UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

BIBLIOTECA DE REALIDADE AUMENTADA PARA A PLATAFORMA IOS

PAULO CESAR MEURER

BLUMENAU
2011

2011/2-21

PAULO CESAR MEURER

BIBLIOTECA DE REALIDADE AUMENTADA PARA A PLATAFORMA 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
2011

2011/2-21

BIBLIOTECA DE REALIDADE AUMENTADA PARA A PLATAFORMA IOS

Por

PAULO CESAR MEURER

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. Paulo Cesar Rodacki Gomes, Dr. – FURB
Membro:	Prof. Francisco Adell Péricas, M. Sc. – FURB

Blumenau, 13 de Dezembro de 2011

Dedico este trabalho a minha família, à minha namorada e aos amigos que me ajudaram diretamente na realização deste.

AGRADECIMENTOS

A Deus, pelo seu imenso amor e graça.

À minha família, que me incentivou durante a realização de todo o curso.

À minha namorada, Raquel, que teve paciência para aturar meu constante mau humor quando dos momentos de dificuldade.

Aos meus amigos, por compreenderem a minha ausência.

À Microton Tecnologia em Informação, pela compreensão e incentivo.

Ao meu orientador, Dalton Solano dos Reis, por sua fantástica presença de espírito, orientação e contribuição para a realização deste trabalho.

Se a única ferramenta que você possui é um martelo, você tende a ver todo problema como um prego.

Abraham Maslow

RESUMO

Este trabalho descreve o desenvolvimento de uma biblioteca para a criação de aplicativos de Realidade Aumentada na plataforma iOS, denominada RAios-library. A biblioteca desenvolvida é baseada na biblioteca OpenGL ES, para a construção e desenho de objetos virtuais e nas bibliotecas da plataforma iOS para a captura das imagens da câmera, localização e movimentos do dispositivo e realização da interação homem máquina (IHC), através da tela que permite múltiplos gestos. A biblioteca permite que sejam desenvolvidas aplicações que envolvem o conceito de Realidade Aumentada com geolocalização, sem exigir do desenvolvedor um conhecimento profundo de material de base, tal como computação gráfica e interface com o sistema operacional. A biblioteca não implementa rotinas para aquisição de dados dos pontos de interesse.

Palavras-chave: iOS. Realidade aumentada. Mobilidade.

ABSTRACT

This work describes the development of a library for building Augmented Reality applications on the iOS platform, called RAios-library. The developed library is based on OpenGL ES library, for the contruction and design of virtual objects and on iOS platform libraries for capturing images from the camera, device location and movements and to perform the Human-Computer Interaction (IHC), through the screen that allows multiple gestures. The library allows applications to be developed that involve the concept of Augmented Reality with geolocation, without requiring the developer a thorough knowledge of basic material, such as computer graphics and interface with the operating system. The library does not implement routines for data acquisition of points of interest.

Key-words: iOS. Augmented reality. Mobility.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE SIGLAS

2D – Duas Dimensões

3D – Três Dimensões

API – *Application Programming Interface*

ES – *Embedded System*

EAGL – *Embedded Apple Graphics Library*

FPS – Frames Por Segundo

GPS – *Global Position System*

Hz – *hertz*

LED – *Light-Emitting Diode*

OpenGL – *Open Graphics Library*

RA – Realidade Aumentada

RF – Requisito Funcional

RNF – Requisito Não-Funcional

RV – Realidade Virtual

UML – *Unified Modeling Language*

VBO – *Vertex Buffer Object*

Wi-Fi – *Wireless Fidelity*

WPS – Wi-Fi *Positioning Service*

SUMÁRIO

1 INTRODUÇÃO

Atualmente os sistemas de Realidade Aumentada (RA) se fazem presentes nos mais diversos segmentos da sociedade, tais como: publicidade, educação, comunicação e treinamento (KIRNER; TORI, 2004). É notável a transformação que esta tecnologia vem produzindo no cotidiano das pessoas, permitindo a vivência de experiências muito mais ricas com os objetos e o ambiente que as cercam. A possibilidade de explorar situações diferenciadas de visualização, imersão e interação através da RA cria uma condição que favorece a atenção e a retenção de informação, simplificando a realização de tarefas outrora consideradas complexas (DAINESE; GARBIN; KIRNER, 2003).

Observa-se que a crescente demanda por novas aplicações de RA e a sua potencialidade combinada com a explosão de popularidade e uso dos dispositivos móveis, cria muitas oportunidades e estimula cada vez mais os desenvolvedores a produzir sistemas para este mercado. O uso da RA em *smartphones* torna-se um meio atrativo para o usuário devido ao seu baixo custo e a ampla disponibilidade de recursos multimídia. Bajura e Neumann (1995) consideram que os sistemas de RA devem ser executados em dispositivos capazes de registrar e processar as imagens com precisão, levando o usuário a crer que os mundos real e virtual ocupam o mesmo espaço. Neste contexto, o iPhone destaca-se por ser um dispositivo móvel pequeno, leve, fácil de utilizar e que reúne capacidade de processamento e recursos multimídia mínimos para este tipo de aplicação.

A robustez, padronização de bibliotecas e da estrutura de programação e um poderoso *kit* de desenvolvimento tornam o iOS (sistema operacional do iPhone) uma plataforma ideal para a criação de sistemas de RA móvel. Porém, o desenvolvimento deste tipo de aplicação envolve a utilização e o gerenciamento de muitos recursos, como hardware, software, periféricos e redes, que exigem do desenvolvedor um conhecimento profundo de material de base, tal como computação gráfica e interface com o sistema operacional. Surge então a necessidade de criar bibliotecas e ferramentas de desenvolvimento que promovam o reuso de estruturas e algoritmos comuns à gerência dos recursos utilizados, simplificando a criação de novas aplicações. Estas ferramentas de alto nível permitem que o desenvolvedor foque apenas no aplicativo em si, abstraindo a complexidade e reduzindo o tempo de desenvolvimento do software (GUIMARÃES; GNECCO; DAMAZIO, 2007).

Diante do exposto, é proposta a criação de uma biblioteca de software que simplifique a utilização dos recursos disponíveis na plataforma iOS na criação de aplicações de RA.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver uma biblioteca de RA em Objective-C para a plataforma iOS.

Os objetivos específicos do trabalho são:

a) permitir o uso de recursos disponíveis pela plataforma, tais como *Global Position System* (GPS), acelerômetro, bússola e câmera digital;
b) utilizar o conceito *markerless tracking*, efetuando o registro dos objetos virtuais através de coordenadas geográficas;
c) criar uma camada de abstração que oculte os detalhes de implementação da aquisição de imagens da câmera, do registro, do rastreamento, do ajuste visual dos objetos virtuais e da sobreposição das cenas do mundo real e virtual;
d) disponibilizar um aplicativo exemplo desenvolvido utilizando a biblioteca criada.

1.2 ESTRUTURA DO TRABALHO

A estrutura deste trabalho está apresentada em quatro capítulos, sendo que o segundo capítulo contém a fundamentação teórica necessária para o entendimento deste trabalho.

O terceiro capítulo apresenta como foi desenvolvida a biblioteca na plataforma iOS, os casos de uso, os diagramas de classe e toda a especificação que define a biblioteca. Ainda no terceiro capítulo são apresentadas as partes principais da implementação e também os resultados e discussões que aconteceram durante toda a etapa de desenvolvimento do trabalho.

Por fim, o quarto capítulo refere-se às conclusões do presente trabalho e sugestões para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Na seção 2.1 é apresentado o conceito de RA e suas características. A seção 2.2 contém uma introdução à plataforma iOS, incluindo os componentes do sistema operacional, os aspectos básicos do desenvolvimento de aplicações e os principais recursos utilizados para o desenvolvimento da biblioteca proposta. Por fim, a seção 2.3 apresenta três trabalhos correlatos ao trabalho proposto.

2.1 REALIDADE AUMENTADA

Bimber e Raskar (2005, p. 2) descrevem a RA como um meio de inserir informação sintética[1] dentro de um ambiente real. Esta definição, apesar de genérica, é importante para que se possa compreender a diferença entre RA e Realidade Virtual (RV). Enquanto a RV imerge o usuário em um ambiente completamente artificial, a intenção da RA é manter o vínculo do usuário com a cena real, permitindo que ele a perceba normalmente, apenas acrescida de novas informações virtuais (AZUMA, 1997, p. 356). O resultado desta integração entre uma cena real e estímulos gerados por computador é um ambiente híbrido, mais rico e completo, que permite ao usuário explorar situações diferenciadas de visualização e interação com os objetos e o lugar que o cercam. Porém, para que este efeito seja alcançado por um sistema de RA, o mesmo deve possuir três características básicas definidas por Azuma et al. (2001, p. 34), que são abordadas nas próximas seções.

2.1.1 Integração entre real e virtual em um ambiente real

A primeira característica descrita por Azuma (1997, p. 356) trata da combinação da realidade com a virtualidade em um cenário real. Para esta combinação, é preciso o uso de algum dispositivo de formação de imagens que permita a inserção, em tempo real, de

[1] Informação sintética refere-se a um estímulo virtual dos sentidos. Pode ser a simulação de uma imagem, um toque, um som, um odor ou até mesmo um sabor.

informações sintéticas em algum lugar no campo de visão entre o olho do observador e o ambiente real, transmitindo ao usuário a sensação de que os objetos virtuais coexistem com os reais (BIMBER; RASKAR, 2005, p. 71).

Atualmente a maior parcela das aplicações de RA utiliza dispositivos do tipo *head-mounted* (acoplados à cabeça) para a combinação da realidade com a virtualidade. Isso se deve, em grande parte, ao fato de representarem a única possibilidade de suporte a RA móvel durante muitos anos (BIMBER; RASKAR, 2005, p. 5). Entretanto, Azuma et al. (2001, p. 35) observam que estes dispositivos constituem um fator limitante no desenvolvimento de aplicações de RA devido ao fato de não possuírem resolução, campo de visão, brilho e contraste suficientes para combinar uma grande variedade de imagens reais e virtuais.

Observando a crescente evolução tecnológica dos *tablets* e *smartphones*, Bimber e Raskar (2005, p. 5) concluem que os dispositivos *hand-held* (móveis) são a plataforma de exibição mais promissora da atualidade, pois além de permitir a combinação entre o real e o virtual, conferem ao usuário maior liberdade de movimentação, uma vez que não possuem restrição de mobilidade causada por fios e equipamentos acoplados ao corpo do usuário. Neste tipo de dispositivo, o *video see-throuth* (visão através de vídeo) é a abordagem mais utilizada. Uma câmera integrada ao dispositivo captura o fluxo de vídeo do ambiente que é então sobreposto por objetos gráficos, em tempo real, antes de ser exibido ao usuário (BIMBER; RASKAR, 2005, p. 79).

Algumas aplicações de RA, porém, não necessitam de mobilidade. Nestes casos, Bimber e Raskar (2005, p. 5) consideram que os dispositivos *spatial display* (dispositivo espacial) são muito mais eficientes, pois conseguem alcançar um nível de qualidade e realismo maior do que qualquer dispositivo móvel (*hand-held* ou *head-mounted*) da atualidade.

2.1.2 Interação em tempo real

A segunda característica descrita por Azuma (1997, p. 356) trata da interatividade em tempo real. Uma vez que a realidade aumentada concentra-se em sobrepor o ambiente real por elementos gráficos, os mesmos devem ser capazes de criar uma ilusão convincente de realidade, incorporando as sombras do ambiente e o comportamento de inter-reflexão dos demais objetos. A manutenção de uma ilusão realista é muito importante para que seja

mantida a idéia de continuidade entre o mundo real e virtual e, para tanto, métodos para renderização em tempo real têm um importante papel (BIMBER; RASKAR, 2005, p. 5).

Segundo Fournier (1994, p. 2), existem dois tipos de problemas de realismo na RA: os geométricos e os de iluminação. Os problemas geométricos residem na computação de parâmetros de visualização, que remetem ao problema do registro apresentado na seção 2.1.3, e na determinação da visibilidade dos objetos virtuais. Já os problemas de iluminação consistem na iluminação de objetos reais com fontes de luz geradas por computador, iluminação de objetos virtuais com fontes de luz real e problemas considerados secundários como reflexos, transparências e sombras.

2.1.3 Registro e rastreamento dos objetos em terceira dimensão

A terceira e última característica descrita por Azuma (1997, p. 356) trata do registro dos objetos virtuais em terceira dimensão. Entende-se registro como o alinhamento dos objetos virtuais com a cena real através de algum mecanismo de rastreamento que pode ser ótico, mecânico, magnético, inercial ou ultrassônico.

Bimber e Raskar (2005, p. 4) consideram que o sucesso de uma aplicação de RA depende do registro e rastreamento preciso, rápido e robusto da posição dos objetos reais e virtuais em relação ao usuário e seu ponto de vista, pois o desalinhamento entre estes objetos pode causar a perda de noção da realidade.

De acordo com Cawood e Fiala (2007, p. 4), existem vários métodos para registrar a posição dos objetos virtuais no ambiente real, que vão desde a utilização de *Light-Emitting Diodes* (LEDs) até a mão de uma pessoa. Entretanto, o modo mais simples de registro é conhecido como *marker-based tracking*. Este método consiste em definir um marcador que possui um padrão único e adicioná-lo fisicamente no ambiente real. Quando este padrão é reconhecido pelo sistema de RA, é possível determinar a posição e o ângulo do marcador e então utilizar estas informações para exibir o objeto virtual na posição e orientação corretas.

O uso de marcadores, porém, não é efetivo quando o sistema de RA é executado em grandes áreas abertas. Nestes ambientes o registro é comumente feito sem o uso de marcadores, utilizando um método conhecido como *markerless tracking*. Cawood e Fiala (2007, p. 6) consideram este método de registro e rastreamento como sendo o ideal, pois não exige a adição prévia de marcadores no cenário. Para Bimber e Raskar (2005, p. 5), esta é a

solução mais promissora para o futuro das aplicações de RA, porém, mesmo contando com o apoio de tecnologias como o GPS para registrar os objetos e dispositivos de medição relativa como o giroscópio e o acelerômetro para determinar a posição e orientação corretas, é a solução mais desafiadora.

2.2 PLATAFORMA IOS

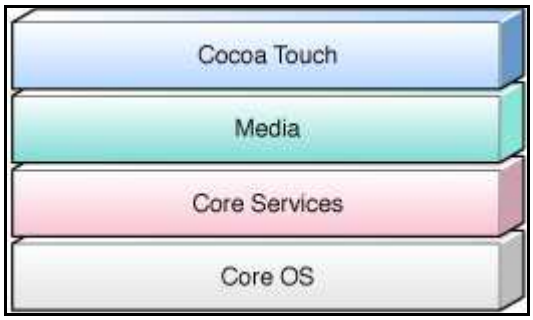
O iOS é uma plataforma de software que abrange o sistema operacional e um conjunto de tecnologias que são utilizadas para executar aplicativos nativamente nos dispositivos móveis da empresa Apple, como o iPad, iPhone e iPod Touch. Embora sua estrutura herde muitas tecnologias da plataforma Mac OS X[2], o iOS possui funcionalidades adicionais, como por exemplo a interface *Multi-Touch* e suporte ao acelerômetro, para satisfazer as necessidades de um ambiente móvel, onde as necessidades dos usuários são ligeiramente diferentes (APPLE INC, 2010a).

2.2.1 Arquitetura do sistema operacional

Nas camadas mais baixas, a arquitetura do iOS é semelhante à arquitetura básica do Mac OS X. Estes componentes compartilhados entre as duas plataformas são conhecidos coletivamente como Darwin (RIDEOUT, 2010, p. 1).

Em seu nível mais alto, porém, o iOS possui um conjunto de tecnologias que age como um intermediário entre o hardware e os aplicativos que aparecem na tela. A implementação destas tecnologias pode ser vista como um conjunto de quatro camadas: *Cocoa Touch*, *Media*, *Core Services* e *Core OS*, como observado na Figura 1. A abstração fornecida por estas camadas faz com que as aplicações dificilmente comuniquem-se diretamente com o hardware, permitindo que funcionem de forma consistente mesmo em dispositivos com recursos de hardware diferentes (APPLE INC, 2010b).

[2] Refere-se a versão 10 do sistema operacional Macintosh.



Fonte: Apple Inc (2010b).

Figura 1 – Visão geral das camadas de alto nível da plataforma iOS

Nas duas camadas inferiores estão os serviços fundamentais e as tecnologias de que todos os aplicativos dependem, como o acesso a arquivos, tipos de dados de baixo nível e *sockets* de redes, por exemplo. Já as duas camadas superiores fornecem abstrações orientadas a objetos das camadas de nível inferior e disponibilizam serviços e tecnologias mais sofisticados.

A camada *Core OS* contém os recursos de baixo nível sobre os quais a maior parte das outras tecnologias é construída. Geralmente é acessada pelas outras camadas, a não ser em casos onde há necessidade de lidar explicitamente com questões de segurança ou efetuar comunicação com algum equipamento externo. Esta camada lida basicamente com acesso ao sistema de arquivos, entrada e saída padrão de dados, alocação de memória, computações matemáticas e segurança dos dados manipulados pelas aplicações (APPLE INC, 2010f).

A camada *Core Services* oferece os serviços fundamentais que são utilizados em todas as aplicações, como por exemplo, o *framework CFNetwork* que oferece suporte para trabalhar com os protocolos de rede, o *framework Core Data* que permite utilizar ferramentas gráficas para construir e gerenciar o modelo de dados de uma aplicação e o *framework Foundation* que oferece suporte às coleções de dados, pacotes, gerenciamento de preferências, *strings*, data, hora, *threads*, etc. (APPLE INC, 2010e).

A camada *Media* contém os *frameworks* necessários para acessar a maioria dos protocolos de áudio e vídeo. Dentre esses, o *framework AV Foundation* oferece serviços como edição de mídia, reprodução de áudio e vídeo, gerência de metadados para itens de mídia e a captura de vídeo. Esta camada ainda oferece suporte a desenhos 2D e 3D através de tecnologias gráficas como o OpenGL ES[3], o *Embedded Apple Graphics Library* (EAGL), o *Quartz* e o *Core Animation* (APPLE INC, 2010d).

A camada *Cocoa Touch* contém os *frameworks* chave para a construção de aplicações no iOS. Nela são definidas a infra-estrutura básica das aplicações e o suporte para tecnologias

[3] *Open Graphics Library for Embedded Systems* (OpenGL ES).

fundamentais, tais como multitarefa, entrada baseada em toque, notificações *push*, reconhecedores de gestos e diversos serviços de alto nível. O *framework* UIKit, por exemplo, disponibiliza recursos em interfaces gráficas, como as classes de visões, controles, janelas e os controladores destes objetos (APPLE INC, 2010c).

2.2.2 OpenGL ES

O OpenGL ES é uma versão simplificada do OpenGL que elimina algumas funcionalidades redundantes, com a finalidade de fornecer uma *Application Programming Interface* (API) reduzida para uso em dispositivos embarcados como os *smartphones*.

Por ser uma API baseada na linguagem C, o OpenGL ES é extremamente portável e se integra perfeitamente com aplicações *Cocoa Touch* desenvolvidas com Objective-C. Porém, a especificação do OpenGL ES não define uma camada de janelas, em vez disso, o sistema operacional deve disponibilizar funções para criar um contexto de renderização do OpenGL que aceita comandos e um *framebuffer* onde os resultados dos comandos de desenho são escritos. Para tanto, o iOS disponibiliza o *framework* EAGL, que age como uma interface entre o OpenGL ES e o UIKit (APPLE INC, 2011a).

Um detalhe importante a ser observado quando se utiliza o OpenGL ES no desenvolvimento de um aplicativo é que apesar de possuir um poderoso sistema *multitasking*, o iOS não permite que aplicações que estejam executando em *background* chamem as suas funções. Isso é feito para que o processador gráfico esteja sempre completamente disponível para a aplicação que executa em *foreground*. Quando um aplicativo acessa o processador gráfico enquanto está em *background*, ele é automaticamente encerrado pelo iOS. Esta restrição não se aplica somente a novas chamadas de funções do OpenGL ES, mas também a comandos que foram previamente submetidos e não tenham sido completados. Portanto, as aplicações que utilizam o OpenGL ES devem garantir que todos os comandos previamente submetidos sejam concluídos antes de mudar o seu estado para o *background* (APPLE INC, 2011a).

2.2.3 Bibliotecas dinâmicas `.dylib`

Grande parte das funcionalidades de uma aplicação provém de bibliotecas de código executável implementadas pelos sistemas operacionais. Durante o desenvolvimento de uma aplicação, o desenvolvedor deve ligar o código de sua aplicação com estas bibliotecas para obter acesso a estas funcionalidades. Esta ligação, quando ocorre de maneira estática, faz com que código de biblioteca que o aplicativo utiliza seja copiado para o arquivo executável gerado e carregado na memória durante toda a sua execução.

