

```
varying lowp vec4 colorVarying;

void main()
{
    gl_FragColor = colorVarying;
}

```

Quadro 10 – Implementação `Fragment Shader`

```
attribute vec4 position;

varying lowp vec4 colorVarying;

uniform mat4 modelViewProjectionMatrix;
uniform mat2 colorMatrix;

void main()
{
    colorVarying = vec4(colorMatrix[0][0], colorMatrix[0][1], colorMatrix[1][0], colorMatrix[1][1]);

    gl_Position = modelViewProjectionMatrix * position;
}
```

Quadro 11 – Implementação `Vertex Shader`

O método `loadShaders` é responsável por efetivar a criação de um `Shader Program`, que será ligado aos `Shaders` da aplicação. Os `Shaders` são criados, compilados através do método `compileShader`, e caso não ocorra nenhum erro na compilação, os mesmos são anexados ao programa criado.

Além disso, o método vincula a posição dos atributos do `Vertex Shader` e grava a localização das matrizes uniformes do `Shader`.

Após a execução do método `setupGL`, é executado o método `setupScene`, responsável por inicializar a cena e criar os objetos que a compõem. No Quadro 12, podem ser observados os objetos utilizados na aplicação, declarados como atributos da classe `ViewController`.

```
@interface ViewController () {

    /...

    VART::MeshObject base;
    VART::Material mat;
    VART::Dof* dofPtr1;
    VART::Dof* dofPtr2;
    VART::Dof* dofPtr3;
    VART::MeshObject arm1;
    VART::MeshObject arm2;
    VART::MeshObject arm3;
    VART::UniaxialJoint baseJoint;
    VART::UniaxialJoint joint12;
    VART::UniaxialJoint joint23;
}
```

Quadro 12 – Objetos utilizados na aplicação

Os objetos declarados na classe `ViewController` compõem a cena a ser renderizada.

O objeto `base` representa a base onde serão inseridos os braços mecânicos, enquanto o objeto `mat` representa o material do qual são feitos os objetos. Os objetos do tipo `Dof` representam o grau de liberdade de cada articulação, que são representadas pelos objetos das classes `UniaxialJoint`, `BiaxialJoint` e `PolyaxialJoint`. Os objetos `arm1`, `arm2` e `arm3` representam os 3 braços da aplicação.

No Quadro 13, é possível visualizar a execução do método `setupScene` que configura a cena e seus objetos.

```
-(void) setupScene {
    dofPtr1 = baseJoint.AddDof(Point4D::Y(),Point4D::ORIGIN(), -3.141592654, 3.141592654)
        ;
    base.AddChild(baseJoint);

    base.MakeBox(-1,1,-0.1,0.1,-1,1);
    mat = VART::Material::DARK_PLASTIC_GRAY();
    base.SetMaterial(mat);

    //    base -> arm1
    arm1.MakeBox(-0.1,0.1, 0,0.5, -0.1,0.1);
    arm1.SetMaterial(VART::Material::PLASTIC_GREEN());
    baseJoint.AddChild(arm1);

    dofPtr2 = joint12.AddDof(Point4D::Z(), Point4D(0,0.5,0), -1.570796327, 1.570796327);
    arm1.AddChild(joint12);

    //    joint12 -> arm2
    arm2.MakeBox(-0.1,0.1, 0.5,1, -0.1,0.1);
    arm2.SetMaterial(VART::Material::PLASTIC_GREEN());
    joint12.AddChild(arm2);

    dofPtr3 = joint23.AddDof(Point4D::Z(), Point4D(0,1,0), -1.570796327, 1.570796327);
    arm2.AddChild(joint23);

    //    joint23 -> arm3
    arm3.MakeBox(-0.1,0.1, 1,1.5, -0.1,0.1);
    arm3.SetMaterial(VART::Material::PLASTIC_GREEN());
    joint23.AddChild(arm3);
}
```

Quadro 13 – Método `setupScene` da classe `ViewController`

O método `AddDof` é responsável pela criação dos graus de liberdade das articulações, recebendo como parâmetro pontos que definem sobre qual eixo a articulação irá rotacionar, a posição do `Dof` em relação ao objeto pai e os ângulos mínimo e máximo de rotação.

Para composição da estrutura hierárquica da cena, é utilizado o método `AddChild`, que adiciona um objeto na lista de filhos de um objeto pai. As transformações sofridas pelo objeto pai refletem também na renderização de todos os seus filhos.

Os métodos `MakeBox` e `SetMaterial` são responsáveis por facilitar a criação de objetos tridimensionais retangulares e por definir o material utilizado na criação do objeto, respectivamente.

