UNIVERSIDADE FEDERAL DE GOIÁS – UFG

CÂMPUS CATALÃO – CAC

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO – DCC

Bacharelado em Ciência da Computação

Projeto Final de Curso

Utilização de Recursos dos Dispositivos Móveis da Apple Inc. para Auxiliar Pequenos e Médios Produtores Rurais

Autor: Adriano Mendonça Rocha

Orientador: Vaston Gonçalves da Costa

Catalão - 2012

Adriano Mendonça Rocha

Utilização de Recursos dos Dispositivos Móveis da Apple Inc. para Auxiliar Pequenos e Médios Produtores Rurais

Monografia apresentada ao Curso de
Bacharelado em Ciência da Computação da
Universidade Federal de Goiás Câmpus Catalão
como requisito parcial para obtenção do título de
Bacharel em Ciência da Computação

Área de Concentração: Computador e Sociedade
Orientador: Vaston Gonçalves da Costa

Catalão - 2012

Rocha, Adriano M.

Utilização de Recursos dos Dispositivos Móveis da Apple Inc. para Auxiliar Pequenos e Médios Produtores Rurais/Vaston Gonçalves da Costa- Catalão - 2012

Número de paginas: 70

Projeto Final de Curso (Bacharelado) Universidade Federal de Goiás, Câmpus Catalão, Curso de Bacharelado em Ciência da Computação, 2012.

Palavras-Chave: 1. Plataforma iOS. 2. Inovação Tecnológica. 3. Produtores Rurais

Adriano Mendonça Rocha

Utilização de Recursos dos Dispositivos Móveis da Apple Inc. para Auxiliar Pequenos e Médios Produtores Rurais

Monografia apresentada e aprovada em _____ de _____
Pela Banca Examinadora constituída pelos professores.

_____

Vaston Gonçalves da Costa – Presidente da Banca

_____

Roberto Mendes Finzi Neto

_____

Marcelo Henrique Stoppa

Dedico essa monografia aos meus pais que sempre me apoiaram em minhas escolhas, que acreditaram no meu potencial e que me prepararam para os desafios da vida.

AGRADECIMENTOS

Primeiramente agradeço aos meus pais pela força, apoio e compreensão que depositaram em mim durante toda a vida.

Ao meu orientador Prof. Dr. Vaston Gonçalves da Costa, por ter acreditado no meu potencial em desenvolver esse trabalho.

Ao Prof. Dr. Roberto Finzi por me encaminhar a este projeto.

À minha amiga Cassiana por ter me ajudado com o LaTeX e também por ter me ajudado na revisão do texto.

Ao meu amigo Leandro Resende Mundim, que me ajudou durante o final dessa caminhada.

Ao meu amigo Cleriston Aparecido Dantas por ter me ajudado com as interfaces gráficas de usuário do prótotipo do aplicativo implementado nesse trabalho.

À minha namorada Carolina Prata por ter me ajudado a revisar todo o texto deste trabalho e também pelo o apoio psicológico durante a elaboração da monografia.

Aos meus colegas e amigos(Azamor, Danilo, Diego, Jayme, Leandro, Lucas, Marcelo, Samuel, Thiago) que estiveram comigo durante o período da graduação, sendo estudando ou simplesmente nos momentos de descontração.

Ao Ministério da Educação pela bolsa de pesquisa, através do Proext 2011.

E a todos os professores que contribuíram na minha formação como cientista da computação.

"A fé é querer ignorar tudo aquilo que é verdade."
Friedrich Nietzsche.

RESUMO

Rocha, A. M. Utilização de Recursos dos Dispositivos Móveis da Apple Inc. para Auxiliar Pequenos e Médios Produtores Rurais. Curso de Ciência da Computação, Câmpus Catalão, UFG, Catalão, Brasil, 2012, 70p.

A tecnologia está evoluindo em um ritmo tão acelerado que é difícil acompanhar seus avanços. Atualmente, os celulares possuem tantas funcionalidades que às vezes o próprio usuário não tem noção do quanto estas podem auxiliá-lo nas tarefas do cotidiano. Quanto mais longe as pessoas estão dos centros de produção, menos conhecem essa tecnologia, o que agrava no caso dos moradores da zona rural. Com a evolução dos sistemas operacionais para celulares, hoje em dia, pode-se desenvolver aplicativos com maior facilidade para esses dispositivos, aproveitando ao máximo seus recursos. Pensando nisso, esse trabalho tem como objetivo aproximar os produtores rurais da tecnologia de uma forma prática, através de um protótipo de um aplicativo desenvolvido na plataforma iOS (iPhone *Operating System*) que irá auxiliá-los nas atividades agrícolas, bem como reduzir seus gastos com mão de obra qualificada e otimizar sua produção, aproveitando o máximo a área de plantio.

Palavras-Chaves: Plataforma iOS, Inovação Tecnológica, Produtores Rurais

Sumário

1 Introdução **1**

2 Fundamentação Teórica **4**

2.1 Conceitos 4

2.2 Definições 5

2.2.1 GPS 6

2.2.2 Teorema de Herão 7

3 Arquitetura Apple iOS **8**

3.1 Arquitetura do iOS 8

3.1.1 Camada Cocoa Touch 9

3.1.2 Camada *Media* 11

3.1.3 Camada *Core Services* 13

3.1.4 Camada *Core OS* 13

3.2 Frameworks do iOS 14

4 Linguagem Objetive-C **18**

4.1 Objetive-C e Linguagem C 18

4.2 Classes 18

4.2.1 Objetos 19

4.2.2 Id 19

4.2.3 Tipagem Dinâmica 20

4.3 Métodos, Mensagem e Propriedades 21

4.3.1 Métodos 21

4.3.2 Sintaxe da Mensagem 22

4.3.3 Declaração de propriedades 23

4.4 Gerenciamento de Memória, Strings e Protocolos 24

4.4.1 Gerenciamento de Memória 24

4.4.2 Strings 25

4.4.3 Protocolos 25

5 iOS SDK **26**
5.1 iOS SDK . . . 26
5.2 Xcode . . . 27
5.3 Interface Builder . . . 30
5.4 Instruments . . . 31