Segundo Apple Inc (2009), dois fatores importantes para que os aplicativos apresentem boa performance são o tempo gasto na inicialização e o uso correto da memória. Estes requisitos podem ser alcançados reduzindo o tamanho do arquivo executável e o uso de memória pelas aplicações. Uma solução que atende a estas necessidades é o uso de bibliotecas dinâmicas, que permite às aplicações retardarem o carregamento de bibliotecas de funcionalidades especiais até o momento em que elas são necessárias, ao invés de carregar a biblioteca inteira durante sua inicialização. As aplicações escritas para o iOS são beneficiadas por este recurso, pois todas as bibliotecas de sistema, os arquivos `.dylib`, no Mac OS X são dinâmicas.

2.3 TRABALHOS CORRELATOS

Existem várias ferramentas para a RA que auxiliam na construção dos objetos virtuais e em sua integração com o ambiente real. Para o uso em computadores pessoais, pode-se citar o ARToolkit (LAMB, 2010). Nas plataformas móveis, dentre outras, pode-se destacar ferramentas como o Junaio (METAIO INC, 2011), Layar (LAYAR, 2010) e Vasselai (2010).

O ARToolkit é um software que utiliza marcadores para o rastreamento e registro de objetos virtuais no ambiente real. Estes marcadores são expressos por padrões 2D na forma de um quadrado com bordas pretas e são previamente cadastrados e inseridos no ambiente real antes da execução da aplicação. A detecção destes marcadores é feita analisando a imagem com o auxílio de técnicas de visão computacional. Segundo Lamb (2010), as seguintes etapas são executadas para detectar os marcadores: a imagem capturada é transformada em uma imagem binária. Logo após, o software analisa a imagem e detecta os marcadores e os

compara com os previamente cadastrados. Quando um marcador é encontrado, um objeto virtual é adicionado à imagem real, na posição e orientação do marcador original.

Junaio é um *framework* aberto disponível para as plataformas iOS, Android e Nokia (N8) que permite a criação de aplicações de realidade aumentada. Os Aplicativos criados com o Junaio permitem a renderização de objetos 3D e registro dos pontos de interesse utilizando marcadores e coordenadas GPS. O Junaio permite a exibição de informação dos pontos de interesse, reprodução de áudio e vídeo, consulta do mapa e ainda oferece um serviço de publicação e hospedagem de conteúdo que permite aos editores criar seu próprio canal de pontos de interesse sem qualquer programação (METAIO INC, 2011). A Figura 2 mostra uma aplicação desenvolvida utilizando o *framework* Junaio.

Fonte: Android Apps (2011).

Figura 2 - Aplicação desenvolvida no *framework* Junaio

Layar é um *framework* que permite o desenvolvimento de aplicativos com realidade aumentada através de um servidor próprio (LAYAR, 2010). Os aplicativos criados com o Layar utilizam GPS, bússola e câmera para identificar o ambiente real e mostrar, em tempo real, informação digital no campo de visão do usuário através do seu dispositivo móvel. Conforme Layar (2010), o Layar destaca-se por ser uma plataforma que suporta um nível sofisticado de interatividade, incluindo elementos de áudio e vídeo, modelos 3D e capacidades de compartilhamento social. A Figura 3 mostra o desenho das antigas torres World Trade Center em 3D na localização exata onde estavam as torres originais, totalmente renderizadas no aplicativo Layar.

Fonte: Layar (2011).

Figura 3 – Torres do World Trace Center em 3D desenvolvidas no Layar

Vasselai (2010) descreve uma aplicação de RA (Figura 4) desenvolvida para a plataforma Android que apresenta características semelhantes às propostas por este trabalho, dentre as quais se podem destacar:

a) os objetos virtuais são registrados através de coordenada geográfica;
b) utiliza a câmera do dispositivo para capturar as imagens reais;
c) permite selecionar os objetos virtuais através do multitoque;
d) utiliza o GPS e o acelerômetro do dispositivo;
e) os pontos de interesse podem ser objetos 2D.

Fonte: Vasselai (2010, p. 85).

Figura 4 – Aplicação FURB Realidade Aumentada

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 tecnologias 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 para a biblioteca proposta são:

a) permitir visualizar o ambiente real através da câmera do dispositivo (Requisito Funcional - RF);
b) sobrepor ao ambiente real com objetos virtuais em 2D (RF);
c) utilizar o multitoque para visualizar detalhes dos objetos virtuais (RF);
d) utilizar o conceito de *markerless tracking* para registro dos objetos virtuais através de coordenada geográfica (RF);
e) utilizar o acelerômetro para rastrear a direção que o usuário está visualizando com o dispositivo (Requisito Não Funcional - RNF);
f) ser implementado na plataforma iOS (RNF).

3.2 ESPECIFICAÇÃO

A especificação da biblioteca em questão foi desenvolvida seguindo a análise orientada a objetos, utilizando a notação *Unified Modeling Language* (UML) em conjunto com a ferramenta Enterprise Architect 6.5.802 para a elaboração dos casos de uso e dos diagramas de classes.

3.2.1 Visão da biblioteca

A biblioteca de realidade aumentada RAios-library é uma camada intermediária que se localiza entre a aplicação e o OpenGL ES, como pode ser observado na Figura 5. Apesar de a biblioteca disponibilizar rotinas para o desenvolvimento da aplicação, ainda é possível que o desenvolvedor acesse diretamente o OpenGL ES. A RAios-library também utiliza outros *frameworks* disponibilizados pela Apple como o UIKit que possibilita capturar os toques na tela, o *Core Motion* que permite a manipulação dos dados provenientes do acelerômetro, o *Core Location* para a manipulação dos dados recebidos do GPS e o AV *Foundation* para acessar e manipular a câmera do dispositivo.

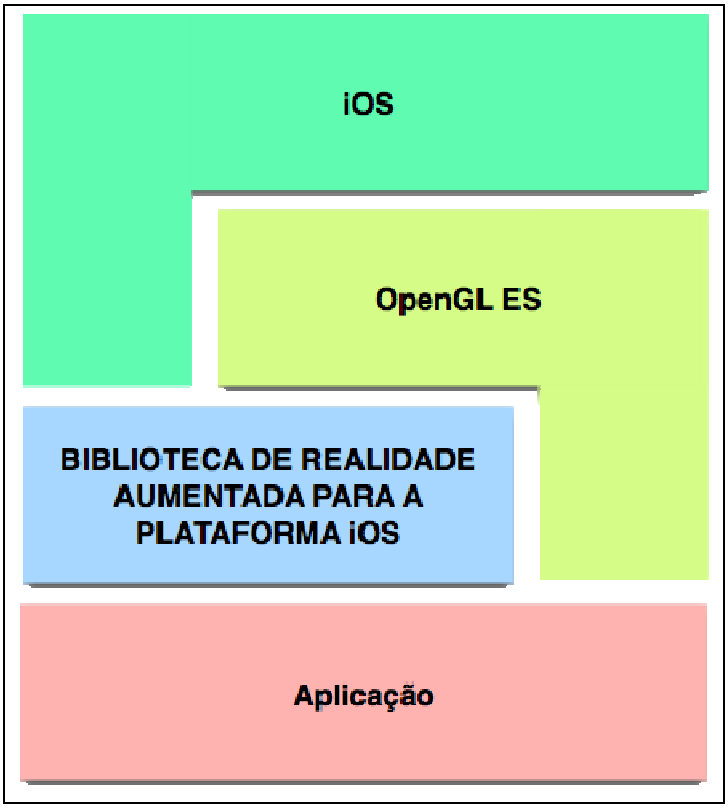


Figura 5 – Visão da camada da RAios-library

3.2.2 Casos de uso

Nesta seção são descritos os casos de uso da RAios-library. Foi identificado como sendo o ator principal o `Desenvolvedor` que irá utilizar os recursos disponibilizados. Na Figura 6 é apresentado o diagrama de casos de uso da RAios-library.

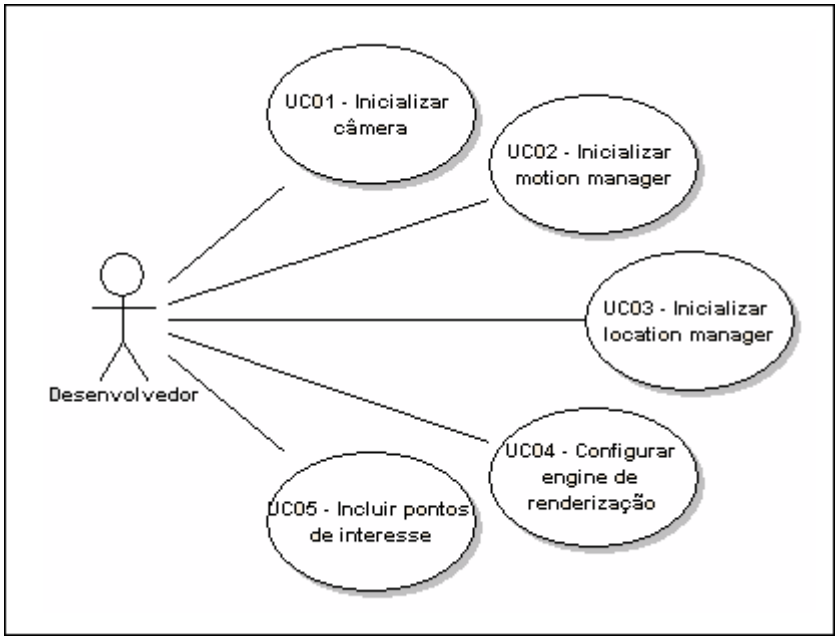


Figura 6 – Diagrama de casos de uso

3.2.2.1 Inicializar câmera

Este caso de uso descreve o processo de configuração e inicialização da captura de vídeo efetuado pela aplicação. Detalhes deste caso de uso estão descritos no Quadro 1.

UC01 – Inicializar câmera	
Descrição	Para combinar a realidade com a virtualidade a aplicação deve permitir ao usuário a visualização do mundo real através do dispositivo móvel.
Pré-Condição	O `Desenvolvedor` deve ter inicializado a biblioteca corretamente, fornecendo uma `view` onde os frames capturados pela câmera do dispositivo serão exibidos. O dispositivo deve possuir uma câmera ativa.
Cenário Principal	1. O `Desenvolvedor` utiliza o método `arVideoCapConfigFrameRate` da biblioteca e informa a taxa de *frames* por segundo que a câmera irá capturar; 2. A biblioteca valida e configura a câmera com a informação passada; 3. O `Desenvolvedor` utiliza o método `arVideoCapConfigVideoQuality` da biblioteca e informa a qualidade desejada para os *frames* capturados pela câmera; 4. A biblioteca valida e configura a câmera com a informação passada; 5. O `Desenvolvedor` utiliza o método `arVideoCapStart` da biblioteca para iniciar a captura de vídeo; 6. As imagens da câmera são exibidas na tela do dispositivo.
Exceção 1	1. Se no passo 2 do cenário principal a configuração passada esteja incorreta, a biblioteca configura a câmera com uma taxa de 60 *frames* por segundo.
Exceção 2	1. Se no passo 4 do cenário principal a configuração passada esteja incorreta, a biblioteca configura a câmera com qualidade média para os *frames* capturados.
Pós-Condição	As imagens da câmera são exibidas na `view` configurada.

Quadro 1 – Caso de uso UC01

3.2.2.2 Inicializar *motion* manager

Este caso de uso descreve o processo de configuração e inicialização da captura dos dados do acelerômetro pela aplicação. Detalhes deste caso de uso estão descritos no Quadro 2.

UC02 – Inicializar *motion* manager	
Descrição	A biblioteca deve permitir ao `Desenvolvedor` configurar o acelerômetro para que a resposta da aplicação às mudanças de direção do dispositivo seja a mais precisa possível, mantendo o alinhamento dos objetos virtuais com o ambiente real e evitando a perda da noção de realidade.
Pré-Condição	O dispositivo deve possuir um sensor de acelerômetro ativo.
Cenário Principal	1. O `Desenvolvedor` utiliza o método `arMotionConfigAccelUpdateFrequency` da biblioteca e informa a taxa de atualização da leitura do acelerômetro; 2. A biblioteca valida e configura o acelerômetro com a informação passada; 3. O `Desenvolvedor` utiliza o método `arMotionConfigGravityFilterFactor` e informa o valor do filtro de ação da gravidade sobre o acelerômetro; 4. A biblioteca valida e configura o acelerômetro com informação passada; 5. O `Desenvolvedor` utiliza o método `arMotionStart` da biblioteca para iniciar a captura dos movimentos do dispositivo; 6. A aplicação começa a responder aos movimentos do dispositivo.
Exceção 1	1. Se no passo 5 do cenário principal o sensor do acelerômetro não estiver disponível, a chamada ao método falha e uma mensagem de erro é gravada no registro de eventos da aplicação.
Exceção 2	1. Se no passo 6 do cenário principal ocorrer erro na leitura dos dados do acelerômetro a captura de movimentos do dispositivo é encerrada e uma mensagem com o erro encontrado é gravada no registro de eventos da aplicação.
Pós-Condição	A aplicação deve começar a responder aos movimentos do dispositivo.

Quadro 2 – Caso de uso UC02

3.2.2.3 Inicializar *location* manager

Este caso de uso descreve o processo de configuração e inicialização da captura dos dados do GPS e da bússola pela aplicação. Detalhes deste caso de uso estão descritos no Quadro 3.

UC03 – Inicializar *location* manager	
Descrição	A biblioteca deve permitir ao `Desenvolvedor` configurar e gerenciar o GPS e a bússola para que a aplicação possa responder às mudanças de localização e direção e comece a efetuar o rastreamento dos objetos virtuais.
Pré-Condição	O dispositivo deve possuir um serviço de localização geográfica e outro de orientação magnética, todos ativos.
Cenário Principal	1. O `Desenvolvedor` utiliza o método `arLocationConfigFiltroDistanciaGPS` da biblioteca e informa a distância lateral mínima a ser percorrida para atualização da localização do dispositivo; 2. A biblioteca valida e configura o GPS com a informação passada; 3. O `Desenvolvedor` utiliza o método `arLocationConfigDistanciaMaximaObjeto` da biblioteca e informa a distância máxima entre a localização atual do dispositivo e os pontos de interesse; 4. A biblioteca valida e configura o GPS com a informação passada; 5. O `Desenvolvedor` utiliza o método `arLocationConfigFiltroBussola` da biblioteca e informa o valor da mudança angular mínima para gerar novos eventos da bússola; 6. A biblioteca valida e configura a bússola com as informações passadas; 7. O `Desenvolvedor` utiliza o método `arLocationStart` da biblioteca para iniciar a captura das mudanças de localização e direção do dispositivo; 8. A Aplicação começa a responder às mudanças de localização e direção do dispositivo.
Exceção	1. Se no passo 8 do cenário principal o sensor de orientação magnética registrar erros de leitura uma mensagem de erro é gravada no registro de eventos da aplicação.
Pós-Condição	A aplicação deve começar a responder às mudanças de localização e direção do dispositivo.

Quadro 3 – Caso de uso UC03

3.2.2.4 Configurar *engine* de renderização

Este caso de uso descreve o processo de configuração do motor de renderização dos objetos virtuais. Detalhes deste caso de uso estão descritos no Quadro 4.

UC04 – Configurar *engine* de renderização	
Descrição	A biblioteca deve permitir ao `Desenvolvedor` configurar o motor de renderização para adequar a distância e o raio de disposição dos objetos na tela de acordo com as suas necessidades.
Pré-Condição	A biblioteca de RA deve ter sido inicializada.
Cenário Principal	1. O `Desenvolvedor` utiliza o método `arDrawConfigRaioDesenhoObjeto` da biblioteca e informa o raio de disposição dos objetos na tela; 2. A biblioteca valida e configura o motor gráfico com a informação passada; 3. O `Desenvolvedor` utiliza o método `arDrawConfigDistanciaObjetoTela` da biblioteca e informa a distância que os objetos devem ser desenhados em relação a tela; 4. A biblioteca valida e configura o motor gráfico com a informação passada; 5. O `Desenvolvedor` utiliza o método `arDrawStart` da biblioteca para iniciar a renderização dos pontos de interesse cadastrados; 6. A aplicação começa a exibir os pontos de interesse cadastrados.
Cenário Alternativo 1	1. Se no passo 6 do cenário principal a aplicação apresentar lentidão na renderização dos pontos de interesse o `Desenvolvedor` pode optar por desabilitar a renderização de texturas utilizando o método `arDrawConfigHabilitarTexturas` fornecido pela biblioteca.
Pós-Condição	A aplicação deve começar a exibir os pontos de interesse cadastrados.

Quadro 4 – Caso de uso UC04

3.2.2.5 Incluir pontos de interesse

Este caso de uso descreve o processo de inclusão de pontos de interesse na aplicação. Detalhes deste caso de uso estão descritos no Quadro 5.

UC05 – Incluir pontos de interesse	
Descrição	Os pontos de interesse são os objetos virtuais que serão sobrepostos à realidade. A biblioteca deve permitir ao `Desenvolvedor` cadastrar os pontos de interesse que serão exibidos na aplicação.
Pré-Condição	A biblioteca de RA deve ter sido inicializada.
Cenário Principal	1. O `Desenvolvedor` utiliza o método `arInsertObject` da biblioteca passando um objeto da classe `PontoInteresse` como parâmetro; 2. O ponto de interesse é inserido na lista de pontos de interesse da aplicação e é exibido quando estiver no campo de visão do dispositivo.
Pós-Condição	Um ponto de interesse é adicionado na lista de pontos de interesse da aplicação.

Quadro 5 – Caso de uso UC05

3.2.3 Diagrama de classes

Nesta seção são descritas as classes e estruturas desenvolvidas que formam toda a RAios-library. Para facilitar o entendimento de como as classes estão reunidas, na Figura 7 estão sendo demonstrados os pacotes, suas dependências bem como as classes que os compõem.

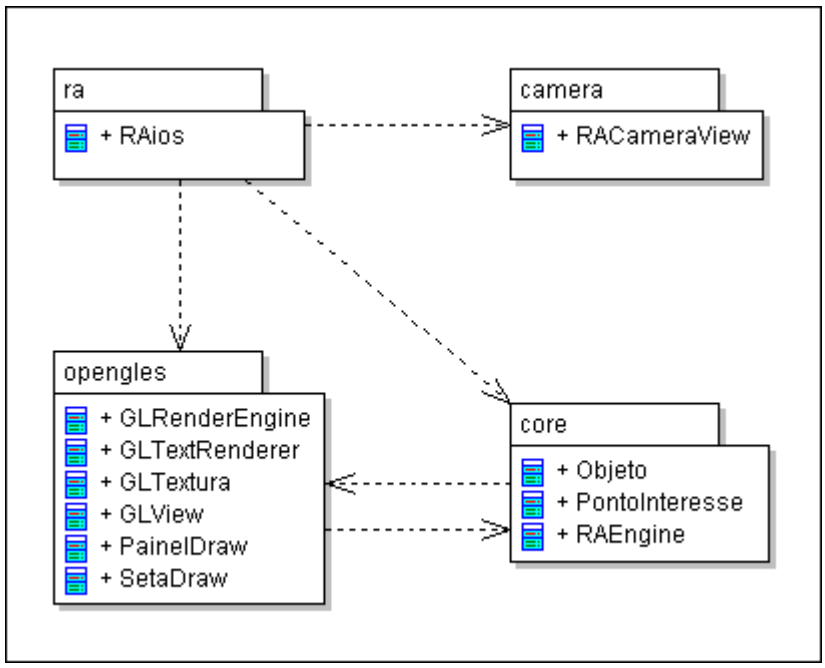


Figura 7 – Diagrama de pacotes da RAios-library

3.2.3.1 Pacote `core`

O primeiro pacote (Figura 8) é denominado `core` e abriga as classes relacionadas à gerência e ao comportamento dos objetos virtuais.

A classe `RAEngine` é a principal classe deste trabalho. Possui a responsabilidade de executar todos os cálculos da aplicação, estabelecendo o posicionamento em relação ao observador, de cada objeto virtual que será exibido na tela do dispositivo. Estes cálculos são efetuados com base nas informações de mudança das coordenadas geográficas, do ângulo da bússola e dos valores do acelerômetro.

A informação sobre mudança das coordenadas geográficas é obtida através da implementação do método `locationManager:didUpdateToLocation:fromLocation:` do protocolo `CLLocationManagerDelegate` que informa ao *delegate* que um novo valor de

localização está disponível. A mudança angular da bússola é obtida através da implementação do método `locationManager:didUpdateHeading:` do protocolo `CLLocationManagerDelegate` que informa ao *delegate* que o *location manager* recebeu uma informação atualizada da bússola. Por fim, as alterações no valor do acelerômetro são obtidas através do método `startAccelerometerUpdatesToQueue:withHandler:` da classe `CMMotionManager`, que disponibiliza as atualizações do acelerômetro em uma fila de operações do tipo `NSOperationQueue`.