Após a inicialização da cena, o método `glkViewdrawInRect` é executado, dando início ao fluxo de renderização da cena, conforme demonstrado no Quadro 14.

```
506 - (void)glkView:(GLKView *)view drawInRect:(CGRect)rect {
507
508     [self setupMatrix];
509
510     glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
511     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
512
513     // Render the object again with ES2
514     glUseProgram(_program);
515
516     glUniformMatrix4fv(uniforms[UNIFORM_MODELVIEWPROJECTION_MATRIX], 1, 0,
        _modelViewProjectionMatrix.m);
517     glUniformMatrix3fv(uniforms[UNIFORM_NORMAL_MATRIX], 1, 0, _normalMatrix.m);
518     glUniformMatrix2fv(uniforms[UNIFORM_COLOR_MATRIX], 1, 0, _color.m);
519
520     base.DrawOGL(model);
521 }
```

Quadro 14 – Método `glkViewdrawInRect` da classe `ViewController`

O método `glkViewdrawInRect` é executado dentro do loop de renderização do OpenGL ES, criado automaticamente pela classe `GLKView`. O método `setupMatrix` é responsável por configurar as matrizes de projeção e modelagem iniciais da cena.

O método também é responsável por definir a cor de fundo da aplicação e indicar o `Shader Program` que será utilizado. As matrizes configuradas no método `setupMatrix` são enviadas para os `uniforms` do `Vertex Shader` e em seguida é executado o método responsável por iniciar a renderização dos objetos, `DrawOGL`.

No Quadro 15 é possível visualizar o método `DrawInstanceOGL`, chamado durante a execução do método `DrawOGL` e responsável por renderizar um `MeshObject`.

```
670 bool VART::MeshObject::DrawInstanceOGL() const {
671 // Note that vertex arrays must be enabled to allow drawing of optimized meshes. See
672 // VART::ViewerGlutOGL.
673 #ifdef VART_OGL
674     /*...*/
747 #else
748     #ifdef VART_OGL_IOS
749         bool result = true;
750 
751         if (show) // if visible...
752         {         // FixMe: no need to keep this old name; rename "show" to "visible".
753             if (vertCoordVec.size() > 0)
754             { // Optimized structure found - draw it!
755                 GLfloat gMeshVertexData[vertCoordVec.size()];
756 
757                 for (int i = 0; i < vertCoordVec.size(); i++) {
758                     gMeshVertexData[i] = static_cast<GLfloat>(vertCoordVec[i]);
759                 }
760 
761                 GLuint _vertexArray;
762                 GLuint _vertexBuffer;
763 
764                 glGenVertexArraysOES(1, &_vertexArray);
765                 glBindVertexArrayOES(_vertexArray);
766 
767                 glGenBuffers(1, &_vertexBuffer);
768                 glBindBuffer(GL_ARRAY_BUFFER, _vertexBuffer);
769                 glBufferData(GL_ARRAY_BUFFER, sizeof(gMeshVertexData), gMeshVertexData,
                        GL_STATIC_DRAW);
770 
771                 glEnableVertexAttribArray(0);
772                 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
773 
774                 glDrawArrays(GL_LINES, 0, sizeof(gMeshVertexData));
775                 glBindVertexArrayOES(0);
776             }
777         }
778 
779         bBox.DrawInstanceOGL();
780         recBBox.DrawInstanceOGL();
781         return result;
782     #else
783         return false;
784     #endif
785 #endif
786 
787     return false;
788 }
```

Quadro 15 – Método `DrawInstanceOGL` da classe `MeshObject`

O método `DrawInstanceOGL` é responsável por carregar em um *array* os vértices que compõem o objeto tridimensional. O *array* contendo os vértices é então vinculado ao *buffer* de renderização da aplicação. Através do método `glDrawArrays` os vértices são renderizados, formando o objeto desejado.

A execução da aplicação no simulador integrado do Xcode é demonstrada na Figura 11.