6 Aplicativo Proposto **32**
6.1 Objetivos do aplicativo . . . 32
6.2 Model View Controller . . . 33
6.2.1 Model . . . 33
6.2.2 View . . . 34
6.2.3 Controller . . . 34
6.2.4 Ligação dos Objetos da Interface Gráfica de Usuário ao Código-Fonte 34
6.3 Algoritmo que Calcula a Área do Terreno . . . 40
6.4 Código do Aplicativo . . . 44
6.4.1 SegundaTelaiPad.m . . . 45
6.5 Diagramas de Classe . . . 53
6.6 Resultados e Testes . . . 55

7 Manual do Aplicativo **58**
7.1 Manual do aplicativo para o iPad . . . 58
7.2 Manual do aplicativo para o iPhone . . . 62

8 Conclusão **65**

Referências **67**

Lista de Figuras

3.1 Aplicações em camadas em cima do iOS (APPLE, 2008) 8
3.2 Camadas do iOS (YATES II, 2010) 9
3.3 Botão *Home* 10

4.1 Sintaxe para declaração de uma classe 19
4.2 Sintaxe para declaração de um método 21

5.1 Janela de um projeto no *Xcode* 29
5.2 Executando um projeto no *Xcode* (APPLE, 2010d) 29
5.3 Construindo uma interface gráfica usando o *Interface Builder* (APPLE, 2010d) 30
5.4 Análise da aplicação usando *Instruments* (APPLE, 2010d) 31

6.1 Modelo de projeto *Model View Controller* (APPLE, 2010a) 33
6.2 Criando um novo arquivo no Xcode 35
6.3 Criando uma classe ViewController 36
6.4 Arquivo de inteface da classe 36
6.5 Declarações de IBOutlets 37
6.6 Declarações de IBActions 37
6.7 Opção View Controller 37
6.8 Opção *Custom Class* 38
6.9 Opção Outlets e Actions 39
6.10 Ligação de um *label* 39
6.11 Ligação de um *Text Field* 40
6.12 Ligação de um método IBAction a um botão na interface gráfica 40
6.13 Coleta de pontos através do GPS 41
6.14 Ligação do ponto 1 ao 2. 41
6.15 Ligação do ponto 2 ao 3. 41
6.16 Ligação do ponto 3 ao 1. 42
6.17 Ligação do ponto 1 ao 3. 42
6.18 Ligação do ponto 3 ao 4. 42
6.19 Ligação do ponto 4 ao 1. 43

6.20 Ligação do ponto 1 ao 4. 43
6.21 Ligação do ponto 4 ao 5. 43
6.22 Ligação do ponto 5 ao 1. 44
6.23 Área do terreno 44
6.24 Diagrama de classe do aplicativo para o iPad 54
6.25 Diagrama de classe do aplicativo para o iPhone 55

7.1 Tela inicial do aplicativo 58
7.2 Segunda tela do aplicativo 59
7.3 GPS desligado 60
7.4 Mensagem informando ao usuário que as coordenadas foram salvas 61
7.5 Mensagem informando ao usuário que entre com pelo menos três pontos . . 61
7.6 Mensagem informando ao usuário a área em metros quadrado e a quantidade de covas e sementes. 62
7.7 Tela inicial do aplicativo 63
7.8 Tela Ponto GPS 63
7.9 Tela de entrada manual de coordenadas 64

Lista de Tabelas

3.1 Frameworks Parte 1 15
3.2 Frameworks Parte 2 16
3.3 Frameworks Parte 3 17

6.1 Resultados dos testes de precisão do GPS 56

Lista de Algoritmos

4.1 Tipos *strong* e *weak*. 20
4.2 Estrutura de dados id. 20
4.3 Expressão de uma mensagem. 22
4.4 Método obterNome. 22
4.5 Parâmetro de um método. 22
4.6 Aninhamento de mensagens. 22
4.7 Método de classe. 23
4.8 Declarações de propriedades. 23
4.9 Diretiva @synthesize. 23
4.10 Gerenciamento de Memória. 24
4.11 String em Objetive-C. 25
4.12 Protocolo em Objetive-C. 25
4.13 Declaração de um protocolo. 25
6.1 Método locationManager. 45
6.2 Método onRoff. 46
6.3 Método salvarPontoGps. 47
6.4 Método salvarPontoManual. 48
6.5 Método grausRadiano. 50
6.6 Método calculaDistancia. 50
6.7 Método calculaAreaSementes. 52

Capítulo 1

Introdução

Existe uma troca de informações crescente entre o mundo físico e o digital, diante das diversas possibilidades de pesquisa, lazer, relacionamento e outros serviços, que acabam ocasionando interferências profundas na forma com que as mais variadas atividades se processam (MUNDIM et al., 2012).

Os componentes denominados tecnológicos tais como, computadores, celulares, Internet, mp3, câmeras digitais, dentre outros, têm sua origem de forma individual e separada. Ou seja, o celular nasceu para ser um telefone móvel, uma câmera apenas registrava imagens e assim por diante. Porém, estes aparelhos hoje têm cada vez mais acentuada a convergência multitarefa, onde cada novo aparelho pode ser um pouco de tudo. O computador é cada vez mais potente e menor, ao contrário dos primeiros que eram gigantescos e com pouca capacidade de processamento. O celular, dentre as tecnologias atuais, é o que mais agrega funções: acessa à Internet, fotografa, filma, é receptor de TV, tem jogos, música, possui GPS (*Global Positioning System*) e muitos outros serviços.

É perceptível o constante crescimento das diversas formas de tecnologia e nas mais diferentes áreas. Os avanços tecnológicos se encontram presentes em nosso cotidiano de tal modo que se tornam corriqueiros e de extrema necessidade. O mundo físico é reproduzido digitalmente, e assim, aumentam as formas dos serviços serem realizados.

O avanço das novas tecnologias interfere diretamente nas diversas possibilidades de lazer, entretenimento, pesquisa, relacionamento e trabalho. Muitas são as possibilidades de comunicação e acesso à informação disponíveis.