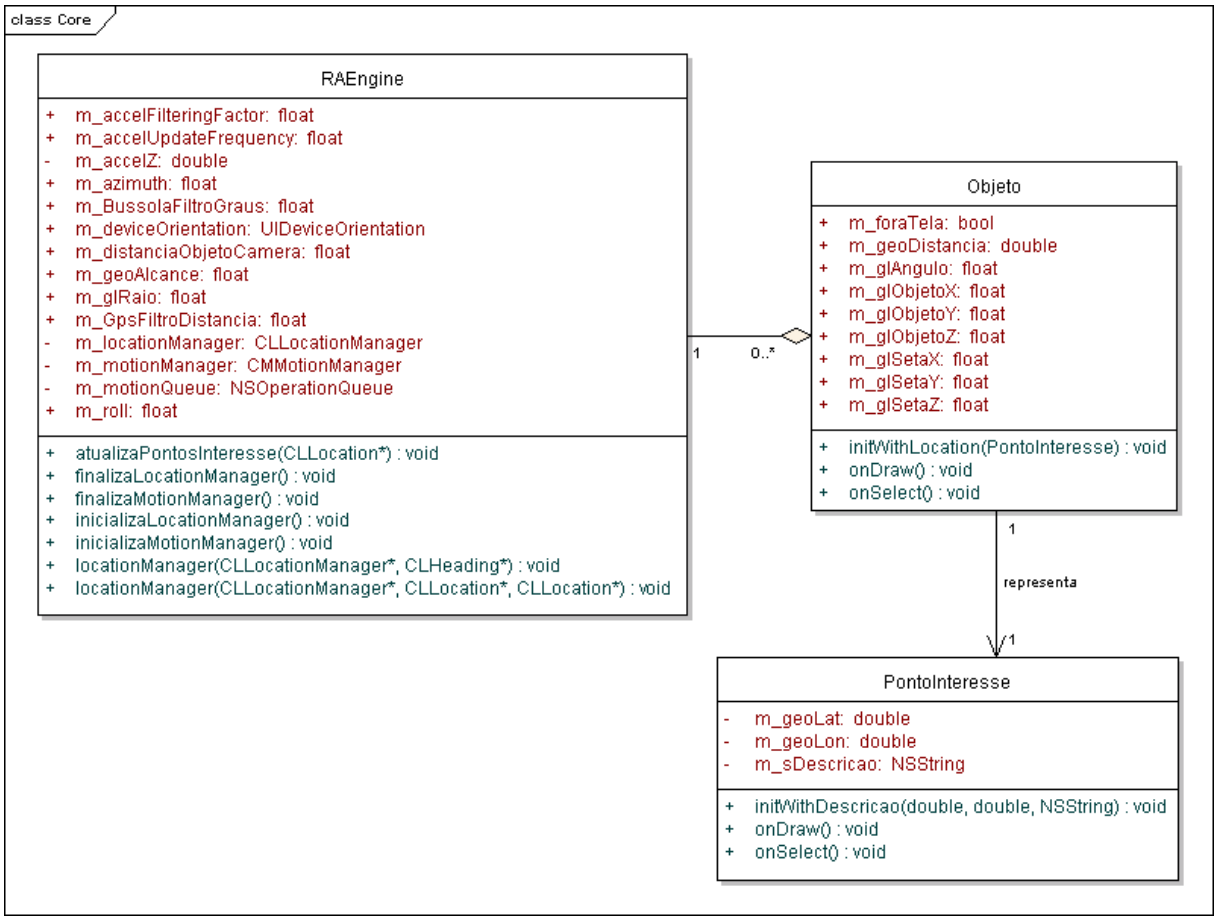


Figura 8 – Pacote `core`

A classe `Objeto` guarda todas as informações relativas a um objeto virtual a ser desenhado pelo motor de renderização OpenGL ES. O atributo `m_geoDistancia` é calculado pela `RAEngine` e representa a distância, em metros, do ponto de interesse até o dispositivo. São calculados também os atributos `m_glAngulo`, `m_glSetaX`, `m_glSetaY` e `m_glSetaZ` que determinam a posição da seta que aponta para o ponto de interesse e os atributos `m_glObjetoX`, `m_glObjetoY` e `m_glObjetoZ` responsáveis por indicar um ponto 3D onde o objeto virtual será desenhado. O atributo `m_azimuth` armazena o valor do norte verdadeiro da bússola e é utilizado como ângulo de rotação no eixo Y quando do desenho dos objetos virtuais. Por último, o atributo `m_accelz` guarda o valor da aceleração no eixo Z do

dispositivo.

A classe `PontoInteresse` representa a localização do objeto virtual no mundo real. Os atributos `m_geoLat` e `m_geoLon` são preenchidos com os valores da coordenada geográfica que se encontra o ponto de interesse. Os métodos `onDraw` e `onSelect` desta classe são chamados pelos métodos de mesma nomenclatura da classe `Objeto` e descrevem como o objeto virtual será desenhado pelo motor gráfico e o conjunto de instruções que devem ser executadas quando o mesmo é tocado na tela do dispositivo, respectivamente. Estes métodos possibilitam ao desenvolvedor que utilizar a RAios-library herdar a classe `PontoInteresse` e especificar suas próprias rotinas de desenho e comportamento do objeto virtual dentro da aplicação.

3.2.3.2 Pacote `opengles`

As classes do pacote `opengles` são responsáveis por renderizar a camada de OpenGL ES da RAios-library e estão representadas na Figura 9.

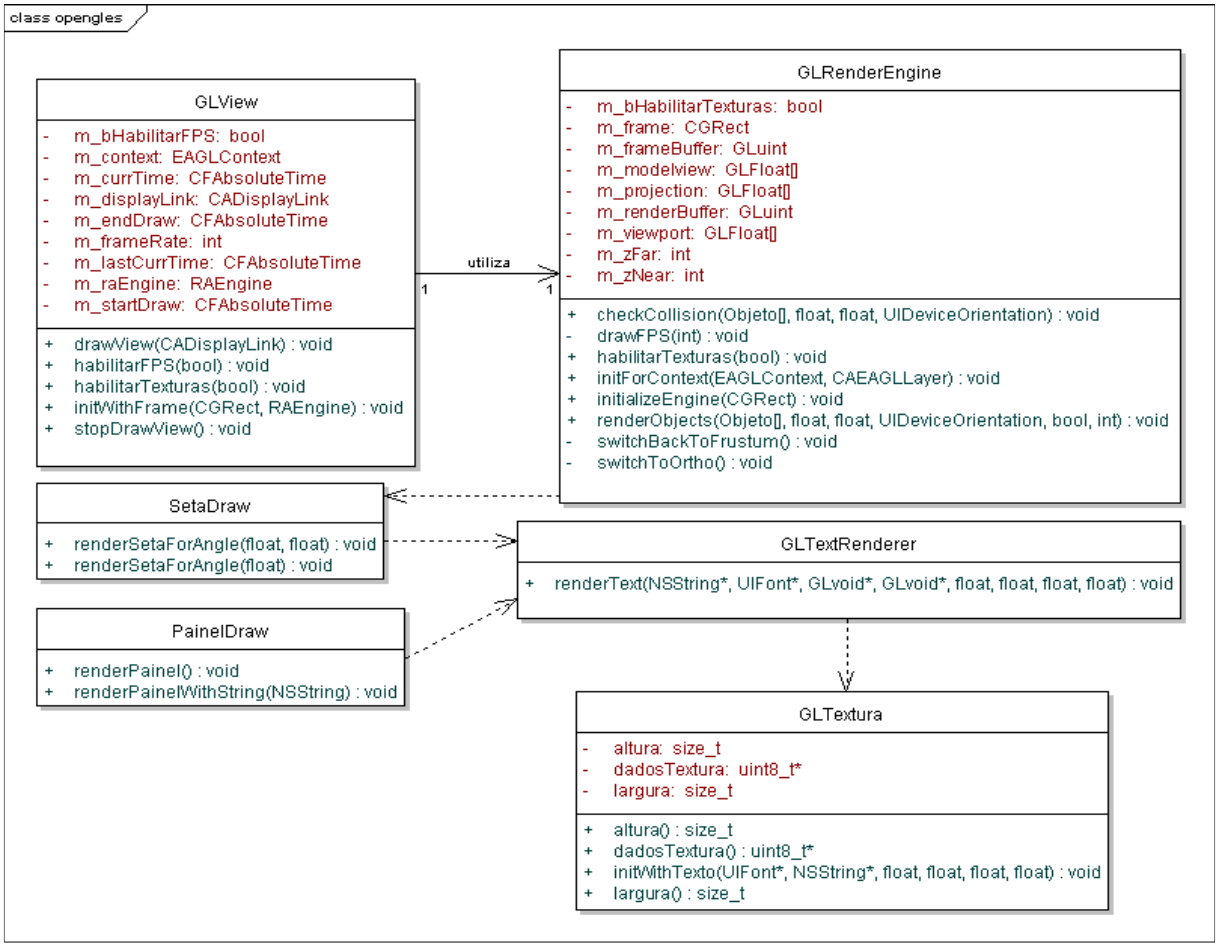


Figura 9 – Pacote `opengles`

Toda renderização no iOS deve ocorrer dentro de uma `view` e, para tanto, foi criada a classe `GLView`. Essa classe é uma extensão da classe do iOS chamada `UIView` que define uma área retangular na tela e as interfaces para gerenciar o conteúdo desta área. O atributo `m_context` é um objeto da classe `EAGLContext`, que é a implementação do iOS para um contexto de renderização de acordo com a especificação do OpenGL ES. Este contexto deve ser inicializado e definido como o contexto atual antes que a aplicação possa efetuar chamadas às funções do OpenGL ES.

A classe `GLView` também controla o *loop* de desenho da aplicação através do atributo `m_displayLink`, um objeto da classe `CADisplayLink` do iOS. Este objeto age como um cronômetro e permite sincronizar a rotina de desenho OpenGL ES contida no método `drawView` com a taxa de atualização da tela do dispositivo. O fim do *loop* de desenho da aplicação é determinado pelo método `stopDrawView`.

A classe `GLRenderEngine` tem por objetivo centralizar todas as chamadas às funções do OpenGL ES. O método `initForContext` inicializa a classe e aloca o espaço de armazenamento necessário para o `frame buffer`. Já o método `initializeEngine` recebe as dimensões da tela, prepara a `viewport`, carrega a matriz de projeção, carrega a matriz de modelos, prepara o `frustum` e configura as variáveis de estado do OpenGL ES que não mudarão durante a execução do aplicativo. O método `renderObjects` é chamado quando é requisitada a renderização, sendo sua principal responsabilidade realizar todas as transformações e desenhar os objetos virtuais de cada ponto de interesse no seu devido lugar da tela. Os métodos `switchToOrtho` e `switchBackToFrustum` permitem alternar entre os modos de projeção em perspectiva e ortogonal. Por último o método `checkCollision` é utilizado para verificar se um toque na tela colide com um objeto virtual.

A classe `SetaDraw` é responsável por desenhar uma seta com o texto da distância entre o ponto de interesse e o dispositivo. A Seta é desenhada através de um conjunto de triângulos definidos em um vetor de vértices criado de forma estática na definição da classe. O texto da distância é desenhado utilizando os recursos da classe `GLTextRenderer`.

A classe `PainelDraw` serve como implementação padrão do método `onDraw` da classe `PontoInteresse` e possui a responsabilidade de desenhar um painel com o texto do nome do ponto de interesse representado. Assim como a seta, o painel também é desenhado utilizando um conjunto de triângulos definidos em um vetor de vértices e o texto através dos recursos da classe `GLTextRenderer`.

Por último, devido a limitação da especificação do OpenGL ES 1.1 em não possuir

recursos para a renderização de texto, foi necessário pesquisar implementações e estratégias existentes para atender a essa necessidade. As classes `GLTExtura` e `GLTextRenderer` foram desenvolvidas baseando-se na solução apresentada na referência STACK EXCHANGE INC (2011), utilizando recursos do *framework Core Graphics* para criar uma textura contendo o texto e então renderizar esta textura utilizando funções do OpenGL ES.

3.2.3.3 Pacote `camera`

A classe do pacote `camera` é responsável pela exibição das imagens capturadas pela câmera do dispositivo e está representada na Figura 10.

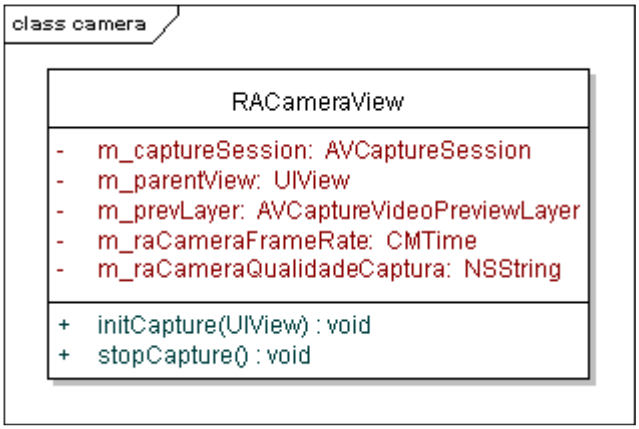


Figura 10 – Pacote `camera`

A classe `RACameraView` é responsável pela aquisição e exibição das imagens da câmera do dispositivo em tempo real. Para tanto, são utilizados recursos do *framework* AV *Foundation* disponibilizado pelo iOS. O método `initCapture` é responsável por iniciar a exibição do vídeo. Esse método recebe como parâmetro o atributo `m_parentView`, que será a `view` onde as imagens capturadas serão projetadas. O atributo `m_captureSession` é um objeto da classe `AVCaptureSession` que é utilizado para criar uma seção de captura de vídeo. Este objeto coordena o fluxo de dados entre a câmera do dispositivo e um meio de saída, que neste caso é o atributo `m_prevLayer`, uma instância da classe `AVCaptureVideoPreviewLayer` que permite exibir em uma `view` os *frames* que a câmera está capturando.

Essa classe ainda permite configurar a taxa máxima de captura de *frames* da câmera através do atributo `m_raCameraFrameRate` e a qualidade (taxa de bits) da imagem capturada através do atributo `m_raCameraQualidadeCaptura`.

3.2.3.4 Pacote `ra`

O pacote `ra` possui apenas a classe `RAios`. Essa classe é responsável por fornecer uma interface simplificada de acesso às funcionalidades da RAios-library. Seus atributos e métodos são demonstrados no diagrama da Figura 11.

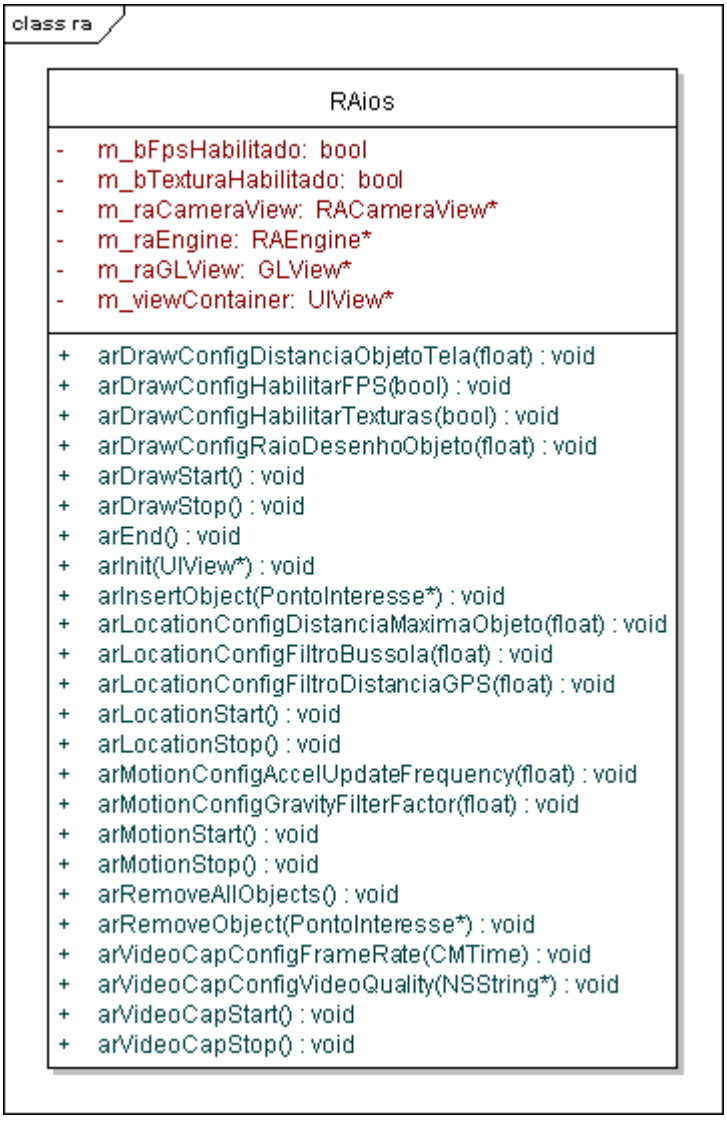


Figura 11 – Pacote `ra`

O método `arInit` configura o atributo `m_viewContainer`, que é a `view` onde as camadas da aplicação de RA serão exibidas e aloca memória para alguns componentes da RAios-library. Os métodos `arVideoCapStart`, `arVideoCapStop`, `arVideoCapConfigFrameRate` e `arVideoCapConfigVideoQuality` permitem a manipulação dos recursos de vídeo da biblioteca. A detecção de movimentos do dispositivo é iniciada utilizando o método `arMotionStart` e pode ser desativada utilizando o método `arMotionStop`. Os métodos `arLocationStart` e `arLocationStop` iniciam e encerram o

serviço de localização, respectivamente. O método `arInsertObject` permite inserir os pontos de interesse que serão exibidos pela aplicação. Os métodos `arDrawStart` e `arDrawStop` devem ser chamados para iniciar e interromper o desenho dos pontos de interesse. Por último, o método `arEnd` encerra os serviços da RAios-library que foram inicializados e libera a memória alocada.

3.2.4 Diagrama de seqüência

O diagrama de seqüência da Figura 12 apresenta a interação do Desenvolvedor com a biblioteca RAios-library no desenho dos objetos virtuais do caso de uso `UC05`.

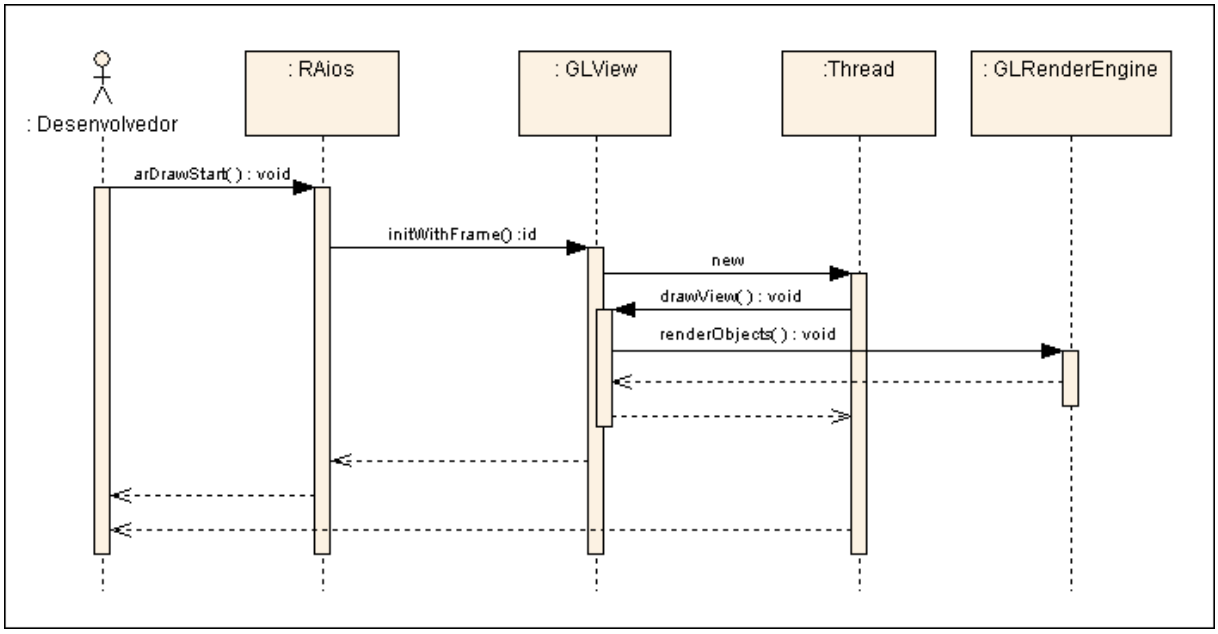


Figura 12 – Diagrama de seqüência, apresentando o desenho dos objetos virtuais

3.3 IMPLEMENTAÇÃO

A seguir são descritas as técnicas e ferramentas utilizadas para o desenvolvimento da biblioteca proposta, junto com uma descrição da sua operacionalidade e trechos de código para um melhor entendimento.

3.3.1 Técnicas e ferramentas utilizadas

Para a implementação da RAios-library foi utilizada a linguagem de programação *Objective*-C, que é uma linguagem derivada do C e acrescida de algumas capacidades do *Smalltalk*. O ambiente de desenvolvimento utilizado foi o Xcode, disponibilizado pela Apple no suíte de ferramentas de desenvolvimento com o mesmo nome e a geração de imagens gráficas no espaço 3D foi implementada utilizando a API OpenGL ES, desenvolvida para sistemas embarcados.