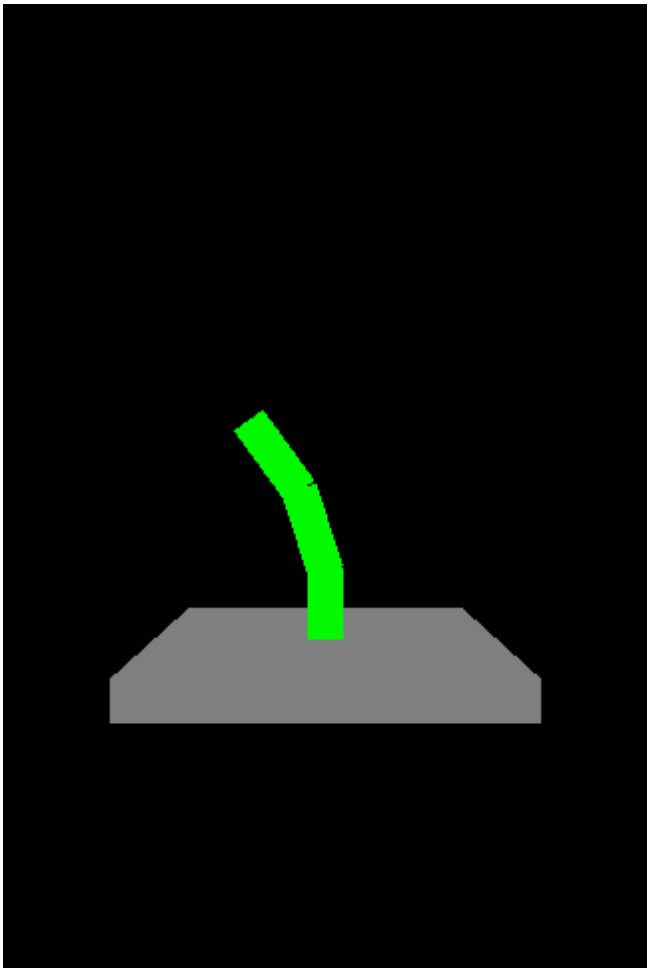


Figura 11 – Aplicação executando no simulador

3.4.2 Operacionalidade da implementação

Esta seção tem como objetivo demonstrar a operacionalidade da implementação em nível de usuário, através do estudo dos casos de uso `UC01` e `UC02`.

O caso de uso `UC01` descreve a interação do usuário com a aplicação com o objetivo de mover as articulações do personagem representado, enquanto o caso de uso `UC02` descreve o movimento de rotação que pode ser realizado na visualização da aplicação.

Para reconhecimento dos toques na tela, a mesma foi dividida em 3 segmentos, conforme visualizado na Figura 12.

Figura 12 – Tela do dispositivo dividida em segmentos

O segmento 3 da tela é utilizado para escolha da articulação que será movimentada. Cada toque realizado nesse segmento alterna a seleção entre as articulações da aplicação. Os segmentos 1 e 2 são responsáveis pelo reconhecimento de toques que irão movimentar efetivamente a articulação. No segmento 2 a movimentação é realizada no sentido horário, enquanto o segmento 1 movimenta no sentido anti-horário.

Em qualquer um dos segmentos é possível tocar o dispositivo e arrastar o dedo sobre o mesmo, dessa forma a aplicação realiza a rotação da câmera utilizada na visualização do personagem.

Na Figura 13 é possível visualizar uma sequencia de movimentações dos objetos, após

a aplicação do caso de uso `UC01`.

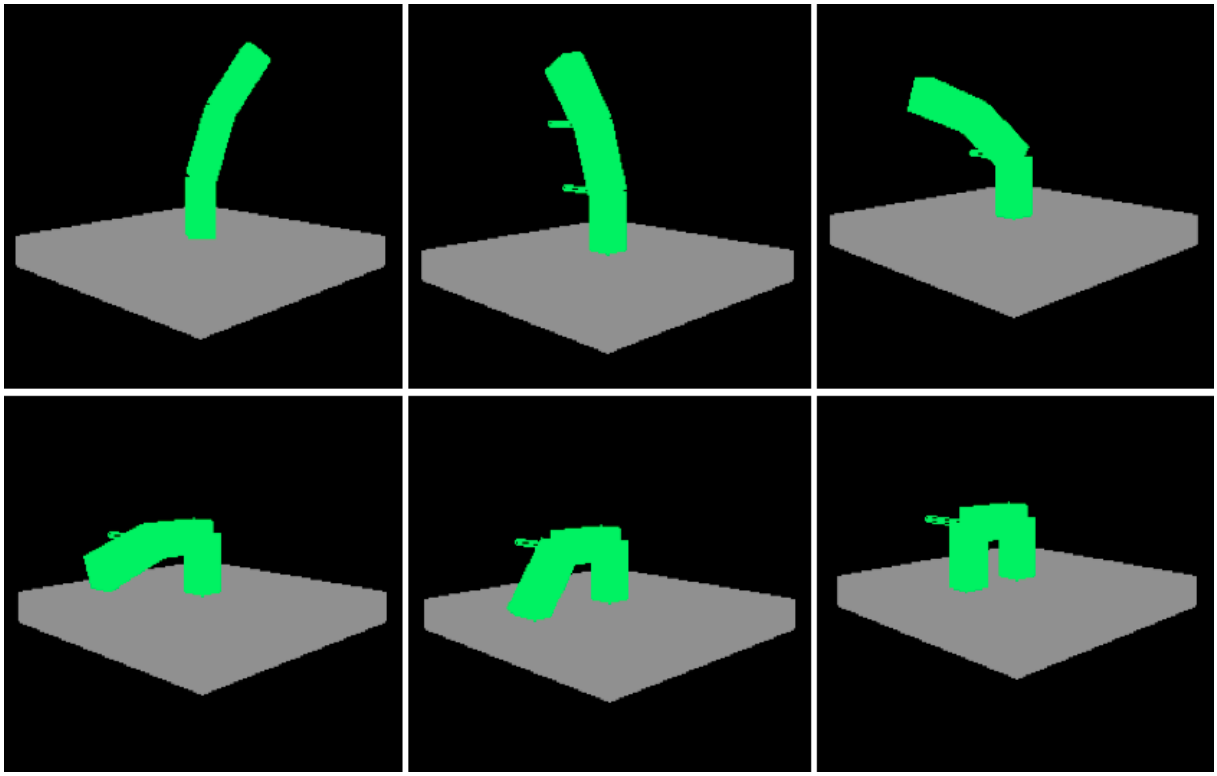


Figura 13 – Objetos após aplicação do caso de uso `UC01`

Na Figura 14 é demonstrada a aplicação após rotacionar a câmera 45 graus, através da aplicação do caso de uso `UC02`.