As tecnologias permitiram e ainda permitem mudanças cada vez mais profundas, uma vez que a sua inserção na sociedade, permanecem praticamente inexploradas pela inércia da cultura tradicional, pelo medo e pelos valores consolidados. Assim, é notório o distanciamento entre as possibilidades e a realidade, pois os usuários avançam imersos em inúmeras dúvidas, e bem mais lentamente que a exposição ao "novo" tecnológico. Paralelamente, o teórico avança muito mais do que a prática, de modo que há um distanciamento cada vez maior entre o desejo e a ação, entre a ferramenta e sua utilidade.

Neste contexto, o Brasil não tem problemas tecnológicos, e sim de desigualdade social. De modo geral, esta interatividade tem muito a ver com poder de compra, com educação de qualidade, com cultura empreendedora e, principalmente, com a capacidade de uso dessas novas tecnologias.

Apesar dos grandes avanços tecnológicos presentes em nossas vidas, poucas pessoas efetivamente fazem uso desses recursos para melhoria de sua qualidade de trabalho e vida. Neste contexto se enquadram tanto moradores da zona urbana quanto rural. Agrava-se, contudo, na zona rural pois se acresce a tudo isso a distância que a separa do centro de produção e utilização destes recursos.

Um exemplo está na forma com que as pequenas comunidades rurais lidam com o plantio de produtos agrícolas. O pequeno produtor precisa dimensionar a quantidade de sementes e insumos necessários ao bom plantio da área demarcada. Quanto melhor a precisão, menores os custos com a produção e maiores os resultados da colheita. Todos os cálculos que determinam os valores precisos se baseiam no tamanho da área delimitada, geralmente em hectares. Um pequeno produtor muitas vezes não possui recursos financeiros necessários para contratar um agrimensor, para lhe suprir com a tal informação.

A contratação de dois técnicos, uma para medir a área e outro para determinar a quantidade de sementes e insumos agrícolas, comprometeria demasiado os recursos financeiros limitados deste produtor.

Para o cálculo da área, agrimensores já fazem uso da tecnologia GPS, um sistema de navegação por satélites que retorna as coordenadas geográficas do ponto onde um aparelho receptor móvel se encontra. Esta é uma das tecnologias amplamente utilizadas na sociedade, principalmente para determinar rotas de trânsito em cidades, a qual ainda não alcançou os pequenos produtores brasileiros. Devido à popularização desta tecnologia, receptores GPS estão cada vez mais acessíveis e podem ser embarcados em vários elementos tais como carros, barcos e celulares. Estes últimos, de fácil aquisição a preços módicos em alguns modelos.

Este trabalho tem por objetivo apresentar um protótipo de uma sistema que auxilie os pequenos produtores rurais em sua lida com o plantio, que trate de maneira mais precisa a distribuição de insumos e sementes, sem contudo gerar gastos extras.

Optou-se por embarcar tal protótipo em aparelhos móveis que também sejam receptores GPS. Dentre as várias opções existentes, os aparelhos da família Apple Inc., por possuirem boa documentação, boa precisão nas coordenadas apresentadas e por possuirem boa autonomia da bateria. Esta útima característica é julgada como fundamental, pois o sistema será utilizado em campo aberto, distante de fontes de alimentação de energia.

Este trabalho está estruturado em oito capítulos. Neste capítulo é apresentada a introdução.

No segundo capítulo é descrita a fundamentação teórica e os principais termos utilizados nesse trabalho, com o objetivo de auxiliar os leitores que não tem muito contato com esses conceitos.

O terceiro capítulo aborda a arquitetura do iOS, descrevendo suas camadas e seus principais *frameworks*. O objetivo desse capítulo é entender como funciona o iOS e quais são as principais funcionalidades oferecidas pelo mesmo.

No quarto capítulo é abordada a linguagem de programação Objetive-C. Esse capítulo busca introduzir essa linguagem, descrevendo suas principais características. Também serão analisados vários trechos de código em Objetive-C, estudando sua sintaxe.

O quinto capítulo descreve o iOS SDK (*Software Development Kit*), onde é abordado as principais ferramentas que auxiliam o desenvolvedor durante o processo de desenvolvimento de aplicativos para a plataforma iOS.

No sexto capítulo é abordado o protótipo do aplicativo proposto nesse trabalho, descrevendo o código-fonte dos seus principais métodos.

O sétimo capítulo aborda o manual do aplicativo, onde suas funcionalidades são explicadas, mostrando como o aplicativo funciona.

No oitavo capítulo são abordadas as conclusões do autor sobre o projeto. Também serão apresentados os trabalhos futuros que podem ser desenvolvidos nas próximas versões do protótipo do aplicativo. Por fim, são descritas as dificuldades encontradas para a realização do projeto.

Capítulo 2

Fundamentação Teórica

Este capítulo tem como objetivo apresentar as definições dos termos utilizados nesse trabalho.

2.1 Conceitos

Existem vários dispositivos móveis no mercado com diferentes configurações de *hardware* e sistema operacional. Nesse trabalho a escolha do dispositivo é fundamental para se obter bons resultados. O GPS do dispositivo deve oferecer uma boa precisão da localização do usuário. O dispositivo também deve oferecer uma boa autonomia da bateria, pois quanto mais preciso for o GPS, mais bateria é necessário. Os dispositivos móveis da Apple Inc. oferecem todos esses recursos, suprindo todos os requisitos necessários para o desenvolvimento desse trabalho.

A plataforma iOS oferece várias funcionalidades necessárias para o bom desempenho do aplicativo. Além de possuir uma rica documentação disponibilizada pela própria Apple Inc. que explica toda tecnologia necessária para desenvolver aplicativos para o iOS.

Esta plataforma oferecem vários recursos, como banco de dados embarcado, manipulação do GPS, e recursos gráficos simples e intuitivos. Estes facilitam a implementação de aplicativos e possibilitam um acervo de funcionalidades maior para o aplicativo.

Objetive-C é a linguagem padrão para se desenvolver aplicativos para a plataforma iOS. Segundo Lee (2010), Objective-C é uma linguagem de programação orientada a objetos, projetada para permitir uma programação simples e sofisticada. Tal linguagem é definida como um conjunto pequeno, mas poderoso de extensões para o padrão da linguagem ANSI C.

O iOS SDK contém as ferramentas necessárias para se desenvolver aplicativos para o iOS. Ele também contém arquivos *header*, amostras de código e documentação para as tecnologias dessa plataforma.