3.3.2 Testes e depuração

Devido às limitações apresentadas pelo simulador oficial disponível no Xcode como a ausência de simulação dos sensores do acelerômetro, GPS e câmera, foram pesquisadas outras alternativas para realizar a depuração da RAios-library.

Uma das alternativas encontradas foi o software *Accelerometer Simulator* (CHRONS, 2010). Esta ferramenta consiste em duas partes: o cliente, que é executado no dispositivo e a biblioteca, que são dois arquivos a serem vinculados ao projeto no Xcode. Este software permite a simulação do sensor de movimento transferindo os dados do acelerômetro do dispositivo para o computador através do protocolo UDP.

Outra ferramenta encontrada foi o iSimulator (VIMOV, 2011). A exemplo do *Accelerometer Simulator*, esta solução também possui um cliente que deve ser instalado no dispositivo e uma biblioteca que deve ser ligada do projeto no Xcode, porém mostra-se mais completa, pois além de simular os dados do acelerômetro, possui outros recursos como a simulação do serviço de localização, *streaming* de vídeo e multitoque.

Apesar das alternativas encontradas, optou-se por efetuar a depuração e a execução da RAios-library utilizando o dispositivo *smartphone* da fabricante Apple chamado iPhone 4, que possui o sistema operacional iOS e permite testar a biblioteca nas reais condições de uso.

3.3.3 A *engine*

Como o próprio nome sugere, a classe `RAEngine` é o motor que impulsiona o

funcionamento da RAios-library. Essa classe possui a responsabilidade de efetuar o alinhamento entre os objetos virtuais e a cena real e, para tanto, deve manter atualizados os pontos de interesse em relação à localização e orientação do dispositivo.

Inicialmente, um objeto da classe `RAEngine` não executa nenhuma tarefa, apenas inicializa e aloca memória para seus atributos. Sendo assim, todos os recursos disponibilizados pela *engine* devem ser inicializados pelo seu usuário, permitindo ao mesmo utilizar apenas as funções que julgar necessárias para atender às necessidades de sua aplicação.

O Quadro 6 apresenta o código do método `inicializaLocationManager` que é responsável por iniciar o serviço de notificação de mudanças nas coordenadas geográficas e no sensor de orientação.

Primeiramente, o objeto `m_locationManager` é instanciado. Esse atributo é utilizado pela classe `RAEngine` para interagir com o serviço de localização geográfica disponibilizado pela API *Core Location* do iOS. Para que a *engine* possa iniciar a captação da localização do dispositivo, ela deve ser atribuída como o *delegate* de `m_locationManager` e estar em conformidade com o protocolo `CLLocationManagerDelegate`.

Em seguida, é definido o grau de precisão desejado. Conforme Mark, Nutting e Lamarche (2011, p.557), o *Core Location* dispõe de três tecnologias para determinar a localização do dispositivo: GPS, triangulação entre torres de telefonia móvel e Wi-Fi *Positioning Service* (WPS). Neste caso, o uso da constante `kCLLocationAccuracyBest` indica ao *Core Location* para utilizar o método de localização de melhor precisão disponível. Na linha seguinte, é configurada a distância em metros do deslocamento lateral a ser percorrido pelo dispositivo para que seja gerado um evento de atualização de localização. Logo após, a mensagem `startUpdatingLocation` informa ao *Core Location* para iniciar a captura da localização do dispositivo.

Após a configuração do serviço de localização geográfica, é configurado e inicializado o sensor de direção. A mensagem `headingAvailable` é utilizada para verificar se o dispositivo possui um sensor disponível. Em caso afirmativo, é configurado valor da mudança angular mínima para que sejam gerados novos eventos de atualização da direção do dispositivo. Uma vez configurado o sensor, a mensagem `startUpdatingHeading` é enviada para iniciar a captura dos eventos de mudança de direção do dispositivo.

```
- (void)inicializaLocationManager{
    // inicializa a instância do location manager
    m_locationManager = [[CLLocationManager alloc]init];

    // seta o delegate
    m_locationManager.delegate = self;

    // ajusta a precisão desejada
    m_locationManager.desiredAccuracy = kCLLocationAccuracyBest;

    // distância em metros de deslocamento lateral para atualizar a localização.
    m_locationManager.distanceFilter = m_GpsFiltroDistancia;

    // inicia a atualização da localização
    [m_locationManager startUpdatingLocation];

    // verifica se o recurso da bússola está disponível
    if ([CLLocationManager headingAvailable]) {

        // mundança angular mínima para gerar novos eventos da bússola
        m_locationManager.headingFilter = m_BussolaFiltroGraus;

        // inicia a atualização da bússola
        [m_locationManager startUpdatingHeading];
    }
    else{
        // caso o sensor de orientação não esteja disponível
        NSLog(@"Bússola não disponível");
    }
}
```

Quadro 6 – Código fonte do método `inicializaLocationManager`

Ao receber a mensagem de que houve uma mudança na localização geográfica do dispositivo (Quadro 7), a *engine* deve realinhar os pontos de interesse com a nova coordenada geográfica.

```
- (void) locationManager:(CLLocationManager *)manager didUpdateToLocation:(CLLocation *)
newLocation fromLocation:(CLLocation *)oldLocation {

    // realinha os pontos de interesse com a nova localização
    [self atualizaPontosInteresse:newLocation];
}
```

Quadro 7 – Realinhar pontos de interesse

O método `atualizaPontosInteresse` é baseado na lógica de rastreamento apresentada por Vasselai (2010) e realiza todos os cálculos de posicionamento, angulação e validação dos pontos de interesse. Para facilitar o seu entendimento, a implementação foi selecionada e dividida em alguns quadros.

O Quadro 8 mostra parte do código principal do método e um algoritmo para facilitar o entendimento do seu comportamento. Os passos do algoritmo estão descritos nos parágrafos seguintes.

```
- (void)atualizaPontosInteresse:(CLLocation *)localizacaoDispositivo{

    float glDiametroCamera = m_glRaio * 2;

    CLLocation *poiLocation;
    for(Objeto *obj in m_arrayObjetos){

        // Calcula a distância do ponto de interesse para o dispositivo
        // Verifica se o ponto de interesse está visível na tela do dispositivo
        // Calcula a localização e o ângulo do ponto de interesse no OpenGL ES
        // Calcula a localização do objeto virtual no OpenGL ES
    }
}
```

Quadro 8 – Algoritmo de atualização dos pontos de interesse

O Quadro 9 mostra como é calculada a distância entre o ponto de interesse e o dispositivo. Para tanto, está sendo utilizado o método `distanceFromLocation` da classe `CLLocation` presente no *framework Core Location*. Segundo APPLE INC (2010g), este método mede a distância entre dois pontos na superfície da Terra, traçando uma linha entre eles que segue a curvatura do globo não levando em conta as alterações específicas de altitude entre os dois pontos. O valor da distância é então armazenado no atributo `m_geoDistancia` no objeto do ponto de interesse.

```
// Instancia um objeto da classe CLLocation com as coordenadas do ponto de interesse
poiLocation = [[CLLocation alloc]initWithLatitude:obj->m_pontoInteresse.geoLat
longitude:obj->m_pontoInteresse.geoLon];

// Calcula a distância entre o ponto de interesse e o dispositivo
CLLocationDistance distance = [localizacaoDispositivo distanceFromLocation:poiLocation];
obj->m_geoDistancia = distance;

// Libera a memória alocada para poiLocation
[poiLocation release];

// Guarda se o POI está muito longe e, portanto, fora da tela do dispositivo
obj->m_foraTela = abs(obj->m_geoDistancia) > m_geoAlcance;

// (...)
```

Quadro 9 – Obtendo a distância entre um ponto de interesse e o dispositivo

A posição da seta e a posição do objeto virtual do ponto de interesse são calculadas a partir do seno e do cosseno do ângulo que há entre o ponto de interesse e o dispositivo. Esses valores são armazenados nos atributos `m_glSetaX`, `m_glSetaY`, `m_glObjetoX` e `m_glObjetoY` do objeto do ponto de interesse. O valor do anglo também é armazenado no atributo `m_glAngulo` e é utilizado para rotacionar a seta e posicionar o objeto virtual do ponto de interesse de forma equivalente, conforme demonstrado no Quadro 10.

```
// Guarda a posição da seta no OpenGL ES.
obj->m_glSetaX = (float) (glDiametroCamera * anguloSin);
obj->m_glSetaY = (float) (glDiametroCamera * anguloCos);

// Guarda a posção do objeto virtual no OpenGL ES.
obj->m_glObjetoX = (float) (m_distanciaObjetoCamera * anguloSin);
obj->m_glObjetoY = (float) (m_distanciaObjetoCamera * anguloCos);

// Guarda o ângulo pois vai usar na seta e no painel.
obj->m_glAngulo = rAngulo * 180 / M_PI;
```

Quadro 10 – Calcular a posição e o ângulo da seta e do objeto virtual do ponto de interesse

Quando ocorre alguma mudança no sensor de orientação (Quadro 11), a *engine* recebe

uma instância da classe `CLHeading`, que contém os dados sobre a direção do dispositivo. Esses dados consistem em valores da bússola computados para o norte verdadeiro e magnético. A propriedade `headingAccuracy` é utilizada para verificar a validade da informação recebida. Um valor negativo indica que os dados recebidos são inválidos, o que ocorre quando o dispositivo está descalibrado ou em um local com forte interferência de campos magnéticos (APPLE INC, 2010g). A nova orientação é então armazenada no atributo `m_azimuth` para uso posterior pelo motor de renderização gráfica.

```
- (void)locationManager:(CLLocationManager *)manager didUpdateHeading:(CLHeading *)
newHeading {

    // A atualização do azimuth somente ocorre quando há mudança angular
    // maior que o valor contido em m_locationManager.headingFilter

    // Verifica se o heading é válido. Valor negativo significa erro
    if( [newHeading headingAccuracy] < 0.00 ){
        NSLog(@"Leitura da bussola invalida!");
        return;
    }

    // obtém o norte verdadeiro
    m_azimuth = [newHeading trueHeading];
}
```

Quadro 11 – Guardar nova direção

O último método relacionado ao serviço de localização é demonstrado no Quadro 12 e tem a responsabilidade de liberar os recursos de localização alocados para a aplicação.

```
- (void)finalizaLocationManager{
    // Encerra o polling de eventos da bússola
    [m_locationManager stopUpdatingHeading];

    // Encerra o polling de eventos de localização
    [m_locationManager stopUpdatingLocation];

    // Libera a memória alocada para o location manager
    [m_locationManager release];
    m_locationManager = nil;
}
```

Quadro 12 – Encerrar monitoração de localização

O Quadro 13 apresenta o código do método `inicializaMotionManager` que é responsável por iniciar o serviço de notificação de mudanças no acelerômetro do dispositivo. Nesse método, optou-se por utilizar uma mecânica diferente da utilizada pelo serviço de localização. Graças a um conceito denominado *block*, foi possível agrupar todo o código relacionado a leitura do sensor neste único método, invés de espalhar partes da funcionalidade em métodos delegados.

```
- (void)inicializaMotionManager{
    m_accelZ = 0.00;

    m_motionManager = [[CMMotionManager alloc]init];
    m_motionQueue = [[NSOperationQueue alloc]init];

    if( m_motionManager.accelerometerAvailable ){
        // frequencia de atualização do acelerômetro
        m_motionManager.accelerometerUpdateInterval = 1.0/m_accelUpdateFrequency;

        // inicializa a leitura dos valores do acelerômetro
        [m_motionManager startAccelerometerUpdatesToQueue:m_motionQueue
            withHandler:^(CMAccelerometerData *accelerometerData, NSError *error)
         {

            // Guardar orientação do dispositivo
            // Calcular ângulo de inclinação do objeto virtual

         }];// fim do bloco de leitura do acelerômetro
    }
    else{
        NSLog(@"Acelerômetro não disponível");
    }
}
```

Quadro 13 – Código fonte do método `inicializaMotionManager`

Inicialmente, o objeto `m_motionManager` é instanciado. Esse atributo é utilizado pela classe `RAEngine` para interagir com o serviço de movimento disponibilizado pela API *Core Motion* do iOS. Também é instanciado o atributo `m_motionQueue`, um objeto da classe `NSOperationQueue` utilizado para enfileirar as notificações de atualização do acelerômetro recebidas. Em seguida, é verificado se o sensor do acelerômetro está disponível e configurada a taxa de atualização, intervalo em segundos, desejada entre as atualizações de leitura.

A mensagem `startAccelerometerUpdatesToQueue:withHandler` informa ao *Motion Manager* para iniciar a captura das informações do acelerômetro e juntamente com esta mensagem é passada a fila que irá armazenar os dados lidos do sensor e o bloco que define o código que deve ser executado sempre que uma atualização ocorre.

O código do bloco (Quadro 14) é responsável por guardar a orientação atual do dispositivo no atributo `m_deviceOrientation` e calcular o ângulo de inclinação do dispositivo com base no valor da gravidade no atributo `m_roll`. Segundo Mark, Nutting e Lamarche (2011, p.581), os valores retornados pelo acelerômetro variam de `-2.3f` a `+2.3f`, quando a movimentação ocorre em um ângulo de 180º, portanto, a cada 90º de movimentação do dispositivo (seja ela no eixo X, Y ou Z), o valor retornado pelo acelerômetro será entre zero e o valor da gravidade (positivo ou negativo). Para que a animação ocorra de forma mais suave o valor retornado pelo acelerômetro é filtrado, separando a parte da aceleração que é causada pela gravidade, da parte causada pelo movimento do dispositivo conforme exemplo disponível na referência APPLE INC (2011b).

```
if( error){
    // Caso ocorra erro encerra as atualizações do acelerêmetro
    [m_motionManager stopAccelerometerUpdates];

    NSLog(@"O acelerômetro encontrou um erro: %@", error);
}
else{
    m_deviceOrientation = [[UIDevice currentDevice]orientation];

    // Filtra a parte da aceleração causada pela gravidade da parte da aceleração causada
    // pelo movimento do dispositivo
    m_accelZ = (accelerometerData.acceleration.z * m_accelFilteringFactor) +
               (m_accelZ * (1.0 - m_accelFilteringFactor));

    // obtem a orientação real no x.
    float newAcelerometroZ = (90.0/2.3) * m_accelZ + 90.0;

    m_roll = newAcelerometroZ;
}
```

Quadro 14 – Calcular ângulo de inclinação

Conforme APPLE INC (2011b), a captura de movimentos do acelerômetro deve ser interrompida assim que a aplicação não necessitar mais dos dados do sensor, permitindo que o Core Motion desligue os sensores de movimento e gere economia de bateria do dispositivo. O método responsável por encerrar a captura de movimento é mostrado no Quadro 15.

```
- (void)finalizaMotionManager{

    // Encerra o monitoramento de movimento
    [m_motionManager stopAccelerometerUpdates];

    // Cancela todas as operações pendentes na fila
    [m_motionQueue cancelAllOperations];

    // Espera o encerramento das operações em andamento
    [m_motionQueue waitUntilAllOperationsAreFinished];

    // Libera a memória alocada para m_motionQueue
    [m_motionQueue release];
    m_motionQueue = nil;

    // Libera a memória alocada para m_motionManager
    [m_motionManager release];
    m_motionManager = nil;

    m_roll = 0.0;
}
```

Quadro 15 – Encerrar monitoração de movimento

Quando a aplicação é finalizada, o método `dealloc` (Quadro 16) é chamado pela classe `RAios`. Este método encerra os serviços de localização e movimento (caso estejam ativos) e libera a memória alocada para a lista de pontos de interesse.

```
- (void)dealloc
{
    // libera a memória alocada para o location manager
    [self finalizaLocationManager];

    // libera a memória alocada para o motion manager
    [self finalizaMotionManager];

    // libera a memória alocada para os pontos de interesse
    [m_arrayObjetos release];

    [super dealloc];
}
```

Quadro 16 – Finalizando a *engine*

3.3.4 A cena real

Um dos principais requisitos de uma aplicação de realidade aumentada é permitir que o usuário visualize a cena real através do dispositivo. O componente da RAios-library que fornece tal funcionalidade é a classe `RACameraView`. Essa classe foi desenvolvida utilizando os recursos do *framework AV Foundation* disponibilizado pelo iOS e permite capturar os dados provenientes da câmera do dispositivo e exibi-los em uma `view` dentro da aplicação.

O método `initCapture` é responsável por iniciar a exibição dos dados da câmera do dispositivo na `view` `m_parentView`. O Quadro 17 mostra parte do código principal do método e um algoritmo para facilitar o entendimento do seu comportamento. Os passos do algoritmo estão descritos nos parágrafos seguintes.

```
- (void)initCapture:(UIView*)parentView{

    m_parentView = parentView;

    // configurar dispositivo de captura de vídeo
    // configurar a saída de vídeo
    // inicializar seção de captura
    // criar e adicionar o layer responsável pela exibição de vídeo em m_parentView

    // inicia a captura de vídeo
    [m_captureSession startRunning];
}
```

Quadro 17 – Algoritmo de exibição dos dados da câmera

Para gerenciar a captura de vídeo do dispositivo, são utilizados objetos que representam entradas (`AVCaptureInput`), objetos que representam saídas (`AVCaptureOutput`) e uma instância de `AVCaptureSession` para coordenar o fluxo da dados entre eles (APPLE INC, 2010h).

O Quadro 18 mostra os passos para configuração do dispositivo de entrada.

```
// Configura a captura de dados da câmera
AVCaptureDeviceInput *captureInput = [AVCaptureDeviceInput deviceInputWithDevice:
    [AVCaptureDevice defaultDeviceWithMediaType:AVMediaTypeVideo] error:nil];

if(!captureInput){
    NSLog(@"Erro ao criar o dispositivo de captura de video.");
    return;
}

// (...)
```

Quadro 18 – Configurar captura de vídeo

Os passos para a configuração do dispositivo de saída são mostrados no Quadro 19. A propriedade `alwaysDiscardsLateVideoFrames` é atribuída como `YES` para evitar que outros *frames* sejam adicionados na fila de exibição enquanto um frame está sendo processado. A duração mínima, em segundos, para cada *frame* é especificada na propriedade `minFrameDuration`, sendo que o `framerate` máximo é de 60 Frames Por Segundo (FPS). Por

último, o dispositivo de saída é configurado para armazenar o frame capturado no espaço de cores BGRA.

```
// (...)

// cria dispositivo de saída
AVCaptureVideoDataOutput *captureOutput = [[AVCaptureVideoDataOutput alloc]init];

// verifica se conseguiu criar
if(!captureOutput){
    NSLog(@"Erro ao criar o dispositivo de saída de video.");
    return;
}

// descarta frames atrasados
captureOutput.alwaysDiscardsLateVideoFrames = YES;

// duração mínima para cada frame
captureOutput.minFrameDuration = m_raCameraFrameRate;

// Configura a saída para BGRA
NSString *key = (NSString*)kCVPixelBufferPixelFormatTypeKey;
NSNumber *value = [NSNumber numberWithUnsignedInt:kCVPixelFormatType_32BGRA];
NSDictionary *videoSettings = [NSDictionary dictionaryWithObject:value forKey:key];
[captureOutput setVideoSettings:videoSettings];

// (...)
```

Quadro 19 – Configurar saída de vídeo

O Quadro 20 mostra a criação e configuração da seção de captura. A mensagem `setSessionPreset` é utilizada para configurar o `bitrate` (nível de qualidade) da saída de vídeo. Em seguida, é verificado se a seção permite adicionar os dispositivos de entrada e saída e os adiciona em caso afirmativo. Por último, a memória que não se faz mais necessária é liberada.

```
// (...)

// inicializa a seção de captura
m_captureSession = [[AVCaptureSession alloc]init];

// configura a qualidade da captura
[m_captureSession setSessionPreset:m_raCameraQualidadeCaptura];

// adiciona a entrada e saída
if([m_captureSession canAddInput:captureInput])
    [m_captureSession addInput:captureInput];

if([m_captureSession canAddOutput:captureOutput])
    [m_captureSession addOutput:captureOutput];

// libera a memória alocada
[captureOutput release];

// (...)
```

Quadro 20 – Criar seção de captura

Para exibir o vídeo capturado na tela do dispositivo, é utilizada uma instância de `AVCaptureVideoPreviewLayer`, que é uma subclasse de `CALayer`. Este `layer` é vinculado ao atributo `m_captureSession` durante a sua instanciação, fazendo com que os dados do dispositivo de saída sejam apresentados nele. Por fim, o `layer` é configurado com as mesmas dimensões de `m_parentView` e adicionado como uma subcamada desta `view` para tornar-se visível na aplicação. A criação do `layer` de exibição do vídeo é demonstrada no Quadro 21.

```
// (...)

// Adiciona o layer responsável pela exibição do vídeo
m_prevLayer = [AVCaptureVideoPreviewLayer layerWithSession:m_captureSession];
m_prevLayer.frame = m_parentView.bounds;
m_prevLayer.videoGravity = AVLayerVideoGravityResizeAspectFill;
[m_parentView.layer addSublayer:m_prevLayer];

// (...)
```

Quadro 21 – Exibir dados capturados pela câmera

O método que encerra a exibição do vídeo chama-se `stopCapture` e é demonstrado no Quadro 22. Esse método finaliza a seção de captura, remove o `layer` de exibição do vídeo da `view` da aplicação e libera a memória alocada.

```
- (void)stopCapture{

    // encerra a seção de captura
    if(m_captureSession){
        if([m_captureSession isRunning])
            [m_captureSession stopRunning];

        [m_captureSession release];
        m_captureSession = nil;
    }

    // remove o layer da view superior
    if(m_prevLayer){
        [m_prevLayer removeFromSuperlayer]; // esta mensagem decrementa o retain count
        m_prevLayer = nil;
    }
}
```

Quadro 22 – Finalizar exibição da câmera

3.3.5 A realidade aumentada

O aumento da realidade através da adição de informação sintética à cena real é implementado pela classe `GLView`. Através dessa classe é criada uma `view` onde os objetos virtuais serão renderizados utilizando uma instância da classe `GLRenderEngine`, que é o motor gráfico da biblioteca. A RAios-library fornece uma implementação padrão para o formato dos objetos virtuais através da classe `PainelDraw`, que pode ser substituída posteriormente por código do desenvolvedor. A implementação dessas classes é descrita nas próximas seções.