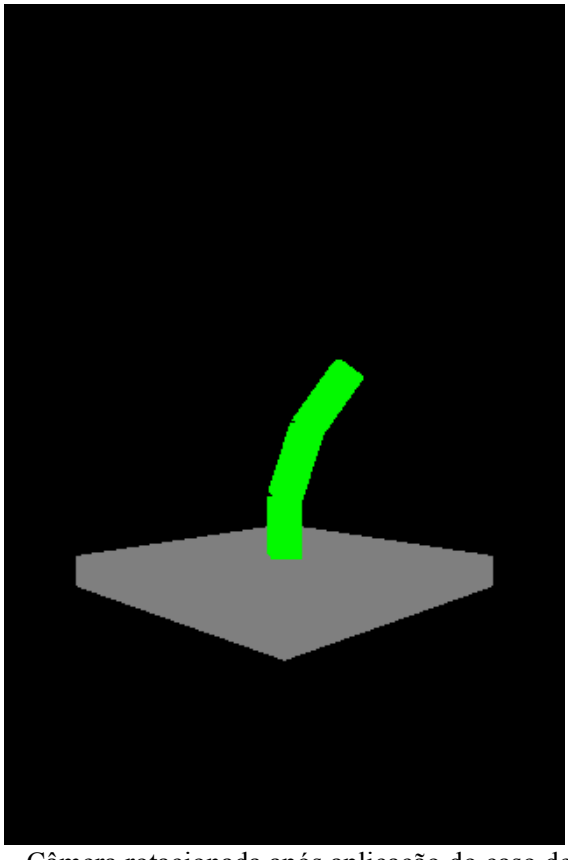


Figura 14 – Câmera rotacionada após aplicação do caso de uso `UC02`

3.5 RESULTADOS E DISCUSSÃO

O trabalho teve como principal objetivo utilizar a biblioteca Virtual Articulations for Virtual Reality (V-ART) para renderização de personagens articulados no sistema operacional iOS, portando para tal uma aplicação já funcional no sistema Mac OS, e disponibilizar um ambiente para visualização desses personagens. Além desses foi definido também como objetivo o processamento de arquivos XML para sintetização de personagens, objetivo esse não concluído no presente trabalho.

A primeira dúvida ao iniciar a implementação foi se a biblioteca deveria ou não ser convertida de C++ para Objective-C, linguagem nativa do iOS. Convertendo a biblioteca para Objective-C, ganha-se na padronização do código fonte, além de facilitar a comunicação entre as classes da biblioteca com as classes da aplicação. Por outro lado, perde-se em portabilidade, uma vez que mantendo a biblioteca em C++, o código fonte já funcional no Mac OS foi incrementado, mas permanece funcional nesse ambiente. Dessa forma optou-se por não converter a biblioteca para a linguagem nativa do iOS.

A grande dificuldade encontrada no desenvolvimento do trabalho foi as diferenças entre as versões da biblioteca OpenGL. Entre a versão ES 2.0, para dispositivos móveis, e a versão *desktop*, não apenas comandos foram alterados ou removidos, mas todo o conceito utilizado na renderização foi modificado. Comandos básicos como `glBegin` e `glEnd` não são mais utilizados, dando lugar à utilização de `buffers`, `arrays` e `shaders`, combinados com o comando `glDrawArrays`. Optou-se pela utilização da biblioteca em sua versão 2.0 devido à utilização de *shaders*, trechos de código que executam diretamente na GPU do dispositivo, permitindo um melhor gerenciamento dos recursos disponíveis.

3.5.1 Testes de desempenho (FPS) e alocação de memória

A realização dos testes de desempenho e alocação de memória foram executados em duas etapas. A primeira etapa consiste em efetuar os testes com a aplicação desenvolvida para o presente trabalho, enquanto uma segunda etapa analisa uma aplicação com uma quantidade superior de objetos renderizados.

Para medição de desempenho da aplicação foi utilizado o Instruments, aplicação integrada ao Xcode para realização de testes de performance, capaz de efetuar a medição da

taxa de FPS, de alocação e vazamento de memória, entre outras funcionalidades.

3.5.1.1 Testes com aplicação de 3 nodos

Para realização do teste foi medida a taxa de FPS segundo à segundo da aplicação, durante cinco minutos. A taxa de atualização da aplicação se manteve estável durante toda a medição, variando entre 29 e 30 FPS. Foi constatada uma queda na taxa de FPS ao rotacionar de maneira constante a câmera da aplicação, nesse caso caindo para cerca de 26 FPS. A taxa foi limitada em 30 FPS pela classe `GLKView`, limitação essa que é *default* da classe.