2.2 Definições

Esta seção tem como objetivo definir os principais termos utilizados nesse trabalho, com intuito de auxiliar os leitores que não estão familiarizados com estes conceitos.

Definição 2.1 *(Tecnologia) - Consiste nos conhecimentos utilizados na indústria e agricultura referentes às causas dos fenômenos físicos e sociais e a sua aplicação à produção, além dos conhecimentos relativos à operação rotineira da produção (MANSFIELD, 1985). Segundo Fleury (1990), tecnologia é um pacote de informações organizadas, de diferentes tipos (científicas, empíricas, etc), provenientes de várias fontes (descobertas científicas, patentes, livros, manuais, desenhos...), obtidas através de diferentes métodos (pesquisa, desenvolvimento, cópia, espionagem...), utilizado na produção de bens e serviços. Para Gonçalves (1994), consiste em um conjunto integrado de conhecimentos, técnicas, ferramentas e procedimentos de trabalho aplicados na produção econômica de bens e serviços. O GPS e plataforma iOS são exemplos de tecnologias utilizadas nesse trabalho.*

Definição 2.2 *(Mac OS X) - Sistema operacional baseado no kernel OpenDarwin(Unix), desenvolvido, fabricado e vendido pela Apple Inc., que é destinado apenas aos produtos Apple, principalmente aos computadores Mac (EVANS BILL; HOFFMAN, 2002).*

Definição 2.3 *(Ambiente Cocoa) - É uma interface de programação de aplicativos para dispositivos Apple (PISKE OTAVIO RODOLFO; SEIDEL, 2006).*

Definição 2.4 *(Framework) - É uma aplicação reusável e semi-completa que pode ser especializada para produzir aplicações personalizadas (FAYAD M.E; SCHIMIDT, 1999). São exemplos de frameworks utilizados nesse trabalho: CoreLocation que contém as interfaces para determinar a localização do usuário e UIKit (User Interface Kit) que contém classes e métodos associadas a interface gráfica de usuário.*

Definição 2.5 *(Objeto) - É um elemento computacional que representa, no domínio da solução, alguma entidade (abstrata ou concreta) do domínio de interesse do problema sob análise.*

Definição 2.6 *(Abstração) - Capacidade de representar um objeto e o comportamento esperado em um único trecho de código (GAÚCHO, 2003).*

Definição 2.7 *(Orientação a Objetos) - Paradigma de programação no qual um programa é estruturado em objetos que representam os objetos do mundo real por meio de abstração (MATSUBARA, 2004).*

Definição 2.8 *(Método Getter) - Convenção de atribuição de nome para o método que obtem variáveis privadas (DEITEL HARVEY M.; DEITEL, 2005). Utilizado para recuperar alguma informação, geralmente implementado para trazer informação de algum atributo, sem ter que utilizar o atributo explicitamente.*

Definição 2.9 *(Método Setter) - Convenção de atribuição de nome para o método que configura variáveis privadas (DEITEL HARVEY M.; DEITEL, 2005). Utilizado para ligar um valor dentro de um objeto, ou de uma variável.*

Definição 2.10 *(Interface) - É uma característica que permite isolar do mundo exterior detalhes de implementação de um componente de software. Para Oracle (2012), é um contrato entre uma classe e o mundo exterior, onde a classe implementa o comportamento descrito pela interface. Um exemplo clássico de utilização de interfaces é o do sistema operacional que, através de uma interface de programação de aplicativos, permite que os programas utilizem os recursos do sistema sem que os seus detalhes de implementação sejam conhecidos pelo programador. Este esquema isola e protege o sistema operacional de eventuais erros cometidos pela aplicação.*

Definição 2.11 *(Streaming) - Técnica de transferência de dados, onde os dados são processados em fluxo contínuo de forma a permitir que o conteúdo seja reproduzido antes de ser completamente enviado (TEIXEIRA, 2006) e (CLEMENTE, 2006).*

Definição 2.12 *(Renderização) - É o processo pelo qual se obtém um produto final de um processamento digital (RODRIGUES MARIA ANDRÉIA FORMICO; BARBOSA, 2006).*

Definição 2.13 *(Thread) - É um fluxo de execução de um processo (TANENBAUM, 2003) e (OLIVEIRA RÔMULO SILVA; CARISSIMI, 2001).*

Definição 2.14 *(Longitude) - É a distância ao meridiano de Greenwich medida ao longo do Equador. Esta distância mede-se em graus, podendo variar entre 0º e 180º para Leste ou para Oeste (TIMBÓ, 2007).*

Definição 2.15 *(Latitude) - É a distância ao Equador medida ao longo do meridiano de Greenwich. Esta distância mede-se em graus, podendo variar entre 0º e 90º para Norte ou para Sul (TIMBÓ, 2007).*

2.2.1 GPS

O sistema de posicionamento global é um sistema de navegação por satélite que fornece a um aparelho receptor móvel determinar sua posição e tempo em qualquer lugar da superfície terrestre e a qualquer hora utilizando dados de apenas quatro satélites. O

sistema pode ter um número ilimitado de usuários simultaneamente em qualquer parte do mundo (ALVES, 1996).

Segundo Anholetto e Fernandes (2005), o princípio de navegação por satélites se baseia no princípio da triangulação, a qual consiste na transmissão de sinais e dados das posições dos satélites GPS em relação a um sistema de coordenadas. O receptor mede o tempo de transmissão do sinal, o que permite calcular a distância entre o usuário e os satélites GPS, e decifra os dados. Se o relógio do receptor estiver sincronizado com os relógios dos satélites GPS, as medidas das distâncias de três diferentes satélites GPS, em posição conhecida, permitirão ao usuário calcular a sua posição. Se o relógio do receptor não estiver sincronizado com os relógios dos satélites, serão necessários quatro satélites GPS, sendo o quarto para determinar o desvio do relógio, uma quantidade desconhecida. As medidas da distância com relógio impreciso são chamadas de pseudo-distância.

AGPS (*Assisted* GPS)

É uma evolução do GPS, onde as funcionalidades deste são distribuídas entre um receptor de referência, o servidor de localização e o cliente. O posicionamento GPS é inicialmente feito através do usuário, que envia a informação parcial para o servidor de localização, o qual executa os cálculos restantes pela interação com o receptor de referência GPS.