3.3.5.1 A classe `GLView`

Para exibir o conteúdo renderizado pelo OpenGL ES em uma `view`, se faz necessário que a mesma esteja em concordância com a versão criada pela Apple para a especificação do

OpenGL ES. Para tanto, é necessário seguir dois passos que são descritos nos próximos parágrafos.

O primeiro passo (Quadro 23) é sobrescrever o método `layerClass` da classe `UIView` para retornar um `CAEAGLLayer`. No iOS, todas as imagens exibidas na tela são manipuladas pelo *framework Core Animation* e, sendo assim, toda `view` possui um `layer` *Core Animation* correspondente. A classe `CAEAGLLayer` age como um tipo especial de `layer` que conecta o conteúdo do OpenGL ES ao *Core Animation*.

```
// Para aplicações OpenGL, este método deve ser sempre sobreescrito
+ (Class)layerClass{
    return [CAEAGLLayer class];
}
```

Quadro 23 – Método `layerClass` sobrescrito

O segundo passo (Quadro 24) é criar um contexto de renderização onde os comandos do OpenGL ES serão executados. Esse contexto é definido pela classe `EAGLContext` e contém todas as informações necessárias para renderizar uma cena criada no OpenGL ES.

Para criar uma instância de `EAGLContext` é necessário especificar a versão do OpenGL ES que será utilizada. Neste caso, a constante `kEAGLRenderingAPIOpenGLES1` indica que será criado um contexto de renderização para a especificação do OpenGL ES 1.1. Depois de criado, o contexto deve ser definido como o contexto de renderização atual através da mensagem `setCurrentContext`, indicando que qualquer chamada ao OpenGL ES na *thread* atual será amarrada a este contexto.

```
// cria o contexto de renderização e especifica qual versão do OpenGl vai ser utilizada
m_context = [[EAGLContext alloc]initWithAPI:kEAGLRenderingAPIOpenGLES1];

// seta o contexto EAGL como corrente.
[EAGLContext setCurrentContext:m_context];
```

Quadro 24 – Criando um contexto de renderização

O Quadro 25 mostra a implementação do construtor da classe `GLView`, aonde além de configurar a `view` para ser translúcida e criar o contexto de renderização, é instanciado o motor de desenho e iniciada a renderização dos objetos virtuais através de um *loop* de animação.

```
- (id)initWithFrame:(CGRect)frame resourceProvider:(RAEngine*)engine
{
    self = [super initWithFrame:frame];
    if (self) {

        // (...)

        // atribui a propriedade layer da classe UIView para eaglLayer isto é possivel por
           causa o override do método layerClass
        CAEAGLLayer* eaglLayer = (CAEAGLLayer*) super.layer;

        // indica que a view será translúcida
        eaglLayer.opaque = NO;
        setBackgroundColor:[UIColor clearColor];

        // (...)

        // instancia do motor de renderização
        m_glRenderEngine = [[GLRenderEngine alloc] initForContext:m_context
                                                   fromDrawable:eaglLayer];

        // inicializa a engine de renderização com o tamanho da tela do dispositivo
        if( ! [m_glRenderEngine initializeEngine:frame]){
            NSLog(@"Erro ao inicializar o motor grafico.");
            [self release];
            return nil;
        }

        // cria o loop de animação
        m_displayLink = [CADisplayLink displayLinkWithTarget:self
                                        selector:@selector(drawView:)];
        m_displayLink.frameInterval = 1;
        [m_displayLink addToRunLoop:[NSRunLoop currentRunLoop] forMode:NSDefaultRunLoopMode];
    }
    return self;
}
```

Quadro 25 – Construtor da classe `GLView`

Dentre as várias formas disponíveis para desenhar os objetos virtuais, optou-se por utilizar um *loop* de animação que, segundo APPLE INC (2011c), é o modelo de renderização apropriado quando os dados a serem desenhados podem mudar a cada `frame`. Neste caso, uma animação suave é mais importante que um modelo de renderização sob demanda. Sendo assim, a escolha pelo *loop* de animação justifica-se pelo fato de o dispositivo estar sendo movimentado a todo instante, fazendo com que imagens estáticas raramente sejam apresentadas.

Conforme APLLE INC (2011c), a melhor maneira de criar um *loop* de animação é através de um objeto da classe `CADisplayLink`. Um *display link* é um objeto de *Core Animation* que sincroniza o desenho com a taxa de atualização da tela do dispositivo. Isto permite uma atualização suave para o conteúdo da tela sem a impressão que os objetos estão tremendo.

Apesar de ser a melhor opção, a utilização de `CADisplayLink` traz consigo uma desvantagem que deve ser contornada pela aplicação. O *display link* é disparado cada vez que a tela do dispositivo é atualizada. No iPhone 4 (dispositivo utilizado para testes), a taxa de atualização da tela é de 60 *hertz* (Hz) o que significa que a tela é atualizada 60 vezes por segundo. Sempre que o *display link* é disparado, o mesmo efetua uma chamada ao método

`drawView` que é responsável por desenhar a cena renderizada pelo motor gráfico. Como o tempo para renderizar uma cena depende da sua complexidade (quantidade de polígonos, texturas e efeitos), a renderização de cenas complexas acaba levando mais tempo que o intervalo entre as atualizações da tela do dispositivo. Isto faz com que sejam empilhadas muitas chamadas aos comandos do OpenGL ES tornando o desenho dos objetos lento e prejudicando o desempenho da aplicação.

Uma possível solução para este problema é apresentada em APPLE INC (2011c), onde é sugerido o uso da propriedade `frameInterval` de `CADisplayLink` para especificar o número de vezes que a tela será atualizada antes da rotina de desenho ser chamada. Atribuindo o valor `3` para esta propriedade, por exemplo, faz com que o *display link* seja disparado a cada três `frames`, ou seja, vinte vezes por segundo. No caso da RAios-library, esta solução não elimina o problema, pois dependendo da cena criada pela aplicação do desenvolvedor, o tempo de renderização pode ser maior que a soma do intervalo entre três `frames`.

A rotina de desenho é mostrada no Quadro 26. A mensagem `presentRenderbuffer` merece destaque, pois o conteúdo renderizado pelo motor gráfico somente é desenhado na tela do dispositivo após este comando.

```
- (void)drawView:(CADisplayLink*)displayLink{

    // (...)

    // renderiza os objetos
    [m_glRenderEngine renderObjects:m_raEngine->m_arrayObjetos toRoll:m_raEngine->m_roll
                      toAzimuth:m_raEngine->m_azimuth
                      orientacaoDispositivo:m_raEngine->m_deviceOrientation
                      desenharFPS:m_bHabilitarFPS
                      fps:m_frameRate];

    // apresenta o conteúdo do render buffer
    [m_context presentRenderbuffer:GL_RENDERBUFFER];

    // (...)
}
```

Quadro 26 – Código fonte do método `drawView`

O *loop* de animação é encerrado utilizando o método `stopDrawView` (Quadro 27). A mensagem `invalidate` remove o *loop* de animação da *thread* principal e decrementa o *retain count* do alvo do *display link*, neste caso a instância de `GLView`, fazendo com que o método `dealloc` da classe seja chamado.

```
- (void)stopDrawView{
    [m_displayLink invalidate];
}
```

Quadro 27 – Encerrando o loop de animação

Além de ser responsável pela animação dos objetos virtuais, a classe `GLView` também

controla a iteração do usuário com os objetos virtuais através de toques na tela do dispositivo. O Quadro 28 mostra o código do método `touchesBegan`, que é disparado quando um toque na tela do dispositivo, que esteja dentro da `view` criada por `GLView`, é identificado pelo iOS. Este método identifica a coordenada da tela onde ocorreu o toque e chama o método `checkCollision` da classe `GLRenderEngine` para verificar se o usuário tocou algum objeto.

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event{

    // Pega a coordenada da tela onde ocorreu o toque
    UITouch*   touch = [[event touchesForView:self] anyObject];
    CGPoint ponto = [touch locationInView:self];

    // Verifica se o toque colide com algum objeto
    [m_glRenderEngine checkCollision:m_raEngine->m_arrayObjetos touchX:ponto.x touchY:ponto.y
    orientacaoDispositivo:m_raEngine->m_deviceOrientation];
}
```

Quadro 28 – Iteração com a tela do dispositivo

3.3.5.2 A classe `GLRenderEngine`

Essa classe é responsável por renderizar os objetos virtuais utilizando o OpenGL ES. O construtor da classe, método `initForContext` demonstrado no Quadro 29, é responsável por alocar a memória necessária para a renderização.

```
- (id)initForContext:(EAGLContext*)context fromDrawable:(CAEAGLLayer*)eaglLayer{
    self = [super init];
    if(self){
        // (...)
        // gera um identificador para o renderBuffer
        glGenRenderbuffersOES(1, &m_renderBuffer);

        // vincula o renderBuffer no pipeline
        glBindRenderbufferOES(GL_RENDERBUFFER_OES, m_renderBuffer);

        // Antes de renderizar qualquer coisa é necessário alocar espaço de armazenamento
           para o renderBuffer enviando uma mensagem para o EAGLContext

        // utiliza a largura, altura e o formato de pixel do layer (eaglLayer)
        [context renderbufferStorage:GL_RENDERBUFFER fromDrawable:eaglLayer];

        // gera um identificador para o frameBuffer
        glGenFramebuffersOES(1, &m_frameBuffer);

        // vincula o frameBuffer no pipeline
        glBindFramebufferOES(GL_FRAMEBUFFER_OES, m_frameBuffer);

        // vincula o renderBuffer ao frameBuffer
        glFramebufferRenderbufferOES(GL_FRAMEBUFFER_OES, GL_COLOR_ATTACHMENT0_OES,
        GL_RENDERBUFFER_OES, m_renderBuffer);

        // checa o status do frame buffer
        if(glCheckFramebufferStatusOES(GL_FRAMEBUFFER_OES) != GL_FRAMEBUFFER_COMPLETE_OES) {
            NSLog(@"Erro ao criar o frame buffer %x",
                  glCheckFramebufferStatusOES(GL_FRAMEBUFFER_OES));
            return nil;
        }
    }
    return self;
}
```

Quadro 29 – Alocar memória para o OpenGL ES

O método `initializeEngine` (Quadro 30) deve ser chamado para inicializar o motor de renderização. Esse método é responsável por configurar a `viewport`, a matriz de projeção, o `frustum` e a matriz modelo, além de configurar alguns estados do OpenGL ES que não são alterados durante a execução do aplicativo, como *array* de vértices, o *array* de cores, as texturas 2D e a transparência.

```
- (BOOL)initializeEngine:(CGRect)frame{

    // Guarda o frame
    m_frame = frame;

    // inicializa a viewport
    glViewport(0, 0, CGRectGetWidth(frame), CGRectGetHeight(frame));
    glGetIntegerv( GL_VIEWPORT, m_viewport );

    // inicializa a matriz de projeção
    glMatrixMode(GL_PROJECTION);

    // configura o frustum
    float aspect = CGRectGetWidth(frame)/CGRectGetHeight(frame);
    glFrustumf(-aspect, aspect, -1, 1, m_zNear, m_zFar);
    glGetFloatv( GL_PROJECTION_MATRIX, m_projection );

    // configura a modelview
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glGetFloatv( GL_MODELVIEW_MATRIX, m_modelview );

    // habilita os atributos de posição e cor para os vertices
    glEnableClientState(GL_VERTEX_ARRAY);
    glEnableClientState(GL_COLOR_ARRAY);

    // para suportar a renderização de texturas
    glEnable(GL_TEXTURE_2D);
    glEnable(GL_BLEND);
    glBlendFunc(GL_ONE, GL_ONE_MINUS_SRC_ALPHA);

    return YES;
}
```

Quadro 30 – Método `initializeEngine`

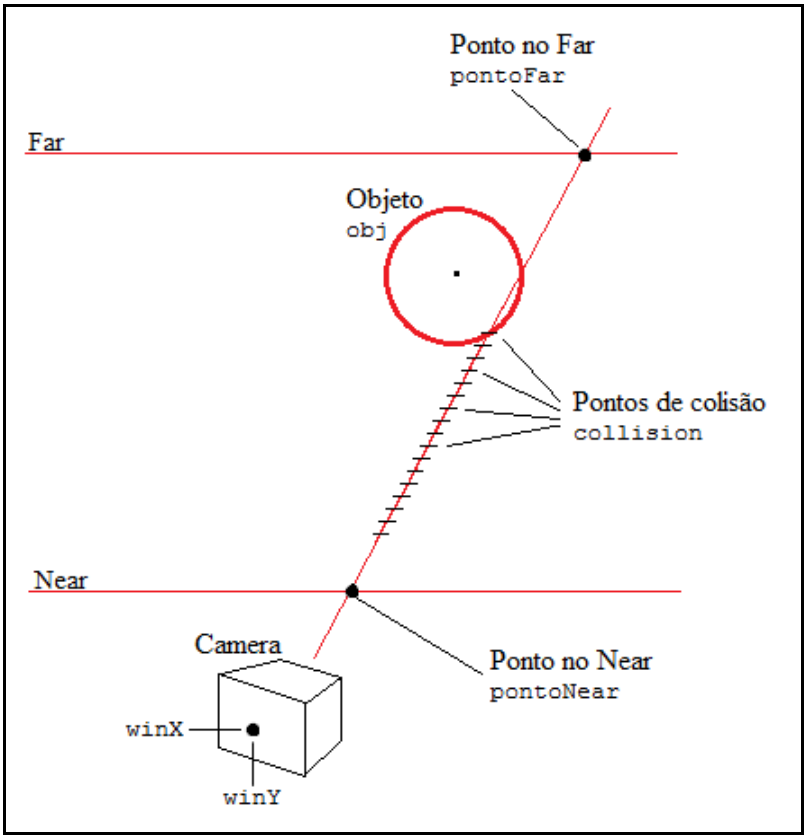
O método que cria a cena contendo os objetos virtuais chama-se `renderObjects` e sua implementação está demonstrada no Quadro 31. Logo ao iniciar é chamado o comando `glLoadIdentity` para limpar as transformações efetuadas. Em seguida, é feita uma limpeza no *buffer* de cores e no *buffer* de profundidade. A orientação atual do dispositivo é então verificada para saber se deve renderizar as setas ou o objeto virtual. Através dos valores de `roll` e `azimuth` calculados pela *engine*, o espaço 3D é rotacionado nos eixos Y e Z. Se as setas estiverem sendo renderizadas, o fundo da cena é preenchido com a cor preta, caso contrário, o fundo da cena é tornado transparente. Por fim, é desenhada a seta ou o objeto virtual para cada ponto de interesse.

```
-(void) renderObjects:/*lista de parâmetros omitida*/
{
    glLoadIdentity();
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);

    // verifica se é para desenhar as setas ou o objeto
    bool bDesenharSetas = (orientacao == UIDeviceOrientationFaceUp);

    // Rotaciona conforme a orientação
    if(bDesenharSetas){
        glRotatef(azimuth, 0.0f, 0.0f, 1.0f);
    }
    else{
        glRotatef(roll, 0.0f, 1.0f, 0.0f);
        glRotatef(azimuth, 0.0f, 0.0f, 1.0f);
    }

    // guarda a model view matrix
    glGetFloatv( GL_MODELVIEW_MATRIX, m_modelview );

    if(bDesenharSetas)
        glClearColor(0, 0, 0, 1); // Limpa o renderBuffer para o fundo opaco
    else
        glClearColor(0, 0, 0, 0); // Limpa o renderBuffer para o fundo transparente

    // desenha os pontos de interesse
    for (Objeto* obj in objects) {
        // verifica se o ponto está visivel
        if (obj->m_foraTela)
            continue;

        // insere a matriz de transformação corrente na pilha, por causa das transformações
        glPushMatrix();

        if(bDesenharSetas)
        {
            // alinha o ângulo e a posição para desenhar a seta
            glTranslatef(obj->m_glSetaX, obj->m_glSetaY, obj->m_glSetaZ);
            glRotatef(-obj->m_glAngulo, 0.0f, 0.0f, 1.0f);

            // desenha a seta
            [SetaDraw renderSetaForAngle:-obj->m_glAngulo forDistance:obj->m_geoDistancia];
        }
        else
        {
            // alinha o ângulo e a posição para desenhar o objeto
            glTranslatef(obj->m_glObjetoX, obj->m_glObjetoY, obj->m_glObjetoZ);
            glRotatef(90.0, 1.0f, 0.0f, 0.0f);
            glRotatef(-obj->m_glAngulo, 0.0f, 1.0f, 0.0f);

            // pega a rotina de desenho do objeto
            [obj onDraw];
        }

        // retira a matriz do topo da pilha
        glPopMatrix();
    }

    // (...)
}
```

Quadro 31 – Renderização da cena

O método `checkCollision` é responsável por verificar se um toque na tela do dispositivo foi efetuado sobre algum objeto virtual de um ponto de interesse. O algoritmo utilizado para realizar esta tarefa chama-se `ray picking` e sua representação pode ser visualizada na Figura 13. O toque na tela é representado por um ponto 2D cujas coordenadas estão contidas nas variáveis `winX` e `winY`. Essa coordenada é convertida em dois pontos no espaço 3D onde os objetos estão dispostos, `pontoNear` e `pontoFar`. Através desses dois

pontos, é obtida uma reta que corta o espaço 3D. O algoritmo então percorre a reta criando pontos de colisão (`collision`) e, para cada ponto gerado, verifica-se se há colisão com cada ponto de interesse, representados na Figura 13 por `obj`.



Fonte: Vasselai (2010, p. 75).

Figura 13 – Algoritmo de colisão do toque

A implementação do método `checkCollision` de acordo com o algoritmo `ray picking` é mostrada no Quadro 32. A conversão do ponto no plano 2D para o espaço 3D é feita utilizando o método `gluUnProject` disponibilizado por MESA (2011). O método `verificaColisao`, utilizado para verificar se um ponto de colisão da reta colide com algum objeto, utiliza o conceito de função `inline`, onde o seu código é copiado para o lugar onde a função é chamada no momento da compilação.

```
- (bool)checkCollision:(NSMutableArray*)objects touchX:(float)x touchY:(float)y
orientacaoDispositivo:(UIDeviceOrientation)orientacao{

    // verifica se é para checar as setas ou o objeto
    bool bChecarSetas = (orientacao == UIDeviceOrientationFaceUp);

    // (...)

    // converte a coordenada y
    y = (float)m_viewport[3] - y;

    // obtem o ponto no plano near
    gluUnProject( x, y , 0, m_modelview, m_projection, m_viewport, &nearPoint.x,
    &nearPoint.y, &nearPoint.z);

    // obtem o ponto no plano far
    gluUnProject( x, y,  1, m_modelview, m_projection, m_viewport, &farPoint.x,
    &farPoint.y, &farPoint.z);

    // (...)

    // percorre o vetor ray para verificar colisões
    float RAY_ITERATIONS = m_zFar * 10;
    for(int i = 0; i < RAY_ITERATIONS; i++)
    {
        collisionPoint.x = rayVector.x * rayLength/RAY_ITERATIONS*i;
        collisionPoint.y = rayVector.y * rayLength/RAY_ITERATIONS*i;
        collisionPoint.z = rayVector.z * rayLength/RAY_ITERATIONS*i;
        // verifica a colisão em cada ponto de interesse
        for(Objeto* obj in objects){
            if(obj->m_foraTela) // verifica se o objeto está fora da tela
                continue;

            if(bChecarSetas){// checa a colisão com a seta
                objectCenter.x = obj->m_glSetaX;
                objectCenter.y = obj->m_glSetaY;
                objectCenter.z = obj->m_glSetaZ;

                //Checking collision
                if( verificaColisao(collisionPoint, objectCenter, 2.0f)){
                    [obj onSelect];
                    return true;
                }
            }
            else{
                // checa a colisão com o painel
                // (...)
            }
        }
    }

    return false;
}
```

Quadro 32 – Verificar colisões no toque

3.3.5.3 A classe `PainelDraw`

A classe `PainelDraw` desenha as formas geométricas utilizando um conjunto de triângulos definidos em um vetor de vértices criado de forma estática na definição da classe. Esses vetores são desenhados utilizando a primitiva `GL_TRIANGLES`. O método contendo os comandos OpenGL ES necessários para criar um painel chama-se `renderPainel` e é mostrado no Quadro 33.