Quanto à alocação de memória, a aplicação manteve cerca de 630kb de memória alocados durante os cinco minutos de medição. Houve um pequeno aumento, cerca de 1 à 2kb, constatados ao realizar toques na tela, pois cada toque realizado é salvo em uma coleção, para que seja possível realizar as tratativas de toque que definem a interação do usuário com a aplicação. Uma aplicação sem nenhum nodo renderizado aloca cerca de 593kb de memória.

3.5.1.2 Testes com aplicação de N nodos

Para a segunda etapa da medição de desempenho foi customizada uma aplicação com uma quantidade variável de objetos. O primeiro teste foi realizado em uma aplicação com seis segmentos renderizados e em cada teste subsequente aumenta-se a quantidade de segmentos em três, até o limite estipulado de 30. Na Figura 15 é possível visualizar a aplicação executando no simulador integrado ao Xcode, demonstrando um braço com 30 articulações.

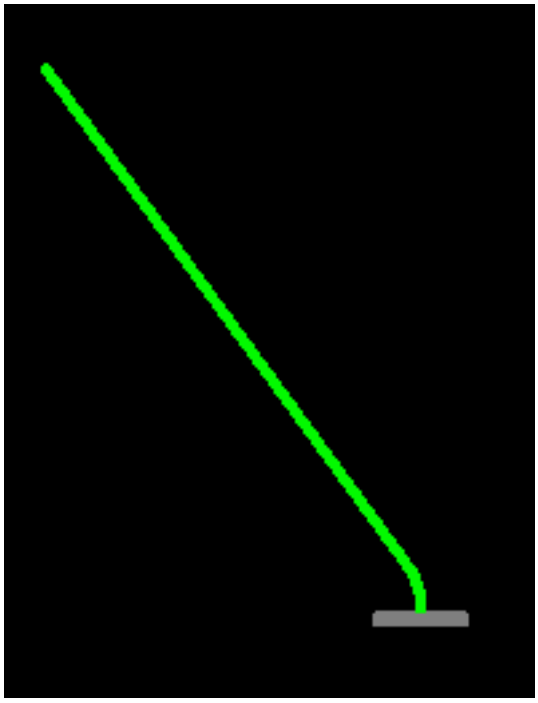


Figura 15 – Aplicação de 30 segmentos executando no simulador

Para realização do teste foram medidas as taxas de FPS segundo à segundo da aplicação, e também a quantidade de memória alocada, durante cinco minutos. A cada nova execução são adicionados 3 segmentos à aplicação.

A Tabela 1 sintetiza as informações obtidas nos testes.

Tabela 1 – Resultados obtidos na aplicação com N nodos

Quantidade de Nodos	Taxa média de FPS	Alocação média de memória (KB)
0	30	593
3	30	630
6	20	640
9	15	647
12	11	656
15	9	666
18	8	676
21	7	687
24	6	695
27	5	708
30	5	720

Ao aumentar a quantidade de objetos desenhados, uma queda bastante considerável na taxa de FPS da aplicação é notada. Além disso uma pequena queda na taxa de FPS é notada ao rotacionar a câmera ou os objetos renderizados, cerca de 2 à 3 FPS.

Algumas melhorias podem ser implementadas para obtenção de um melhor resultado nesse teste, como a eliminação de alguns comandos redundantes, identificados com o auxílio da ferramenta Instruments, além de não haver necessidade de que os objetos sejam desenhados em um loop de renderização da aplicação, redesenhando os objetos somente em

caso de alguma mudança na sua matriz de transformação.

A taxa de alocação de memória possui um crescimento constante, à medida que o número de nodos é aumentado, apresentando um resultado satisfatório em relação à memória disponível.

3.5.2 Comparativo com trabalhos correlatos

Em comparação com os trabalhos correlatos listados na seção 2.4, o motor desenvolvido possui uma clara vantagem em relação ao motor MJ3I, no que diz respeito à renderização de personagens, pois permite a visualização e movimentação de articulações. Por outro lado, funcionalidades como detecção de colisão e importação de modelos 3D, recursos presentes no MJ3I, não foram implementadas no presente trabalho.

Em relação à ferramenta comercial Unity, possui renderização e movimentação de articulações, controle hierárquico de cena e controle de câmeras, porém a Unity possui uma quantidade muito superior de recursos, como iluminação, texturização, simulação de física, inteligência artificial, áudio e efeitos especiais. A aplicação desenvolvida pode ser utilizada como base para o desenvolvimento de uma ferramenta comercial semelhante à Unity.

A Tabela 2 sintetiza um comparativo entre as três ferramentas mencionadas.