O AGPS reduz o tempo que um aparelho com GPS leva para localizar a sua posição atual, conhecido como TTFF (Tempo de Localização Inicial). Como resultado apresenta uma precisão maior no cálculo do posicionamento, porém o custo é mais alto devido à necessidade do servidor de localização. Este sistema é mais útil em áreas urbanas, ajudando a manter o sinal estável até mesmo em regiões cercadas por edifícios.

2.2.2 Teorema de Herão

O Teorema de Herão é utilizado para determinar a área de qualquer triângulo quando se conhece apenas as medidas dos seus lados. Dado um triângulo cujos lados medem a, b e c, a sua área é igual $\sqrt{s(s-a)(s-b)(s-c)}$, onde $s = \frac{a+b+c}{2}$ (SÁ; ROCHA, 2010).

O algoritmo que calcula a área do terreno desenvolvido nesse trabalho utiliza esse teorema para calcular a área dos triângulos formados através dos pontos obtidos por meio do GPS do dispositivo.

Capítulo 3

Arquitetura Apple iOS

Este capítulo tem por objetivo apresentar um breve estudo sobre a arquitetura do sistema operacional iOS que é executado nos dispositivos móveis da Apple Inc.: iPhone, iPad e iPod touch, abordando os principais *frameworks* presentes nas camadas que compõem essa arquitetura.

É muito importante entender como é organizada a arquitetura do iOS antes mesmo de começar a desenvolver aplicativos para a mesma, pois os *frameworks* das camadas do iOS oferecem várias funcionalidades que podem ser utilizadas nos aplicativos.

3.1 Arquitetura do iOS

A arquitetura do iOS é formada por quatro camadas, sendo que cada uma destas oferece um conjunto de *frameworks* que podem ser utilizados durante o desenvolvimento de aplicativos para os dispositivos baseados no iOS.

A arquitetura do iOS é semelhante à arquitetura básica encontrada no Mac OS X (ANVAARI; JANSEN, 2010) e (GRISSOM, 2008). No nível mais alto, o iOS atua como intermediário entre o *hardware* subjacente e os aplicativos que aparecem na tela, como mostrado na Figura 3.1.

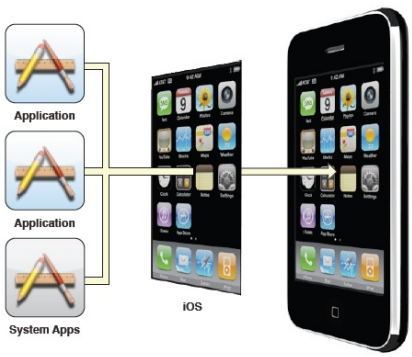

Figura 3.1: Aplicações em camadas em cima do iOS (APPLE, 2008)

Os aplicativos desenvolvidos para o iOS raramente se comunicam diretamente com o *hardware* do dispositivo, em vez disso, os aplicativos se comunicam com o *hardware* por meio de um conjunto de interfaces de sistema bem definidas que protegem o aplicativo de alteração de *hardware* (APPLE, 2008).

O iOS foi projetado para atender às necessidades de um ambiente móvel. Como o iOS é baseado no Mac OS X, então pode-se encontrar muitas tecnologias familiares, mas também é possível perceber tecnologias que só estão disponíveis na plataforma iOS (GONZALEZ-SANCHEZ; CHAVEZ-ECHEAGARAY, 2010) e (ROGERS, 2009).

A arquitetura do iOS é formada pelas camadas: *Core OS, Core Services, Media e Cocoa Touch* (YATES II, 2010), como mostrado na Figura 3.2.

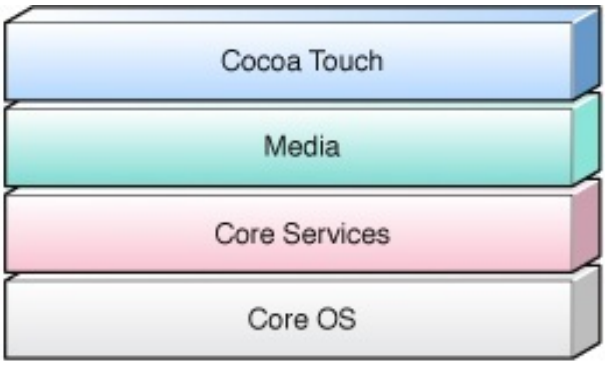

Figura 3.2: Camadas do iOS (YATES II, 2010)

Nas camadas superiores estão as tecnologias e serviços mais sofisticados. Sempre que possível, deve-se olhar primeiro os serviços das camadas superiores, pois nestas estão os *frameworks* que fornecem abstração orientada a objetos das camadas de níveis inferiores. Estas abstrações geralmente facilitam o processo de escrita do código-fonte, além de reduzir a quantidade de linhas do programa. Outra vantagem é o encapsulamento de características complexas, tais como *threads*.

Nas camadas inferiores do sistema estão os serviços fundamentais e as tecnologias das quais todos os aplicativos dependem. Embora as tecnologias de níveis superiores resumam as tecnologias de níveis inferiores, pode-se usar estas últimas que não estão presentes nas camadas superiores (APPLE, 2010d).

3.1.1 Camada Cocoa Touch

Os principais *frameworks* para a construção de aplicações são encontrados na camada *Cocoa Touch*. Esta camada define a infra-estrutura para as tecnologias fundamentais, tais

como multitarefa, serviço de notificação Apple *push* e diversos serviços de alto nível do sistema.

Ao projetar um aplicativo, deve-se investigar esta camada e suas tecnologias observando se tais atendem às suas necessidades. Nessa camada as principais tecnologias disponíveis são: multitarefa, proteção de dados e serviço de notificação Apple *push*.

Quando o usuário pressiona o botão *Home* do iPhone, por exemplo, o aplicativo não termina, em vez disso, ele muda para um contexto de execução em segundo plano. Essa característica da multitarefa é muito importante para preservar a vida da bateria, pois a maioria dos aplicativos ficam suspensos pelo sistema quando entram em segundo plano. A aplicação suspensa permanece na memória, mas nenhum código é executado. Esse comportamento permite que um aplicativo retome rapidamente quando é reiniciado, sem consumo de bateria no mesmo período (APPLE, 2010d). A seta na Figura 3.3 indica o botão *Home* no iPhone.