```
+ (void)renderPainel{
    glVertexPointer(3, GL_FLOAT, sizeof(struct vertex), &VerticesPainel[0].positon[0]);
    glColorPointer(4, GL_FLOAT, sizeof(struct vertex), &VerticesPainel[0].color[0]);
    int vertexCount = sizeof(VerticesPainel)/sizeof(struct vertex);
    glDrawArrays(GL_TRIANGLES, 0, vertexCount);
}
```

Quadro 33 – Desenho de um painel

3.3.6 Renderização de texto no OpenGL ES

As classes da RAios-library responsáveis pela renderização de texto no OpenGL ES foram desenvolvidas baseando-se na solução apresentada em STACK EXCHANGE INC (2011).

A classe `GLTextura` utiliza recursos do *framework Core Graphics* do iOS para criar uma textura retangular, de fundo transparente, contendo o texto a ser renderizado. O método `initWithTexto` (Quadro 34) é responsável pela criação da textura.

```
- (id)initWithTexto: /*parâmetros omitidos*/{
    //(...)
        NSInteger BitsPerComponent = 8;
        int bpp = BitsPerComponent / 2;
        int byteCount = largura * altura * bpp;
        uint8_t *data = (uint8_t*) calloc(byteCount, 1);

        CGColorSpaceRef colorSpace = CGColorSpaceCreateDeviceRGB();
        CGBitmapInfo bitmapInfo = kCGImageAlphaPremultipliedLast | kCGBitmapByteOrder32Big;
        CGContextRef context = CGBitmapContextCreate(data,
                                                     largura,
                                                     altura,
                                                     BitsPerComponent,
                                                     bpp * largura,
                                                     colorSpace,
                                                     bitmapInfo);
        CGColorSpaceRelease(colorSpace);
        CGContextSetRGBFillColor(context, red, green, blue, alpha);
        CGContextTranslateCTM(context, 0.0f, altura);
        CGContextScaleCTM(context, 1.0f, -1.0f);
        UIGraphicsPushContext(context);

        [texto drawInRect:CGRectMake( 0,
                                      0,
                                      size.width,
                                      size.height)
            withFont:fonte
            lineBreakMode:UILineBreakModeWordWrap
            alignment:UITextAlignmentCenter];

        UIGraphicsPopContext();
        CGContextRelease(context);

        dadosTextura = (uint8_t *)[[NSData dataWithBytesNoCopy:data length:byteCount
                                            freeWhenDone:YES] bytes];
    //(...)
}
```

Quadro 34 – Criar textura

A classe `GLTextRenderer` contém os comandos OpenGL ES necessários para renderizar a textura criada pela classe `GLTextura`. O método `renderText` (Quadro 35) é responsável por aplicar a textura gerada na superfície de um polígono tornando o texto visível.

```
+ (void)renderText:/*parâmetros omitidos*/{

    // cria a textura
    GLTextura *textura = [[GLTextura alloc]initWithTexto:text forFonte:font forRed:red
                                                forGreen:green forBlue:blue forAlpha:alpha];

    // habilita o array de coordenadas de textura
    glEnableClientState(GL_TEXTURE_COORD_ARRAY);

    // Cria um identificador para a textura
    GLuint            texture[1];
    glGenTextures(1, &texture[0]);
    glBindTexture(GL_TEXTURE_2D, texture[0]);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

    // isto é necessário para texturas que não são potência de dois
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);

    // utiliza BRGA para gerar o quadro com a textura
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, textura.largura, textura.altura, 0, GL_BGRA,
                 GL_UNSIGNED_BYTE, textura.dadosTextura);

    glVertexPointer(3, GL_FLOAT, 0, vertices);
    glTexCoordPointer(2, GL_FLOAT, 0, textCoords);

    // desenha os polígonos com a textura
    glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);

    // desabilita a primitiva de array de coordenadas de textura
    glDisableClientState(GL_TEXTURE_COORD_ARRAY);

    // deleta a textura criada
    glDeleteTextures(1, &texture[0]);

    // libera a memória alocada para a textura
    [textura release];
}
```

Quadro 35 – Renderizar textura

3.3.7 Operacionalidade da implementação

Para demonstrar a operacionalidade da RAios-library, foi desenvolvida uma aplicação de RA baseada em Vasselai (2010) denominada RAios-sample. Aconselha-se, inicialmente, a utilização da RAios-library em conjunto com a ferramenta de desenvolvimento Xcode, por ser completa e possuir todo suporte oficial da Apple.

3.3.7.1 Visão geral da aplicação

A aplicação RAios-sample foi desenvolvida de forma a possuir um menu inicial com opções que permitem configurá-la e também iniciar o modo de realidade aumentada.

Inicialmente, a aplicação é aberta apresentando a sua tela inicial, conforme é mostrado

na Figura 14.

Figura 14 – Menu inicial da aplicação

Através do botão `configurações` é possível acessar a configuração da aplicação (Figura 15) e habilitar alguns recursos como a informação de sobre a taxa de *frames* por segundo da aplicação, a renderização de texturas e simular a inserção de pontos de interesse. Também é possível definir valores para a distância máxima entre o dispositivo e um ponto de interesse e a distância mínima a ser percorrida lateralmente para que a localização do dispositivo seja atualizada.

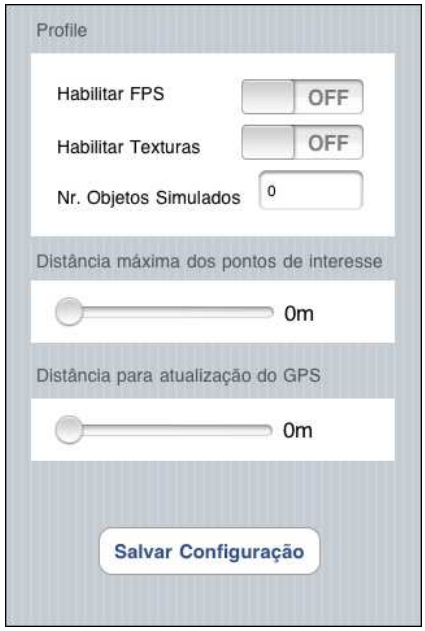


Figura 15 – Tela de configuração da aplicação

Quando o botão `Realidade Aumentada` é pressionado, o código da RAios-library entra em ação para apresentar a cena real sobreposta com as informações sintéticas relacionadas aos pontos de interesse cadastrados. A Figura 16 mostra a tela de realidade

aumentada contendo os objetos virtuais. Esta tela é exibida quando o dispositivo é segurado com a câmera virada para frente. A imagem de fundo é proveniente da câmera do dispositivo e o painel possui o nome do ponto de interesse que está sendo representado.

Figura 16 – Desenho dos objetos virtuais

Quando o dispositivo é segurado com a câmera virada para baixo, são apresentadas as setas indicando a direção e distância dos pontos de interesse, como pode ser observado na Figura 17.

Figura 17 – Desenho das setas

A Figura 18 mostra o diagrama de classes da aplicação RAios-sample e como ela se relaciona com a RAios-library.

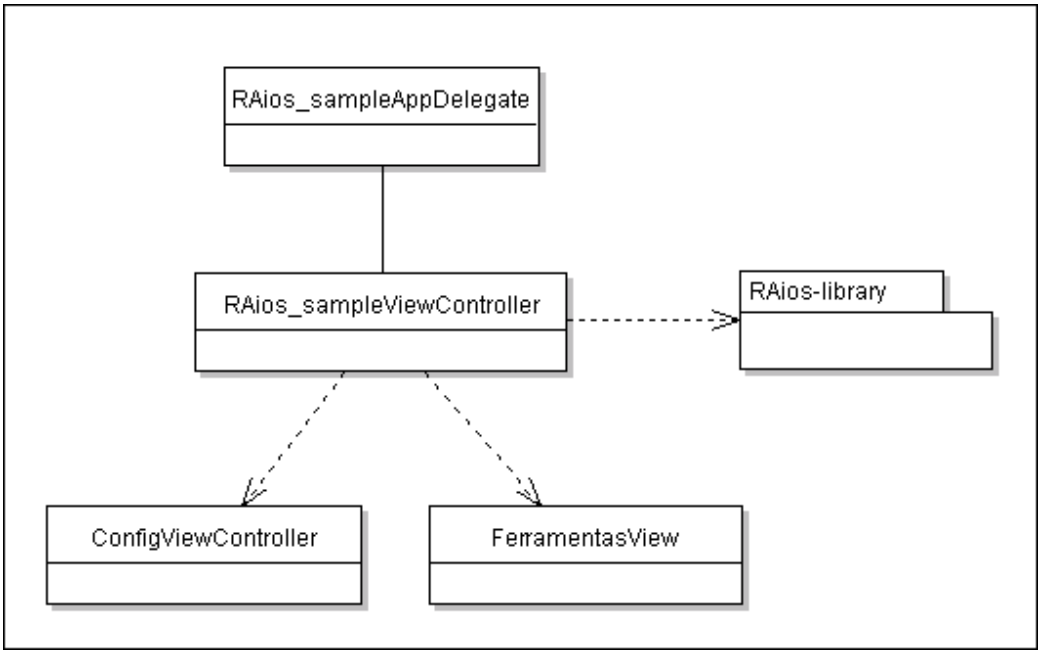


Figura 18 – Diagrama de classes da aplicação RAios-sample

3.3.7.2 Início do projeto

Ao iniciar uma implementação visando à utilização da RAios-library, pode-se optar por utilizar o *template* "*View-based Aplication*" existente no Xcode, conforme Figura 19. Este modelo de projeto fornece um ponto de partida para aplicações que utilizam *views*.

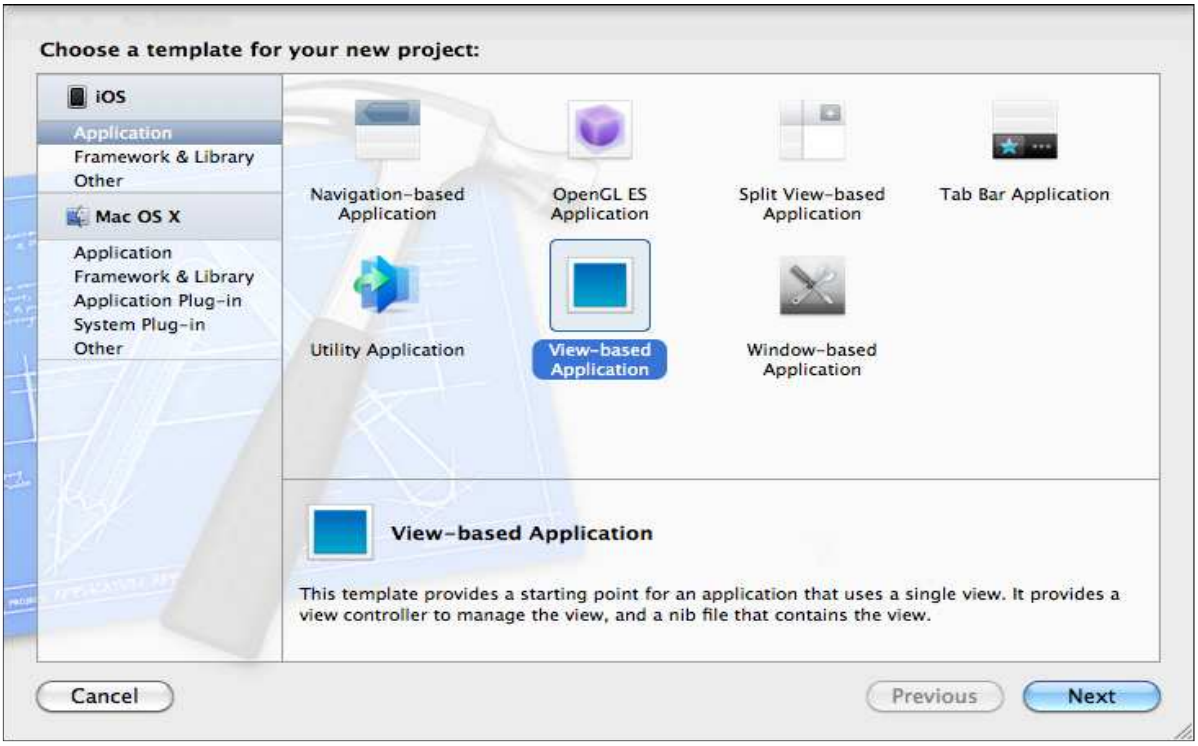


Figura 19 – *Templates* padrões do Xcode

Para o correto funcionamento da RAios-library, é necessário alterar as orientações suportadas pelo o dispositivo na configuração do *template*. Por padrão, este *template* oferece suporte às orientações *portrait*, *landscape lef* e *landscape right*. A configuração deve ser alterada para que seja suportada apenas a orientação *portrait*, conforme Figura 20.

Figura 20 – Configuração da orientação suportada

O próximo passo para a utilização da RAios-library é adicioná-la ao projeto. Para tanto, deve-se arrastar o arquivo `RAios-library.xcodeproj` para a aba "*Project Navigator*" do Xcode, conforme Figura 21.

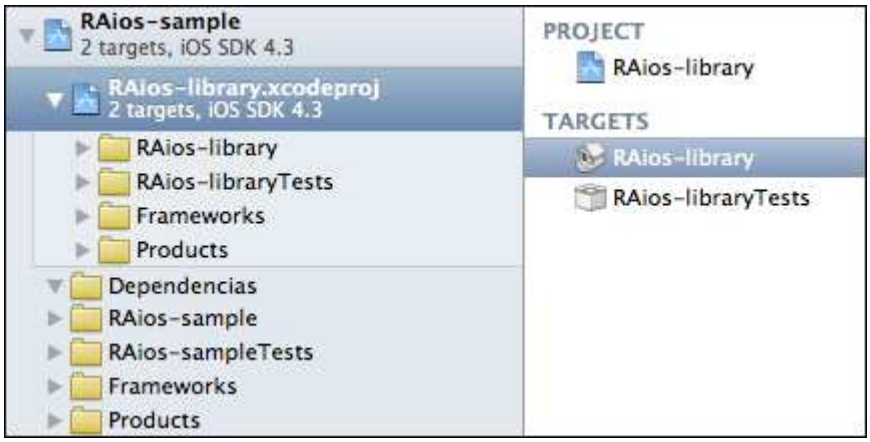


Figura 21 – Adicionando a RAios-library ao projeto

Em seguida, deve-se adicionar a RAios-library como uma dependência do projeto para permitir ao Xcode compilar a biblioteca sempre que o projeto é compilado. Para efetuar este procedimento se deve:

a) selecionar o projeto na aba "*Project Navigator*" do Xcode;

b) selecionar o *target* correspondente;

c) selecionar a aba "*Build Phases*" e expandir a seção "*Target Dependencies*";

d) clicar no botão “+” (adicionar) para exibir o diálogo de adição de dependência;

e) adicionar a RAios-library conforme exibido na Figura 22.

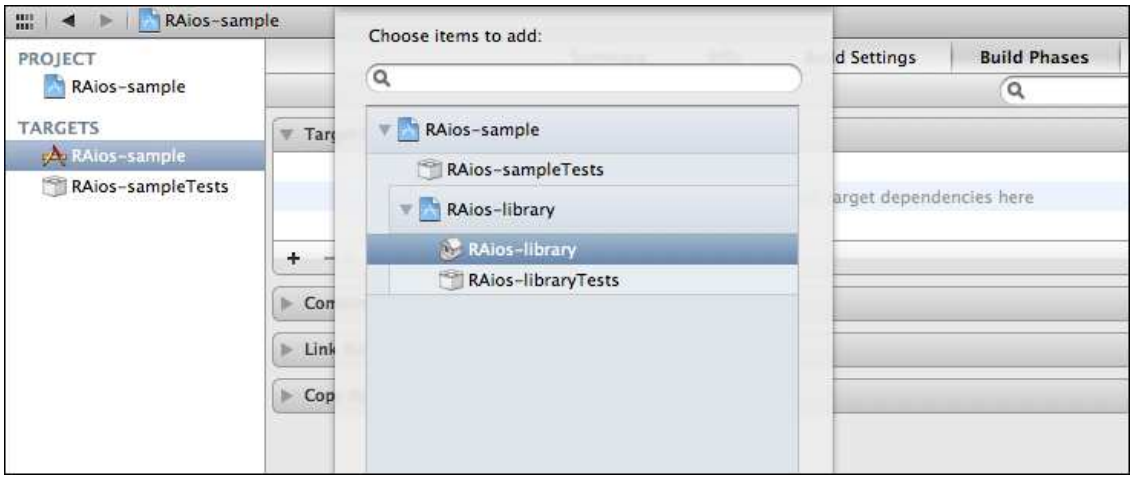


Figura 22 – Configurando as dependências do projeto

Após adicionar a dependência, deve-se vincular a RAios-library (`libRAios-library.a`) ao projeto. Para efetuar este procedimento se deve:

a) selecionar o projeto na aba “*Project Navigator*” do Xcode;

b) selecionar o *target* correspondente;

c) selecionar a aba “*Build Phases*” e expandir a seção “*Link Binary With Libraries*”;

d) clicar no botão “+” (adicionar) para exibir o diálogo de adição de biblioteca;

e) adicionar a `libRAios-library.a` que está localizada na categoria “*Workspace*” conforme exibido na Figura 23.

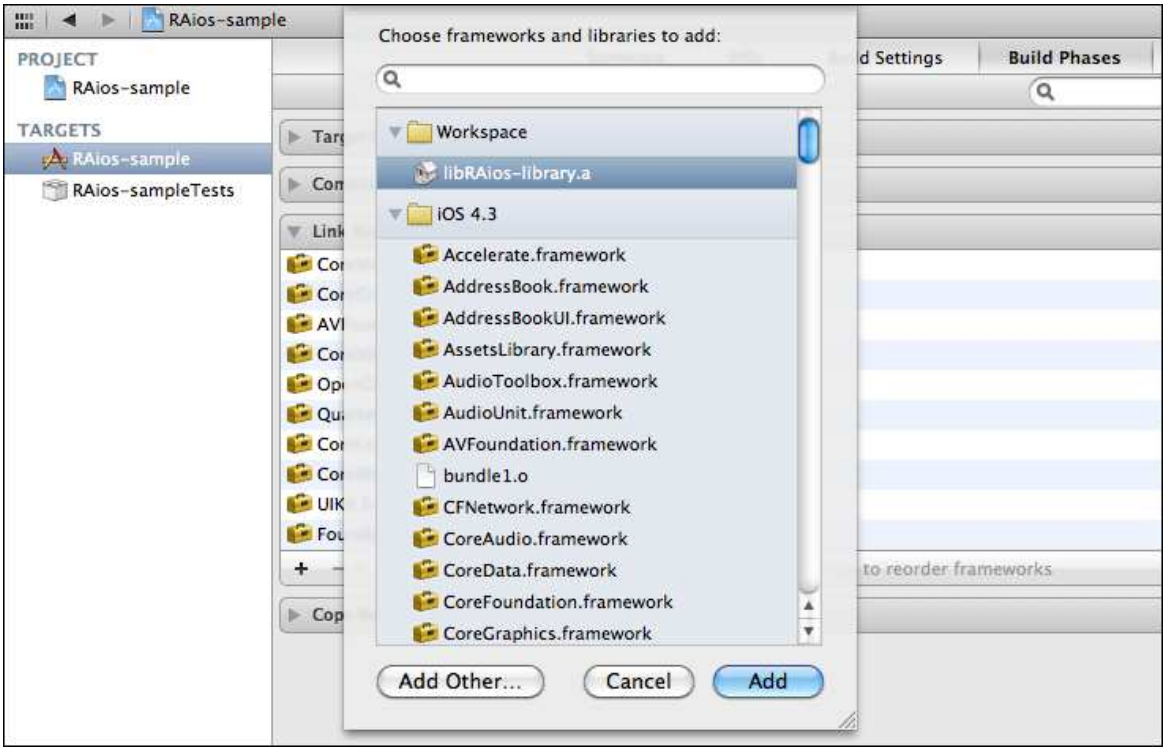


Figura 23 – Vinculando a RAios-library ao projeto

Em seguida, deve-se configurar o caminho de busca de cabeçalhos do projeto para permitir ao Xcode encontrar os cabeçalhos da RAios-library. Para tanto, se deve:

a) selecionar o projeto na aba "*Project Navigator*" do Xcode;

b) selecionar a aba "*Build Settings*" e localizar a seção "*Search Paths*";

c) configurar o item "Header Search Paths" com o caminho para os cabeçalhos da RAios-library. Conforme Figura 24.

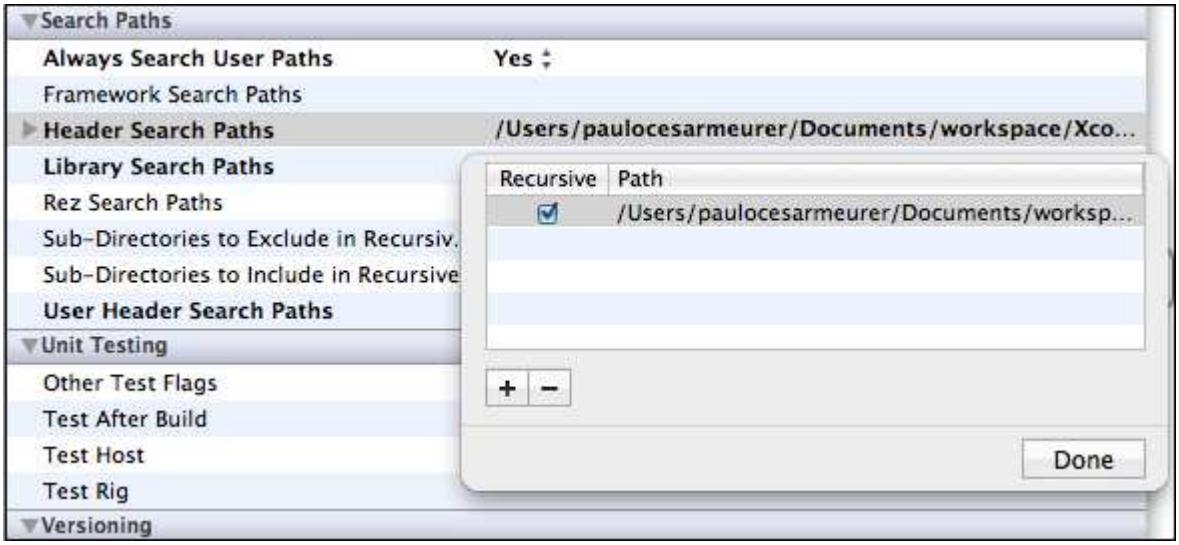


Figura 24 – Confirar *Search Paths* do projeto

Por último, devem ser adicionados ao projeto os *frameworks* do iOS utilizados pela RAios-library. São eles: *Core Media*, *Core Graphics*, AV *Foundation*, *Core Video*, OpenGLES, *Quartz Core*, *Core Location*, *Core Motion*, UIKit e *Foundation.* Os *frameworks* adicionados ao projeto são apresentados na Figura 25.

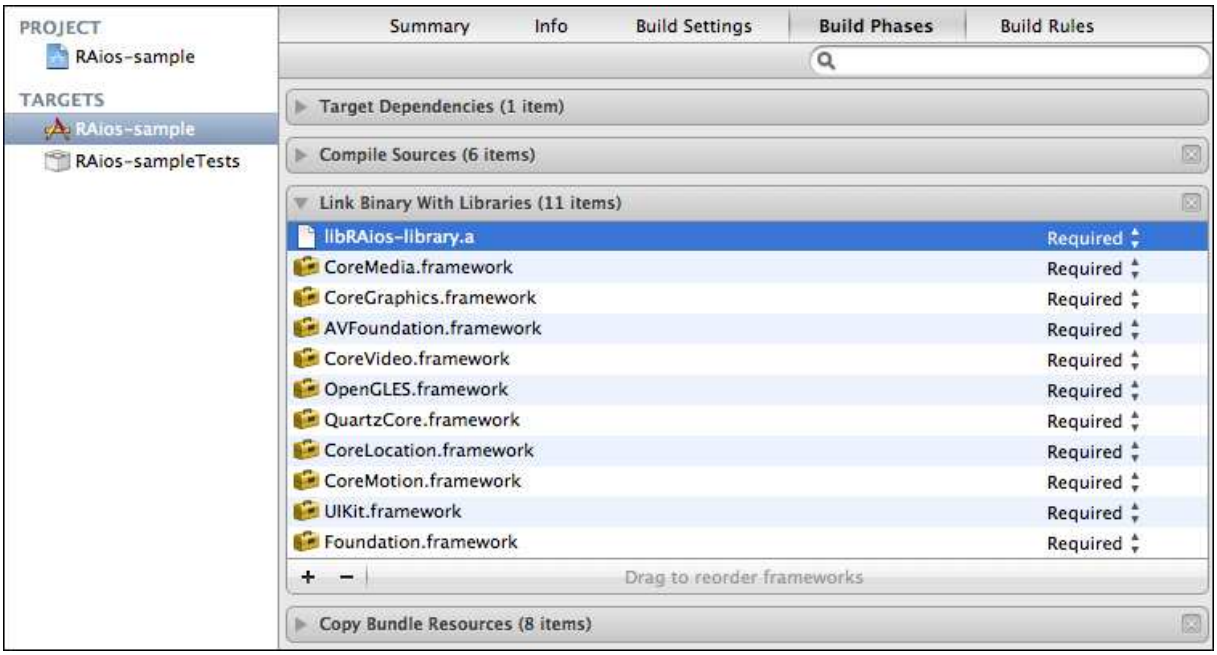


Figura 25 – *Frameworks* utilizados pela RAios-library

3.3.7.3 Utilizando a RAios-library

A RAios-library foi projetada de modo que o desenvolvedor não necessite de conhecimento avançado da plataforma iOS para desenvolver uma aplicação de realidade aumentada com os recursos padrões que a mesma oferece.

O primeiro passo na utilização da RAios-library é a criação de uma instância da classe `RAios`. É através desse objeto que o desenvolvedor terá acesso aos recursos disponibilizados pela biblioteca. Na aplicação RAios-sample desenvolvida, essa tarefa é realizada assim que a `view` principal é carregada (Quadro 36).