Tabela 2 – Comparativo entre ferramentas

Funcionalidade	MJ3I-PA	MJ3I	Unity
Representação e movimentação de articulações	✔		✔
Controle hierárquico de cena	✔	✔	✔
Controle de câmeras	✔	✔	✔
Detecção de colisão		✔	✔
Importação de modelos 3D		✔	✔
Iluminação			✔
Texturização			✔
Física			✔
Inteligência artificial			✔
Áudio e efeitos especiais			✔

4 CONCLUSÕES

Este trabalho apresentou uma biblioteca utilizada para representação de personagens articulados em um ambiente virtual tridimensional e uma aplicação que faz uso desta biblioteca. Definiu-se para a aplicação três objetivos principais iniciais, dos quais dois foram alcançados. A aplicação deveria ser portada do Mac OS para o iOS, além de disponibilizar um ambiente virtual 3D para visualização de personagens articulados, objetivos esses que foram realizados com sucesso. O terceiro objetivo do trabalho, realizar a importação de um personagem através de um arquivo XML não foi implementado.

O diferencial do trabalho desenvolvido em relação ao trabalho correlato MJ3I é a implementação de articulações nos personagens renderizados, sendo possível movimentar cada uma das articulações que compõem um personagem.

A aplicação é limitada no que diz respeito à forma dos objetos renderizados, pois nem todas as classes da biblioteca V-ART foram alteradas no trabalho proposto. Dessa forma, a aplicação e todos os testes conduzidos utilizam objetos tridimensionais retangulares para representação dos personagens.

Os resultados obtidos nos testes para medição da taxa de FPS das aplicações indicam que com uma quantidade superior de braços e suas respectivas articulações sendo renderizados na cena, ocorre uma grande perda de desempenho ao escalonar o aplicativo. Pode-se dizer que com uma quantidade de doze ou mais nodos renderizados, a aplicação torna-se lenta, chegando ao ponto de tornar-se praticamente inutilizável ao renderizar 30 objetos.

Essa perda de desempenho da aplicação pode ser atribuída ao processo de conversão do código fonte. O OpenGL ES 2.0 trabalha com renderização de formas geométricas através de triângulos, diferentemente de outras versões do OpenGL onde é possível realizar a renderização de um objeto quadrilátero ou de um polígono. Dessa forma regras de negócio originais da biblioteca podem ser otimizadas levando em conta esse novo conceito, para um ganho maior de performance.

Os testes realizados quanto à alocação de memória do aplicativo indicaram que existe uma quantidade de memória fixa sendo alocada, para utilização de bibliotecas, arquivos da aplicação, `shaders` e demais componentes, que gira em torno de 593kb. A parte variável do programa diz respeito aos objetos renderizados na cena, nesse ponto o resultado obtido foi bastante satisfatório. Ao aumentarmos o número de objetos de 3 para 30, a aplicação

consumiu algo em torno de 90kb a mais de memória, conforme pode ser visualizado na Tabela 1. Dessa forma é possível estimar que cada braço adicionado na aplicação utiliza em torno de 3,3kb.

O desenvolvimento para a plataforma iOS é feito de forma bastante intuitiva para quem já possui um mínimo conhecimento prévio da linguagem e das ferramentas. As ferramentas disponibilizadas são um facilitador no que diz respeito à testes, entregando um simulador com performance muito semelhante ao dispositivo físico, além de funcionalidades para medição de desempenho e alocação de memória.

Com a entrega do trabalho proposto, o mesmo pode ser utilizado pelo grupo de Computação Gráfica e Entretenimento Digital da Universidade Regional de Blumenau (FURB) para desenvolvimento de aplicações gráficas em dispositivos iOS, somando às demais plataformas suportadas pelo V-ART.

4.1 EXTENSÕES

Como sugestões de extensão e melhoria do presente trabalho, tem-se:

a) alterar as demais classes da biblioteca V-ART, para ampliar a gama de opções de renderização;
b) implementar a importação de arquivo XML que defina personagens e animações;
c) implementar a utilização de luzes e texturas;
d) conversão da biblioteca para a linguagem Objective-c, afim de realizar comparativos de performance.

REFERÊNCIAS BIBLIOGRÁFICAS

APPLE. **The world's most advanced mobile operating system.** [S.l.], 2012a. Disponível em: <http://www.apple.com/iphone/ios/>. Acesso em: 19 nov. 2012.

_____. **Develop for iOS.** [S.l.], 2012b. Disponível em: <http://developer.apple.com/technologies/ios/>. Acesso em: 19 nov. 2012.

_____. **About the iOS technologies.** [S.l.], 2012c. Disponível em: <http://developer.apple.com/library/ios/#documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/Introduction/Introduction.html>. Acesso em: 19 nov. 2012.

_____. **Core OS layer.** [S.l.], 2012d. Disponível em: <http://developer.apple.com/library/ios/#documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/CoreOSLayer/CoreOSLayer.html#//apple_ref/doc/uid/TP40007898-CH11-SW1>. Acesso em: 19 nov. 2012.