Figura 3.3: Botão *Home*

A proteção de dados permite que os aplicativos que trabalham com dados confidenciais do usuário aproveitem o sistema de criptografia disponível em alguns dispositivos. Quando

um aplicativo designa um arquivo específico como sendo protegido, o sistema armazena o arquivo no disco em um formato criptografado. Enquanto o dispositivo estiver bloqueado, o conteúdo do arquivo é inacessível para o aplicativo e para todos os intrusos em potencial. No entanto, quando o aparelho é desbloqueado pelo usuário, uma chave de decodificação é criada para permitir que o aplicativo acesse esse arquivo.

O serviço de notificação Apple *push* fornece uma maneira de alertar os usuários sobre novas informações, mesmo quando o aplicativo não está ativo. Por meio deste serviço, pode-se adicionar notificações de texto, adicionar um emblema no ícone do aplicativo, ou acionar alertas sonoros nos dispositivos do usuário. Essas mensagens fazem com que os usuários saibam que eles deveriam abrir o aplicativo para receber novas informações.

3.1.2 Camada *Media*

A camada *Media* contém as tecnologias de gráfico, áudio e vídeo. As tecnologias nesta camada foram projetadas para tornar mais fácil a implementação de aplicativos multimídia.

Os *frameworks* de nível superior oferecem tecnologias que facilitam a criação de gráficos e animações, enquanto os *frameworks* de nível inferior permitem o acesso às ferramentas fundamentais que podem ser utilizadas para criar aplicativos mais robustos e complexos (APPLE, 2008).

Tecnologias de Gráficos

A camada *Media* disponibiliza o *frameworks* UIKit que oferece várias tecnologias de gráficos e animações. Se o aplicativo exigir uma animação simples, pode-se utilizar o UIKit para fazer isso, se houver situações onde é preciso ir além dos gráficos simples, pode-se usar as seguintes tecnologias:

- *Core Graphics* lida com renderização de vetores com base em imagens 2D.
- *Core Animation* fornece suporte avançado para animar visualizações e outros conteúdos.
- *OpenGL ES* oferece suporte para renderização 2D e 3D acelerados por *hardware* usando interfaces.
- Texto *Core* fornece um *layout* de texto sofisticado e motor de renderização.
- *Image I/O* fornece interfaces de leitura e escrita para maioria dos formatos de imagem.

- O *framework Assets Library* fornece acesso a fotos e vídeos na biblioteca de imagens e vídeos do usuário.

Tecnologias de Áudio

As tecnologias de áudio disponíveis na camada *Media* fornecem uma capacidade de reproduzir e gravar áudio de alta qualidade, além de dispor de recursos de vibração em determinados dispositivos.

A camada *Media* oferece várias maneiras para reproduzir e gravar conteúdo de áudio. Os *frameworks* na lista a seguir estão ordenados do alto para baixo nível. Ao escolher uma tecnologia de áudio, deve-se ter em mente que os *frameworks* de níveis superiores são mais fáceis de usar e são geralmente preferidos. Já os de níveis inferiores oferecem mais flexibilidade e controle, mas exigem mais trabalho por parte do desenvolvedor.

- O *framework Media Player* fornece acesso fácil à biblioteca do iTunes do usuário e suporte para a reprodução de faixas e *playlists*.

- O *framework AV Foundation* fornece um conjunto de interfaces escritas em Objetive-C para o gerenciamento de reprodução e gravação de áudio.

- Os *frameworks Core Audio* oferecem interfaces simples e sofisticadas para reproduzir e gravar conteúdo de áudio. Pode-se usar essas interfaces para reproduzir sons de alerta do sistema, provocar vibrações no dispositivo e reproduzir conteúdo armazenado no dispositivo ou *streaming* de áudio.

Tecnologias de Vídeo

Além das tecnologias de áudio, a camada *Media* oferece tecnologias para reproduzir e gravar conteúdo baseado em vídeo. Em dispositivos com o *hardware* de vídeo apropriado, pode-se usar essas tecnologias para capturar e incorporar vídeos na aplicação. Os *frameworks* na lista a seguir estão ordenados do alto para baixo nível.

- A classe *UI ImagePickerController* presente no *framework* UIKit fornece uma interface padrão para gravação de vídeo em dispositivos com câmera.

- O *framework Media Player* fornece um conjunto de interfaces que podem ser usadas para apresentar filmes completos ou parciais no aplicativo.

- O *framework AV Foundation* fornece um conjunto de interfaces em Objetive-C para o gerenciamento de captura e reprodução de filmes.

- *Core Media* descreve os tipos de dados de baixo nível usados pelos *frameworks* de níveis superiores e oferece interfaces de baixo nível para a manipulação de mídias.

As tecnologias de vídeos no iOS suportam a reprodução de arquivos de filme com as extensões de arquivo mov, mp4, m4v e 3gp.

3.1.3 Camada *Core Services*

A camada *Core Services* contém os serviços fundamentais do sistema que todos os aplicativos utilizam. Mesmo não utilizando esses serviços diretamente, muitas partes do sistema são construídas beseadas neles (APPLE, 2010d). As principais tecnologias disponíveis na camada *Core Services* são: *grand central dispatch*, *in-app purchase* e *SQLite*.

Grand central dispatch é uma tecnologia utilizada no gerenciamento de execução de tarefas no aplicativo. Essa tecnologia combina um modelo de programação assíncrona, com um núcleo altamente otimizado para oferecer uma alternativa mais eficiente para *threading*. Ele também fornece alternativas para muitos tipos de tarefas de baixo nível, como ler e escrever em arquivos, implementação de temporizadores e monitoramento de sinais e eventos do processo.

In-app dispatch é uma tecnologia que pode ser utilizada para vender conteúdos e serviços dentro das aplicações. Este recurso é implementado usando a estrutura *Kit Store*, que fornece a infra-estrutura necessária para processar as transações financeiras usando a conta do iTunes do usuário.