```
// Implement viewDidLoad to do additional setup after loading the view, typically from a nib.
- (void)viewDidLoad
{
    [super viewDidLoad];

    self.view.exclusiveTouch = NO;

    // Inicializa a engine
    m_RAios = [[RAios alloc]init];

    // (...)
}
```

Quadro 36 – Alocação da RAios-library

Para simplificar o entendimento, todas as chamadas às funções da RAios-library pela aplicação RAios-sample foram agrupadas na função de `callback` do botão `realidade aumentada`. O Quadro 37 mostra parte do código principal do método e um algoritmo para facilitar o entendimento de como criar uma aplicação utilizando a RAios-library. Os passos do algoritmo estão descritos nos parágrafos seguintes.

```
- (IBAction)onButtonRealidadeAumentada:(id)sender{
    // (...)

    // Inicializa a biblioteca
    [m_RAios arInit:self.view];

    // configurar a captura de vídeo
    // iniciar a captura de vídeo
    // configurar o acelerômetro
    // iniciar a captura de movimento
    // configurar o GPS e a Bussola
    // iniciar a captura das coordenadas GPS
    // cadastrar pontos de interesse
    // configurar os parametros para desenho
    // iniciar o desenho dos objetos

    // (...)
}
```

Quadro 37 – Chamadas aos métodos da RAios-library

Após a inicialização da RAios-library utilizando o método `arInit`, o próximo passo para a criação de uma aplicação de realidade aumentada é a configuração e exibição do conteúdo capturado pela câmera do dispositivo. Os passos necessários para executar esta tarefa são mostrados no Quadro 38.

```
// configura a captura de vídeo
[m_RAios arVideoCapConfigFrameRate:CMTimeMake(1,30)];
[m_RAios arVideoCapConfigVideoQuality:AVCaptureSessionPresetMedium];

 // inicia a captura de vídeo
[m_RAios arVideoCapStart];
```

Quadro 38 – Configuração e exibição da câmera

O Quadro 39 mostra os passos necessários para que a aplicação possa responder aos movimentos do dispositivo.

```
// configura o acelerômetro
[m_RAios arMotionConfigAccelUpdateFrequency:50.0];
[m_RAios arMotionConfigGravityFilterFactor:0.1];

// inicia a captura de movimento
[m_RAios arMotionStart];
```

Quadro 39 – Configuração e ativação do acelerômetro

O próximo passo (Quadro 40) é a configuração do GPS e da bússola, para que a aplicação possa iniciar a captura da localização e direção do dispositivo.

```
// configura o GPS e a Bussola
int distAtuGPS = [[NSUserDefaults standardUserDefaults] integerForKey:@"atuGPS"];
int distMaxObjetos = [[NSUserDefaults standardUserDefaults] integerForKey:@"distPOI"];
[m_RAios arLocationConfigFiltroBussola:0.0];
[m_RAios arLocationConfigDistanciaMaximaObjeto:distMaxObjetos];
[m_RAios arLocationConfigFiltroDistanciaGPS:distAtuGPS];

// inicia a captura das coordenadas GPS
[m_RAios arLocationStart];
```

Quadro 40 – Configuração do GPS e da bússola

O Quadro 41 mostra os passos necessários para inserir um ponto de interesse na aplicação.

```
PontoInteresse *poi;
poi = [[PontoInteresse alloc]initWithDescricao:@"Meu Ponto" latitude:-26.893395
                                                              longitude:-49.126386];
[m_RAios arInsertObject:poi];
[poi release];
```

Quadro 41 – Inserir pontos de interesse

O último passo para a criação da aplicação é a configuração dos parâmetros de desenho e o início da renderização dos objetos virtuais. Os passos necessários para realizar esta tarefa são mostrados no Quadro 42.

```
// configura os parametros para desenho
[m_RAios arDrawConfigRaioDesenhoObjeto:1.5];
[m_RAios arDrawConfigDistanciaObjetoTela:14.0];
[m_RAios arDrawConfigHabilitarFPS:[[NSUserDefaults standardUserDefaults]boolForKey:@"fps"]];
[m_RAios arDrawConfigHabilitarTexturas:[[NSUserDefaults
standardUserDefaults]boolForKey:@"textura"]];