_____. **Core services layer.** [S.l.], 2012e. Disponível em: <http://developer.apple.com/library/ios/#documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/CoreServicesLayer/CoreServicesLayer.html#//apple_ref/doc/uid/TP40007898-CH10-SW5>. Acesso em: 19 nov. 2012.

_____. **Media layer.** [S.l.], 2012f. Disponível em: <http://developer.apple.com/library/ios/#documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/MediaLayer/MediaLayer.html#//apple_ref/doc/uid/TP40007898-CH9-SW4>. Acesso em: 19 nov. 2012.

_____. **Cocoa touch layer.** [S.l.], 2012g. Disponível em: <http://developer.apple.com/library/ios/#documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/iPhoneOSTechnologies/iPhoneOSTechnologies.html#//apple_ref/doc/uid/TP40007898-CH3-SW1>. Acesso em: 19 nov. 2012.

_____. **iOS dev center.** [S.l.], 2012h. Disponível em: <http://developer.apple.com/devcenter/ios/index.action>. Acesso em: 19 nov. 2012.

_____. **Framework programming guide:** what are frameworks?**.** [S.l.], 2012i. Disponível em: <https://developer.apple.com/library/mac/#documentation/MacOSX/Conceptual/BPFrameworks/Concepts/WhatAreFrameworks.html>. Acesso em: 19 nov. 2012.

_____. **Foundation framework reference.** [S.l.], 2012j. Disponível em: <http://developer.apple.com/library/mac/#documentation/cocoa/reference/foundation/ObjC_classic/_index.html>. Acesso em: 19 nov. 2012.

APPLE. **UIKit framework reference.** [S.l.], 2012k. Disponível em: <http://developer.apple.com/library/ios/#documentation/uikit/reference/UIKit_Framework/_index.html>. Acesso em: 19 nov. 2012.

_____. **Core graphics framework reference.** [S.l.], 2012l. Disponível em: <http://developer.apple.com/library/ios/#documentation/coregraphics/reference/coregraphics_framework/_index.html>. Acesso em: 19 nov. 2012.

_____. **Quartz core framework reference.** [S.l.], 2012m. Disponível em: <http://developer.apple.com/library/mac/#documentation/graphicsimaging/reference/QuartzCoreRefCollection/_index.html>. Acesso em: 19 nov. 2012.

_____. **Core animation programming guide:** what is core animation? [S.l.], 2012n. Disponível em: <https://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/CoreAnimation_guide/Articles/WhatisCoreAnimation.html>. Acesso em: 19 nov. 2012.

AUTODESK. **Autodesk 3ds max products.** [S.l.], 2012. Disponível em: <http://usa.autodesk.com/3ds-max/>. Acesso em: 19 nov. 2012.

BLENDER. **Blender home.** [S.l.], 2012. Disponível em: <http://www.blender.org/>. Acesso em: 19 nov. 2012.

D'SOUZA, Aaron; SCHAAL, Stefan; VIJAYAKUMAR, Sethu. **Learning inverse kinematics**. [Estados Unidos], 2001. Disponível em: <http://www-clmc.usc.edu/~adsouza/papers/dsouza-IROS2001.pdf>. Acesso em: 19 nov. 2012.

DOXYGEN. **Generate documentation from source code.** [S.l.], 2012. Disponível em: <http://www.stack.nl/~dimitri/doxygen/>. Acesso em: 19 nov. 2012.

EBERLY, David H. **3D game engine architecture:** engineering real-time applications with wild magic. San Francisco: Morgan Kaufmann Publishers, 2005.

_____. **3D game engine design:** a practical approach to real-time computer graphics. San Diego: Morgan Kaufmann Publishers, 2001.

GSM ARENA. **Apple iPhone 4S** - full phone specifications. [S.l.], 2012. Disponível em: <http://www.gsmarena.com/apple_iphone_4s-4212.php>. Acesso em: 19 nov. 2012.

KHRONOS GROUP. **About the Khronos Group**. Beaverton, 2012a. Disponível em: <http://www.khronos.org/about>. Acesso em: 19 nov. 2012.

_____. **OpenGL ES**: the standard for embedded accelerated 3D graphics. Beaverton, 2012b. Disponível em: <http://www.khronos.org/opengles/>. Acesso em: 19 nov. 2012.

KHRONOS GROUP. **OpenGL ES 2.0 in more detail.** Beaverton, 2012c. Disponível em: <http://www.khronos.org/opengles/2_X/>. Acesso em: 19 nov. 2012.

_____. **The industry's standard for high performance graphics.** [S.l.], 2012d. Disponível em: <http://www.khronos.org/>. Acesso em: 19 nov. 2012.

_____. **The OpenGL ES shading language**. Beaverton, 2012e. Disponível em: <http://www.khronos.org/files/opengles_shading_language.pdf>. Acesso em: 19 nov. 2012.