A biblioteca *SQLite* permite incorporar um banco de dados *SQL* leve na aplicação, sem ter que executar um processo de servidor de banco de dados remoto. O desenvolvedor pode criar arquivos de banco de dados local e gerenciar as tabelas e registros nestes arquivos a partir do seu aplicativo. Essa biblioteca é otimizada para fornecer acesso rápido aos registros do banco de dados.

3.1.4 Camada *Core OS*

A camada *Core OS* contém características de baixo nível que foram utilizadas na implementação de outras tecnologias. Em situações onde é preciso lidar explicitamente com segurança ou comunicação com acessório de *hardware* externo, isto pode ser feito utilizando os *frameworks* dessa camada.

O *framework Accelerate* é uma tecnologia presente na camada *Core OS* que contém interfaces para a realização de cálculos matemáticos. A vantagem de se usar esse *framework* são as interfaces otimizadas para todas as configurações de *hardware* presentes em dispositivos baseados no iOS. Portanto, o desenvolvedor pode escrever seu código uma vez e ter certeza que ele será executado de forma eficiente em todos os dispositivos móveis da Apple Inc..

O *framework External Accessory* é uma tecnologia presente na camada *Core OS* que fornece interfaces de comunicação com acessórios de *hardware* conectados a um dispositivo

baseado no iOS. Os acessórios podem ser conectados através de um conector *dock* de 30 pinos, ou através do *Bluetooth*. Por meio das interfaces desse *framework* pode-se manipular o acessório diretamente usando os comandos que ele suporta.

No nível do sistema está presente o ambiente *kernel*, *drivers* e interfaces de baixo nível do sistema operacional UNIX. A biblioteca *kernel* é responsável por todos os aspectos do sistema operacional. Ela gerencia o sistema de memória virtual, *threads*, sistema de arquivos, rede e comunicação entre processos. Os *drivers* nesta camada também fornecem a interface entre o *hardware* disponível e os *frameworks* do sistema. O iOS fornece um conjunto de interfaces para acessar muitas características de baixo nível do sistema operacional. O aplicativo acessa esses recursos por meio da biblioteca *Libsystem*. As interfaces são baseadas em C e fornecem suporte para:

- *Threading*
- *Networking*
- Acesso ao sistema de arquivos
- *Standard I / O*
- Serviços DNS
- Alocação de memória
- Cálculos matemáticos

3.2 Frameworks do iOS

Esta seção contém informações sobre os *frameworks* do iOS. Esses *frameworks* fornecem as interfaces necessárias para escrever aplicativos para essa plataforma. As Tabelas 3.1, 3.2 e 3.3 listam os principais prefixos dos *frameworks* usados pelas classes, métodos, funções, tipos, ou constantes.

Tabela 3.1: Frameworks Parte 1

Nome	**Prefixos**	**Descrição**
Accelerate.framework	cblas, vDSP	Contém funções matemáticas aceleradas e funções DSP.
AddressBook.framework	AB	Contém funções para acessar contatos do usuário diretamente no banco de dados.
AssetsLibrary.framework	AL	Contém classes para acessar fotos e vídeos do usuário.
AudioToolbox.framework	AU, Audio	Contém as interfaces para lidar com fluxo de dados de áudio e com reprodução e gravação de áudio.
AudioUnit.framework	AU, Audio	Contém as interfaces para carregar e reproduzir o áudio das unidades.
AVFoundation.framework	AV	Contém interfaces em Objective-C para reproduzir e gravar áudio e vídeo.
CFNetwork.framework	CF	Contém interfaces para acesso à rede Wi-Fi e celular.
CoreData.framework	NS	Contém interfaces para o gerenciamento de modelo de dados da aplicação.
CoreFoundation.framework	CF	Fornece serviços fundamentais de *software*, incluindo abstrações para tipos comuns de dados, utilitários para *string*, utilitários para coleção, gerenciamento de recursos e preferências.
CoreGraphics.framework	CG	Contém as interfaces para Quartz 2D.
CoreLocation.framework	CL	Contém as interfaces para determinar a localização do usuário.
CoreMedia.framework	CM	Contém rotinas de baixo nível para manipulação de áudio e vídeo.
CoreMIDI.framework	MIDI	Contém rotinas de baixo nível para manipulação de dados MIDI.
CoreMotion.framework	CM	Contém interfaces para acessar dados do acelerômetro e do giroscópio.
CoreTelephony.framework	CT	Contém rotinas para acessar informações relacionadas a telefonia.

Tabela 3.2: Frameworks Parte 2

Nome	**Prefixos**	**Descrição**
CoreText.framework	CT	Contém um *layout* de texto e motor de renderização.
CoreVideo.framework	CV	Contém rotinas de baixo nível para manipulação de áudio e vídeo.
EventKit.framework	EK	Contém interfaces para acessar os dados de evento no calendário do usuário.
External-Accessory.framework	EA	Contém interfaces de comunicação com acessórios de *hardware*.
Foundation.framework	NS	Contém interfaces para gerenciamento de *strings*, coleções e outros tipos de dados de baixo nível.
GameKit.framework	GK	Contém as interfaces para o gerenciamento de conexão *peer-to-peer*.
iAd.framework	AD	Contém classes para a exibição de anúncios na aplicação.
ImageIO.framework	CG	Contém classes para ler e gravar dados de imagem.
IOKit.framework	N/A	Contém interfaces utilizadas pelo dispositivo.
MapKit.framework	MK	Contém classes para a incorporação de mapa na aplicação.
MediaPlayer.framework	MP	Contém interfaces para reproduzir vídeo em tela cheia.
MessageUI.framework	MF	Contém interfaces para compor e para pesquisar mensagens de e-mail.
OpenGLES.framework	EAGL GL	Contém as interfaces para o OpenGL ES, que é um versão embutida do OpenGL para multi-plataforma com biblioteca de renderização de Gráficos 2D e 3D.
QuartzCore.framework	CA	Contém as interfaces *Core Animation*.
QuickLook.framework	QL	Contém interfaces para a visualização de arquivos.
MobileCore-Services.framework	UT	Define os identificadores tipo uniforme (UTIs) suportado pelo sistema.