// inicia o desenho dos objetos
[m_RAios arDrawStart];
```

Quadro 42 – Iniciar desenho dos pontos de interesse

3.4 RESULTADOS E DISCUSSÃO

Os resultados obtidos pela RAios-library foram medidos através de testes experimentais da sua capacidade de renderizar cenas. Entende-se que a velocidade de renderização dos objetos virtuais é um fator crítico para o sucesso da biblioteca, pois o atraso no desenho da cena pode causar a perda da noção de realidade pelo usuário e a impressão de que os objetos reais e virtuais estão desalinhados.

Todos os testes foram realizados iniciando a aplicação a partir do Xcode e observando a sua execução no dispositivo iPhone 4. O método utilizado para medir o desempenho foi a contagem do número FPS apresentados pela a aplicação desenvolvida utilizando a RAios-library.

Acredita-se em três fatores principais que podem afetar o desempenho da RAios-library, sendo eles: a quantidade de objetos virtuais presentes na cena a ser renderizada, a renderização de conteúdo transparente e a utilização de texturas.

O primeiro conjunto de testes realizado objetivou medir o impacto da quantidade de objetos a serem renderizados na taxa de FPS da aplicação. Para tanto, foram realizadas medições da taxa de FPS com diferentes quantidades de objetos sendo desenhados ao mesmo tempo. As medições foram realizadas com a transparência habilitada, a câmera desativada e utilizando objetos simples, formados por 2 faces e 6 vértices. Os resultados obtidos são apresentados na Tabela 1.

Tabela 1 – Medição da taxa de FPS na renderização de objetos

Quantidade de objetos	**Média de FPS**
1	40
300	40
600	40
900	35
1200	27
1500	22
1800	19
2100	16
2400	14
2700	12
3000	11

Com base nos resultados obtidos, é possível observar que o desempenho da aplicação mostrou-se sensível a quantidade de objetos que serão renderizados. A Figura 26 mostra o gráfico gerado com base nas medições, com o desempenho obtido através da medição do FPS na vertical e a quantidade de objetos na horizontal.

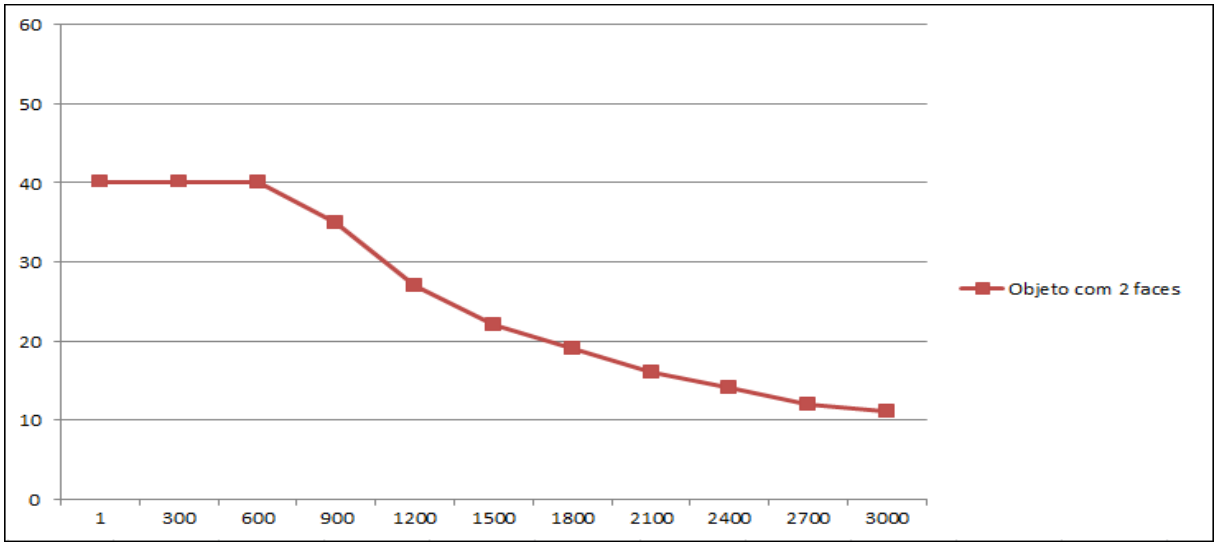


Figura 26 – Gráfico do impacto da quantidade de objetos no desempenho da aplicação

O segundo conjunto de testes realizado visou medir a influência da renderização de conteúdo transparente na taxa de FPS da aplicação. Segundo APPLE INC (2011c), misturar o conteúdo desenhado pelo OpenGL ES com o conteúdo de outras *views* através da transparência acarreta em uma queda de desempenho grave. Este cenário ocorre na aplicação quando os objetos virtuais são desenhados sobre o conteúdo proveniente da câmera do dispositivo, o que pode ser observado na Figura 16. Para a realização deste teste, a câmera do dispositivo foi desabilitada e foram utilizados objetos formados por 2 faces e 6 vértices sem a utilização de texturas. O resultado do FPS obtido está disponível na Tabela 2.

Tabela 2 – Medição da taxa de FPS quanto ao uso de transparência

Quantidade de objetos	**Média FPS sem transparência**	**Média FPS com transparência**
1	60	40
300	60	40
600	52	40
900	36	35
1200	28	27
1500	23	22
1800	19	19
2100	17	16
2400	14	14
2700	13	12
3000	12	11

Para uma melhor análise, este conjunto de testes foi refeito utilizando outra técnica para o desenho dos objetos virtuais. A classe `PainelDraw` foi adaptada para desenhar os objetos utilizando uma técnica conhecida como *Vertex Buffer Objects* (VBOs). Neste caso, o OpenGL ES gerencia o *buffer* de objetos podendo alocá-lo em uma memória mais acessível ao hardware gráfico ou pré-processar os dados para o melhor formato aceito pelo hardware. O

resultado do FPS obtido neste teste está disponível na Tabela 3.

Tabela 3 – Medição da taxa de FPS quanto ao uso de transparência utilizando VBOs

Quantidade de objetos	Média FPS sem transparência	Média FPS com transparência
1	60	40
300	60	40
600	58	40
900	43	39
1200	34	31
1500	27	26
1800	23	22
2100	20	19
2400	17	16
2700	16	15
3000	15	13

O desenho dos objetos utilizando VBOs mostrou um desempenho superior comparado ao *array* de vértices utilizado pela RAios-library. Pode-se notar também que independentemente do método utilizado para o desenho dos objetos, quando a transparência está habilitada a aplicação não atinge uma taxa superior a 40 FPS, revelando que o *Core Animation* utiliza algum tipo de heurística para gerenciar o desenho do conteúdo de *views* com transparência habilitada. Percebe-se também que na medida em que a taxa de FPS da aplicação fica abaixo de 40 a transparência deixa de ser um fator limitante e a aplicação apresenta resultados similares para ambos os casos.

A Figura 27 apresenta o gráfico gerado com base nas quatro medições, com o desempenho obtido através da medição do FPS na vertical e a quantidade de objetos na horizontal. Neste gráfico é bastante perceptível que o uso de transparência afeta o desempenho da aplicação de forma severa apenas com taxas de FPS acima de 40.

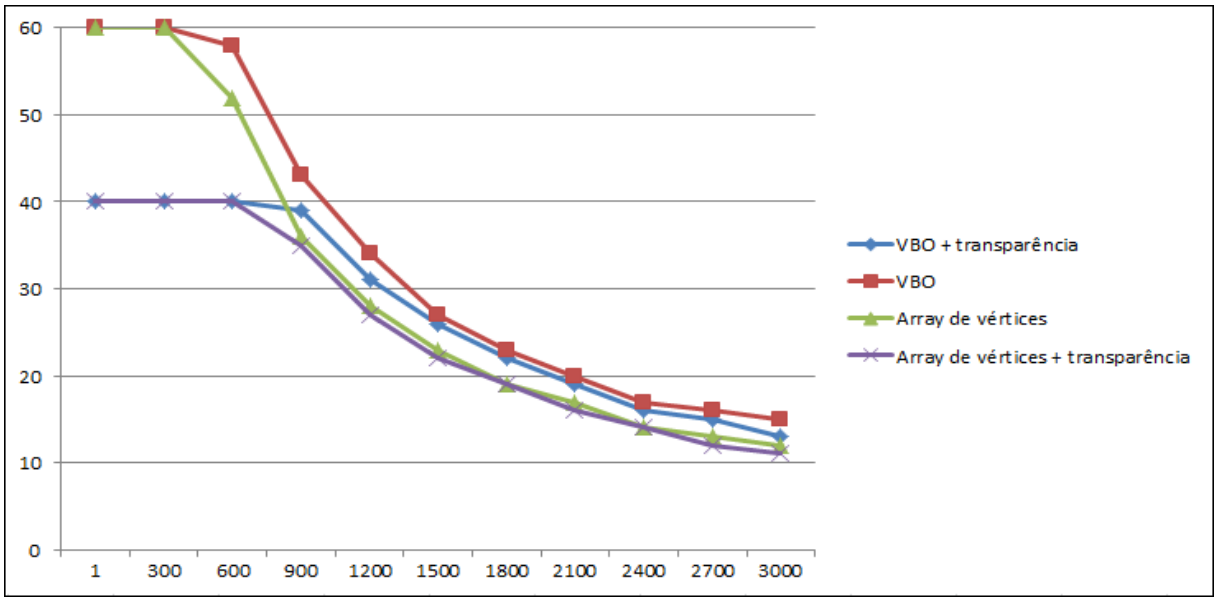


Figura 27 – Gráfico do impacto do uso de transparência

O último conjunto de testes realizado visou medir o desempenho da aplicação quanto à sua capacidade de renderização de texturas. Para tanto, foram realizadas medições da taxa de FPS da aplicação utilizando objetos contendo 2 faces e 6 vértices sobre os quais era aplicada uma textura. Todos os testes de textura foram efetuados com a transparência habilitada e a câmera configurada para capturar 30 `frames` por segundo.

A primeira medição foi realizada criando a textura antes do desenho de cada objeto e deletando a mesma ao final do desenho, refletindo a lógica de desenho da classe `PainelDraw`. Uma segunda medição foi efetuada adaptando a rotina de desenho de texturas da RAios-library para criar a textura uma única vez e atualizar o seu conteúdo quando do desenho dos objetos seguintes. Por fim, uma terceira medição foi realizada desenhando uma textura fixa, cujo conteúdo era igual para todos os objetos. Os resultados obtidos neste conjunto de testes são apresentados na Tabela 4.

Tabela 4 – Medição da taxa de FPS na renderização de texturas

Quantidade de objetos	**Média FPS recriando a textura**	**Média FPS atualizando a textura**	**Média FPS com textura fixa**
1	40	40	40
2	37	40	40
3	31	40	40
4	25	37	40
5	23	36	40
6	20	34	39
7	17	31	38
8	16	28	37
9	15	26	36
10	13	25	36

Os resultados obtidos mostram que a renderização de texturas afeta bastante o desempenho da aplicação. A atualização do conteúdo da textura ao invés de criar uma nova textura para cada objeto mostrou-se uma forma mais eficiente de renderizar texturas dinâmicas. A aplicação apresenta um desempenho muito superior quando o conteúdo das texturas é estático, revelando que a queda de rendimento se deve em grande parte ao tempo gasto na criação das texturas. A Figura 28 apresenta o gráfico gerado com base nas três medições, com o desempenho obtido através da medição do FPS na vertical e a quantidade de objetos na horizontal.

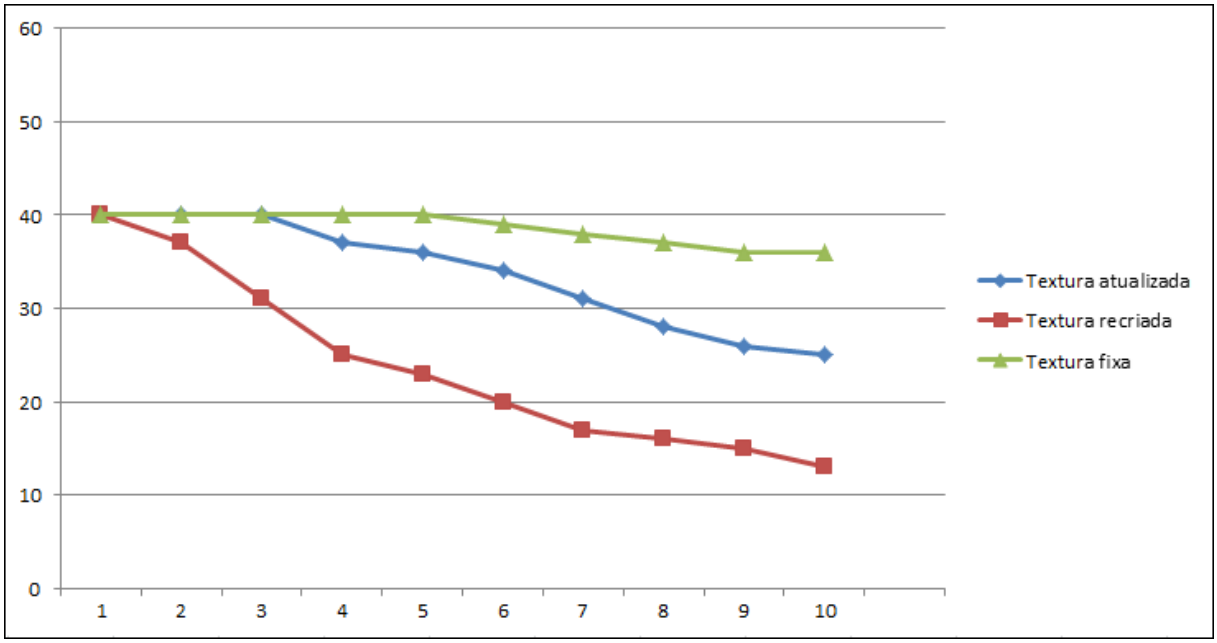


Figura 28 – Gráfico do impacto da renderização de textura no desempenho da aplicação

3.4.1 Objetivos alcançados

O presente trabalho apresentou o projeto e desenvolvimento de uma biblioteca de realidade aumentada para a plataforma iOS, que permite a construção de aplicativos que seguem o conceito de RA aplicando as três propriedades básicas definidas por Azuma (1997, p. 356) e explicadas na seção 2.1.

A biblioteca desenvolvida obteve êxito na criação de uma camada de abstração que oculta os detalhes de implementação da aquisição de imagens da câmera, do registro, do rastreamento, do ajuste visual dos objetos virtuais e da sobreposição das cenas do mundo real e virtual utilizando os recursos de GPS, acelerômetro, bússola e câmera digital disponíveis na plataforma iOS.

Outro objetivo alcançado, foi a utilização com sucesso do conceito *markerless tracking*, que permitiu efetuar o registro e rastreamento dos objetos virtuais através de coordenadas geográficas evitando a necessidade da presença de marcadores fixos no ambiente.

Foi apresentada também, uma solução para renderização de texto no OpenGL ES, já que o mesmo não suporta o desenho de texto nativamente. Apesar de não apresentar o desempenho esperado, esta solução pode ser adaptada ou utilizada como ponto de partida para a criação de rotinas de desenho de texto mais eficientes.

O presente trabalho ainda disponibiliza uma aplicação de RA simples (RAios-sample), construída utilizando os recursos fornecidos pela biblioteca desenvolvida.

Inicialmente a proposta era de utilizar também o giroscópio para detectar a mudança de direção do dispositivo. Porém, o sensor do giroscópio não foi utilizado para manter a RAios-library compatível com todos os dispositivos que utilizam a plataforma iOS, haja vista que este sensor foi introduzido apenas nos dispositivos mais recentes da empresa Apple como o iPhone 4, o iPhone 4S, o iPad 2 e o iPod *touch* de quarta geração.

Em sua concepção inicial, a RAios-library também deveria fornecer meios para a aquisição de coordenadas de pontos de interesse através da rede 3G e/ou *wireless*. Este requisito acabou não sendo implementado, pois no decorrer do desenvolvimento da RAios-library verificou-se que não se tratava de uma necessidade primária. Os esforços de pesquisa foram então voltados à criação de recursos para a criação de uma aplicação em si, deixando a aquisição de dados sob responsabilidade do desenvolvedor.

3.4.2 Dificuldades encontradas

A principal dificuldade encontrada durante o desenvolvimento do trabalho foi a ausência de uma obra literária abordando com profundidade a utilização do OpenGL ES com a linguagem Objective-C na plataforma iOS. Apesar de a documentação disponibilizada pela Apple apresentar um conteúdo rico e abrangente, muitos assuntos são abordados de forma muito rápida e abstrata, exigindo do leitor um conhecimento prévio e experiência com desenvolvimento de aplicações utilizando o OpenGL.

A falta de suporte do simulador oficial da Apple aos recursos utilizados pela biblioteca também trouxe dificuldades na depuração do projeto. Muitas das funcionalidades do *instruments*, aplicação disponibilizada pela Apple para análise de performance da aplicação, não estão disponíveis quando a mesma é depurada diretamente no dispositivo. Um recurso muito útil desta aplicação que não pôde ser utilizado foi o `NSZombie`, que detecta a passagem de mensagem para objetos cujo *reference count* é zero.

Renderizar texto no OpenGL ES 1.1 também se mostrou um tarefa desafiadora. A solução encontrada possui grandes limitações e afeta de forma severa o desempenho da aplicação. Uma forma de melhorar o desempenho da rotina de renderização de texto da biblioteca foi descoberta durante os testes de rendimento. A textura que contém o texto deve ser gerada uma única vez durante a execução da aplicação, e apenas o conjunto de dados que

forma a textura deve ser alterado quando há mudança no texto.

Outra dificuldade encontrada foi a utilização de um fundo transparente na cena renderizada pelo OpenGL ES. Através dos testes realizados, ficou perceptível que a transparência limita o desempenho da aplicação a 40 FPS. Como solução a este problema, sugere-se eliminar a `view` que exibe os dados provenientes da câmera e, ao invés de utilizar a transparência, criar uma textura com a imagem capturada pela câmera e renderizá-la no fundo da cena.

A última dificuldade encontrada, diz respeito à sincronização da rotina de desenho com a taxa de atualização da tela do dispositivo. Como a classe `CADisplayLink` dispara a rotina de desenho sempre que a tela do dispositivo está prestes a ser atualizada, sempre que um `frame` leva mais tempo para ser gerado do que o intervalo entre as atualizações da tela, ocorre um atraso na apresentação do `frame` seguinte, o que acarreta na queda da taxa de FPS da aplicação.

4 CONCLUSÕES

Este trabalho apresentou o projeto e desenvolvimento de uma biblioteca de realidade aumentada para a plataforma iOS. Foi desenvolvida uma camada de abstração que oculta os detalhes de implementação da aquisição de imagens da câmera, do registro, do rastreamento, do ajuste visual dos objetos virtuais e da sobreposição das cenas do mundo real e virtual utilizando os recursos de GPS, acelerômetro, bússola e câmera digital. Estas funcionalidades, porém, não foram suficientes para cumprir todos os requisitos inicialmente propostos e a RAios-library não permite a aquisição de coordenadas através da rede 3G e o uso do sensor do giroscópio.

Os resultados finais foram satisfatórios, pois foi possível construir uma aplicação de realidade aumentada utilizando os recursos disponibilizados pela RAios-library. Se houver a necessidade de aprimorar o desempenho, poderia se ter optado em uma estrutura menos fundamentada na orientação a objetos. Outro ganho de desempenho pode ser obtido eliminando-se a utilização do fundo transparente na cena renderizada pelo OpenGL ES. Quanto a rotina de renderização de texturas, acredita-se em no mínimo outros dois caminhos que iriam melhorar o desempenho.

Quanto às tecnologias utilizadas, o ambiente de desenvolvimento Xcode mostrou-se eficiente e facilitou o desenvolvimento. Já o conjunto de recursos da ferramenta *instruments* foi de grande valia para a depuração da biblioteca.

A RAios-library possibilitou criar uma aplicação com funcionalidades semelhantes às dos trabalhos correlatos. Destacando-se entre elas o rastreamento de pontos de interesse sem o uso de marcadores através de coordenadas geográficas, a visualização da cena real através da câmera do dispositivo e a sobreposição de informação sintética através de objetos desenhados em um sistema 3D.

Por fim, a biblioteca desenvolvida possui uma interface simples de utilizar e que permite a desenvolvedores sem conhecimento profundo da plataforma iOS desenvolverem aplicações que implementam o conceito de realidade aumentada.

4.1 EXTENSÕES

Durante o desenvolvimento foram verificadas algumas possíveis melhorias que não foram contempladas neste trabalho.

A versão do OpenGL ES utilizada para o desenvolvimento da biblioteca foi a 1.1. Como melhoria, sugere-se portar a RAios-library para a versão 2.0, com a finalidade de usufruir do ganho de desempenho proporcionado pela utilização de *shaders*.

Ainda com relação ao desempenho, é sugerida a utilização de VBOs como técnica de desenho, por se tratar de um método mais rápido e que permite otimizar o consumo de memória pela aplicação, pois neste caso é o OpenGL ES que gerencia a estrutura contendo os vértices do desenho. A redução do número de chamadas ao método `glDrawArrays` também é aconselhada. Neste caso, deve-se preparar o maior conjunto de vértices possível antes de efetuar esta operação. Quanto à renderização de texturas, se sugere criar a textura apenas uma vez e durante a execução do aplicativo alterar o seu conteúdo quando necessário.

Outra melhoria sugerida é a alteração do motor de desenho gráfico para desenhar apenas os objetos que estejam no campo de visão do usuário. Hoje a RAios-library renderiza todos os objetos que estão dentro da distância máxima configurad a, demandando tempo de processamento para o desenho de objetos que não serão exibidos na tela do dispositivo.

A correção da rotina de desenho de texto no OpenGL ES também representa uma melhoria. Um meio para a realização desta tarefa dá-se através da criação de um atlas de textura. Inicialmente, deve ser gerada em algum programa de edição de imagens, uma textura contendo todos os caracteres, números e símbolos que o texto irá suportar. De posse desta textura, deve-se criar e manter atualizado um VBO dinâmico com a posição e coordenadas da textura. O texto é gerado calculando novas coordenadas de textura para cada caractere da palavra que se pretende desenhar e então armazenar estas coordenadas em uma área contígua dentro do VBO. Cada caractere necessita de quatro vértices para ser desenhado, cujas coordenadas devem ser determinadas pela RAios-library e combinadas com as coordenadas correspondentes da textura armazenada no VBO.

Quanto ao aspecto de realidade da cena gerada pela RAios-library, é sugerida a criação de rotinas para o tratamento da oclusão de objetos. Atualmente, objetos que estão geograficamente mais distantes que outros são desenhados com o mesmo tamanho. Além disso, quanto um objeto está atrás de outro, os mesmos são desenhados de forma sobreposta, causando dificuldade na sua identificação.

O suporte à mudança de direção do dispositivo é outra melhoria que se entende necessária. Além de melhorar a interação entre a aplicação e o ambiente real, evita que o usuário seja obrigado a segurar o dispositivo em uma orientação específica.

A criação de um radar que possibilite ao usuário ter noção da posição dos objetos presentes ao seu redor é outra sugestão que agrega valor ao trabalho apresentado.

Quanto à organização do código fonte, é sugerido implementar a rotina que utiliza o *Location Manager* utilizando o conceito de *block* (ver seção 3.3.3), tornando o código mais legível e fácil de manter.

O registro de pontos de interesse através do uso de marcadores fixos também apresenta uma melhoria interessante. Através desta técnica, seria possível criar aplicações de realidade aumentada de melhor precisão para serem utilizadas em ambientes pequenos ou lugares de difícil captação do sinal do GPS.

Também é sugerida a utilização do giroscópio para detectar as mudanças de direção do dispositivo de forma mais precisa, pois os valores retornados pelo sensor referem-se apenas a aceleração instantânea do dispositivo. No caso deste trabalho, a utilização do giroscópio permitiria efetuar a translação dos objetos virtuais de forma mais suave, evitando grandes saltos de posição e a impressão de que o objeto está tremendo. A diferença causada pela utilização do giroscópio em uma aplicação de RA pode ser vista na referência (YOUTUBE, 2010), onde é apresentado um vídeo comparando a mesma aplicação com e sem o seu uso.

Outra sugestão interessante é criar e manter uma documentação para que os desenvolvedores possam consultar detalhes sobre a estrutura e organização da biblioteca.

Por último, é sugerida a implementação de um meio para obter as coordenadas dos pontos de interesse através de uma rede 3G e/ou *wireless*.

REFERÊNCIAS BIBLIOGRÁFICAS

ANDROID APPS. **See the world through virtual eyes with junaio**. [S.l.], 2011. Disponível em: < http://android-apps.com/apps/see-the-world-through-virtual-eyes-with-junaio>. Acesso em: 29 nov. 2011.

APPLE INC. **iOS overview.** [S.l.], 2010a. Disponível em: <http://developer.apple.com/library/ios/#referencelibrary/GettingStarted/URL_iPhone_OS_Overview/index.html>. Acesso em: 8 mar. 2011.

_____. **About iOS development.** [S.l.], 2010b. Disponível em: <http://developer.apple.com/library/ios/#documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/IPhoneOSOverview/IPhoneOSOverview.html#//apple_ref/doc/uid/TP40007898-CH4-SW1>. Acesso em: 8 mar. 2011.

_____. **Cocoa touch layer.** [S.l.], 2010c. Disponível em: <http://developer.apple.com/library/ios/#documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/iPhoneOSTechnologies/iPhoneOSTechnologies.html#//apple_ref/doc/uid/TP40007898-CH3-SW1>. Acesso em: 8 mar. 2011.

_____. **Media layer.** [S.l.], 2010d. Disponível em: <http://developer.apple.com/library/ios/#documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/MediaLayer/MediaLayer.html#//apple_ref/doc/uid/TP40007898-CH9-SW4>. Acesso em: 8 mar. 2011.

_____. **Core services layer.** [S.l.], 2010e. Disponível em: <http://developer.apple.com/library/ios/#documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/CoreServicesLayer/CoreServicesLayer.html#//apple_ref/doc/uid/TP40007898-CH10-SW5>. Acesso em: 8 mar. 2011.

_____. **Core os layer.** [S.l.], 2010f. Disponível em: <http://developer.apple.com/library/ios/#documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/CoreOSLayer/CoreOSLayer.html#//apple_ref/doc/uid/TP40007898-CH11-SW1>. Acesso em: 8 mar. 2011.

_____. **CLLocation class reference.** [S.l.], 2010g. Disponível em: <http:// http://developer.apple.com/library/ios/#DOCUMENTATION/CoreLocation/Reference/CLLocation_Class/CLLocation/CLLocation.html#//apple_ref/doc/uid/TP40007126>. Acesso em: 17 ago. 2011.

_____. **AV Foundation programming guide.** [S.l.], 2010h. Disponível em: <http://developer.apple.com/library/ios/#documentation/AudioVideo/Conceptual/AVFoundationPG/Articles/03_MediaCapture.html#//apple_ref/doc/uid/TP40010188-CH5-SW2>. Acesso em: 9 ago. 2011.

_____. **About OpenGL ES.** [S.l.], 2011a. Disponível em: <http://developer.apple.com/library/ios/#documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/Introduction/Introduction.html#//apple_ref/doc/uid/TP40008793>. Acesso em: 8 mar. 2011.

_____. **Event handling guide for iOS.** [S.l.], 2011b. Disponível em: <http://developer.apple.com/library/ios/#documentation/EventHandling/conceptual/EventHandlingiPhoneOS/MotionEvents/MotionEvents.html>. Acesso em: 17 ago. 2011.

_____. **Drawing with OpenGL ES.** [S.l.], 2011c. Disponível em: <http://developer.apple.com/library/ios/#documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/WorkingwithEAGLContexts/WorkingwithEAGLContexts.html#//apple_ref/doc/uid/TP40008793-CH103-SW1>. Acesso em: 2 ago. 2011.

_____. **Overview of dynamic libraries.** [S.l.], 2009. Disponível em: <http://developer.apple.com/library/mac/#documentation/DeveloperTools/Conceptual/DynamicLibraries/000-Introduction/Introduction.html#//apple_ref/doc/uid/TP40001908-SW1>. Acesso em: 14 mar. 2011.

AZUMA, Ronald T. A Survey of augmented reality. **Presence**: Teleoperators And Virtual Environments, [S.l.], v. 6, n. 4, p. 355-385. Aug. 1997. Disponível em: <http://www.cs.unc.edu/~azuma/ARpresence.pdf>. Acesso em: 25 fev. 2011.

AZUMA, Ronald T. et al. Recent advances in augmented reality. **IEEE Computer Graphics & Applications**, Los Alamitos, v. 21, n. 6, p. 34-47, Nov. 2001.

BAJURA, Michael; NEUMANN, Ulrich. Dynamic registration correction in video-based augmented reality systems. **IEEE Computer Graphics & Applications**, Los Alamitos, v. 15, n. 5, p. 53-60, Sept. 1995.

BIMBER, Oliver; RASKAR, Ramesh. **Spatial augmented reality merging real and virtual worlds**. Wellesley: A K Peters, 2005.

CAWOOD, Stephen; FIALA, Mark. **Augmented reality**: a practical guide. Raleigh: The Pragmatic Bookshelf, 2007.

CHRONS, Otto. **Home of accelerometer simulator**. [S.l.], 2010. Disponível em: <http://code.google.com/p/accelerometer-simulator/wiki/Home>. Acesso em: 15 out. 2011

DAINESE, Carlos A.; GARBIN, Tânia R.; KIRNER, Claudio. Sistema de realidade aumentada para desenvolvimento cognitivo da criança surda. In: SBC SYMPOSIUM ON VIRTUAL REALITY, 6., 2003, Ribeirão Preto. **Proceedings…** Ribeirão Preto: UNICOC, 2003. p. 273-282.

FOURNIER, Alain. Illumination problems in computer augmented reality. In: JOURNEE INRIA ANALYSE-SYNTHÈSE D'IMAGES, 1., 1994, Paris. **Actes Texte Impremé**… Paris: INRIA, 1994. p. 1-21. Disponível em: <http://www.cs.ubc.ca/labs/imager/tr/ps/fournier.19>. Acesso em: 3 mar. 2011.

GUIMARÃES, Marcelo P.; GNECCO, Bruno B.; DAMAZIO, Rodrigo. Ferramentas para desenvolvimento de aplicações de realidade virtual e aumentada. In: KIRNER, Claudio; SISCOUTTO, Robson. (Ed.). **Realidade virtual e aumentada**: conceitos, projetos e aplicações. Petrópolis: SBC, 2007. p. 108-128.

KIRNER, Claudio; TORI, Romero. Introdução à realidade virtual, realidade misturada e hiper-realidade. In: KIRNER, Claudio; TORI, Romero. (Ed.). **Realidade virtual:** conceitos, tecnologia e tendências. São Paulo: SENAC, 2004. p. 3-20.

LAMB, Philip. **ARToolkit**. Washington, 2010. Disponível em: <http://www.hitl.washington.edu/artoolkit/>. Acesso em: 14 mar. 2011.

LAYAR. **Layar for iPhone**. Amsterdam, 2010. Disponível em: <http://www.layar.com/download/iphone/>. Acesso em: 14 mar. 2011.

_____. **Layar´s visit to the US –** first stop: New York. Amsterdam, 2011. Disponível em: <http://www.layar.com/blog/2010/08/12/layars-visit-to-the-us-first-stop-new-york/>. Acesso em: 29 nov. 2011.

MARK, Dave; NUTTING, Jack; LAMARCHE, Jeff. **Beginning iphone 4 development**: exploring the ios sdk. New York: Apress, 2011.

MESA. **The mesa 3D graphics library**. [S.l.], 2011. Disponível em: <http://www.mesa3d.org/>. Acesso em: 2 ago. 2011.

METAIO INC. **Junaio**. [S.l.], 2011. Disponível em: <http://www.junaio.com/>. Acesso em: 19 set. 2011.

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

STACK EXCHANGE INC. **iPhone OpenGLES 2.0**. [S.l.], 2011. Disponível em: <http://stackoverflow.com/questions/6060578/iphone-opengles-2-0-text-texture-w-strange-border-not-stroke-issue>. Acesso em: 03 set. 2011.

VASSELAI, Gabriela T. **Um estudo sobre realidade aumentada para plataforma Android**. 2010. 103 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Centro de Ciências exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

VIMOV. **iSimulate:** documentation. [S.l.], 2011. Disponível em: <http://www.vimov.com/isimulate/documentation/>. Acesso em: 15 out. 2011.

YOUTUBE. **iPhone 4 augmented reality with Gyroscope**. [S.l.], 2010. Disponível em: <http://www.youtube.com/watch?v=VP4-wdMMLFo&feature=player_embedded>. Acesso em: 8 set. 2011.