MACIEL, Anderson. **Modelagem de articulações para humanos virtuais baseada em anatomia**. 2001. 101 f. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Rio Grande do Sul. Disponível em: <http://www.lume.ufrgs.br/bitstream/handle/10183/2629/000373843.pdf?sequence=1>. Acesso em: 19 nov. 2012.

MICROSOFT. **DirectX end-user runtime.** [S.l.], 2012a. Disponível em: <http://www.microsoft.com/download/en/details.aspx?id=35>. Acesso em: 19 nov. 2012.

_____. **DirectX**: perguntas freqüentes. [Redmond], 2012b. Disponível em: <http://windows.microsoft.com/pt-BR/windows-vista/DirectX-frequently-asked-questions>. Acesso em: 19 nov. 2012.

MUNSHI, Aaftab; GINSBURG, Dan; SHREINER, Dave. **OpenGL ES 2.0 programing guide**. Boston: Pearson Education, 2009.

OMNIGROUP. **Omnigraffle for mac:** diagramming worth a thousand words. [S.l.], 2012. Disponível em: <http://www.omnigroup.com/products/omnigraffle/>. Acesso em: 19 nov. 2012.

PATEL, Nilay. **iPhone OS 4 renamed iOS 4, launching june 21 with 1500 new features.** [S.l.], 2010. Disponível em: <http://www.engadget.com/2010/06/07/iphone-os-4-renamed-ios-gets-1500-new-features/>. Acesso em: 19 nov. 2012.

PCMAG. Smartphone. **PC magazine encyclopedia.** Estados Unidos, 2012a. Disponível em: <http://www.pcmag.com/encyclopedia_term/0,2542,t=Smartphone&i=51537,00.asp>. Acesso em: 19 nov. 2012.

_____. Tablet computer. **PC magazine encyclopedia.** Estados Unidos, 2012b. Disponível em: <http://www.pcmag.com/encyclopedia_term/0,1237,t=tablet+computer&i=52520,00.asp>. Acesso em: 19 nov. 2012.

PLIB. **PLIB:** a suite of portable game libraries. [S.l.], 2012. Disponível em: <http://plib.sourceforge.net/>. Acesso em: 19 nov. 2012.

RIDEOUT, Philip. **iPhone 3D programming**. Sebastopol: O'Reilly Media, 2010.

TAKANO, Rafael H. **MJ3I:** um motor de jogos 3D para o iPhone OS. 2009. 51 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

UNITY. **Unity:** Unity 3 engine features. [S.l.], 2010a. Disponível em: <http://unity3d.com/unity/>. Acesso em: 19 nov. 2012.

_____. **Scripting languages**. [S.l.], 2010b. Disponível em: <http://unity3d.com/unity/workflow/scripting>. Acesso em: 19 nov. 2012.

_____. **Publishing**. [S.l.], 2010c. Disponível em: <http://unity3d.com/unity/multiplatform/>. Acesso em: 19 nov. 2012.

_____. **NVIDIA PhysX**. [S.l.], 2010d. Disponível em: <http://unity3d.com/unity/quality/physics>. Acesso em: 19 nov. 2012.

_____. **Integrated editor**. [S.l.], 2010e. Disponível em: <http://unity3d.com/unity/workflow/integrated-editor>. Acesso em: 19 nov. 2012.

V-ART. **Virtual articulations for virtual reality**. [S.l.], 2009a. Disponível em: <http://vart.codeplex.com/>. Acesso em: 19 nov. 2012.

_____. **O framework V-ART**. [S.l.], 2009b. Disponível em: <http://algol.dcc.ufla.br/~bruno/v-art/>. Acesso em: 19 nov. 2012.

V-ART DOCS. **V-ART**: VART - dof class reference. [S.l.], 2009c. Disponível em: <http://algol.dcc.ufla.br/~bruno/v-art/docs/html/classVART_1_1Dof.html>. Acesso em: 19 nov. 2012.

VEPA, Ranjan. **Biomimetic robotics mechanisms and control**. [Londres], 2009. Disponível em: <http://ebooks.cambridge.org/chapter.jsf?bid=CBO9780511609688&cid=CBO9780511609688A028>. Acesso em: 19 nov. 2012.

VPAT. **Previous projects on computer graphics, visualization and interaction**. [S.l.], 2012. Disponível em: <https://wiki.inf.ufrgs.br/Previous_Projects_on_Computer_Graphics,_Visualization_and_Interaction>. Acesso em: 19 nov. 2012.

VRJUGGLER. **Code once**. [S.l.], 2012. Disponível em: <http://vrjuggler.org/features.php>. Acesso em: 19 nov. 2012.

VSG. **Open inventor:** overview. [S.l.], 2012. Disponível em: <http://vsg3d.com/open-inventor/sdk>. Acesso em: 19 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: 19 nov. 2012.

WRIGHT JR., Richard S. et al. **OpenGL super bible**: comprehensive tutorial and reference. 5th ed. Boston: Pearson Education, 2010.