Tabela 3.3: Frameworks Parte 3

Nome	**Prefixos**	**Descrição**
Security.framework	CSSM,Sec	Contém interfaces para o gerenciamento de certificados, chaves públicas e privadas, e as políticas de confiabilidade.
StoreKit.framework	SK	Contém interfaces para lidar com transações financeiras associadas a loja virtual.
System-Configuration.framework	SC	Contém interfaces para determinar a configuração da rede de um dispositivo.
UIKit.framework	UI	Contém classes e métodos associadas a interface gráfica de usuário.

Capítulo 4

Linguagem Objetive-C

Devido a característica de orientação a objetos, a linguagem Objetive-C foi escolhida como linguagem padrão de desenvolvimento para a plataforma Apple iOS, o que ajuda no desenvolvimento no ambiente *Cocoa*. Segundo a Apple (2010e), a vantagem de se utilizar uma abordagem orientada a objetos para o desenvolvimento de aplicações, é que torna o programa mais fácil de ser projetado, desenvolvido, modificado e entendido, além de ser mais rápido de ser desenvolvido.

Como Objetive-C é uma extensão do padrão ANSI C, fica mais fácil de adaptar programas existentes em linguagem C utilizando Objetive-C. Outra vantagem é que a linguagem Objetive-C incorpora a linguagem C, assim tem-se todos os benefícios da linguagem C quando se trabalha dentro da linguagem Objetive-C (APPLE, 2010f).

4.1 Objetive-C e Linguagem C

Objetive-C é um superconjunto do padrão ANSI C e suporta a mesma sintaxe básica da linguagem C. Como na linguagem C, o programador define arquivos *header* e arquivos fonte para separar declarações públicas dos detalhes de implementação do código-fonte. O arquivo *header* de Objetive-C usa a extensão .h, nesse arquivo se especificam as classes, tipos, funções, e declarações de constantes. O arquivo fonte usa a extensão .m, o código Objetive-C é escrito nesse arquivo.

Quando se quer incluir arquivos *header* no código fonte, deve-se usar a diretiva *#import*. Isto é semelhante à diretiva *#include*, só que *#import* certifica que o mesmo arquivo nunca será incluso mais de uma vez.

4.2 Classes

A Figura 4.1 mostra a sintaxe para declaração de uma classe chamada MinhaClasse, que é uma subclasse da classe NSObject. A declaração da classe começa com a diretiva

de compilador @interface e termina com a diretiva @end. Em seguida vem o nome da classe, e separado dela por dois pontos, vem o nome da sua superclasse. As variáveis de instância da classe são declaradas no bloco de código que é delimitado por chaves. Seguido do bloco de variáveis de instância vem à lista de declaração dos métodos da classe. O ponto-e-vírgula marca o final de cada variável de instância e a declaração de cada método.

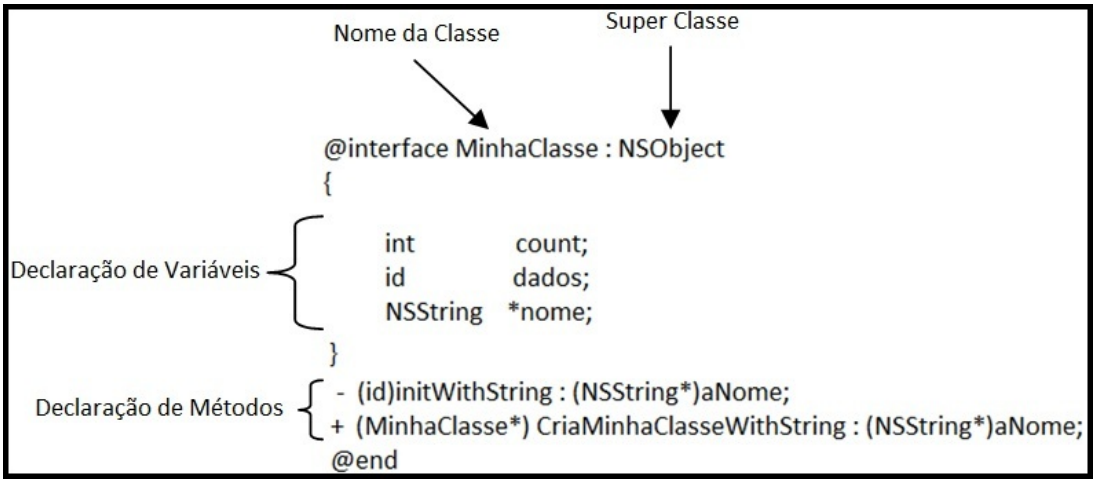

Figura 4.1: Sintaxe para declaração de uma classe

4.2.1 Objetos

Um objeto é uma instância de uma classe em tempo de execução que associa dados com operações particulares que podem usar ou afetar esses dados. Em Objetive-C, estas operações são conhecidas como métodos do objeto, os dados usados ou afetados pelos métodos são conhecidos como variável de instância.

Em Objetive-C, as variáveis de instância de um objeto são internas ao objeto; geralmente, o acesso ao estado de um objeto só é permitido através dos seus métodos. Para se descobrir algo sobre um objeto, tem que haver um método que fornece informações sobre o mesmo.

Além disso, um objeto só acessa os métodos que foram projetados para ele, não podendo executar métodos destinados a outros tipos de objetos. Assim como uma função em C protege suas variáveis locais, escondendo-as do resto do programa, um objeto em Objetive-C esconde suas variáveis de instância e implementações de seus métodos.

4.2.2 Id

Objetive-C suporta os tipos *strong* e *weak* para variáveis que contém objetos. As variáveis do tipo *strong* incluem o nome da classe na declaração das variáveis. As variáveis do tipo *weak* usam o tipo *id* na declaração. Variáveis do tipo *weak* são frequentemente

usadas em classes de coleção, onde o tipo do objeto na coleção não é conhecido. O tipo *weak* fornece uma grande flexibilidade e permite muito mais dinamismo nos programas escritos em Objetive-C.

O Código 4.1 a seguir mostra a declaração dos tipos *strong* e *weak*:

Código 4.1: Tipos *strong* e *weak*.

```
1 minhaClasse *meuObjeto1; // Tipo strong
2 id meuObjeto2;           // Tipo weak
```

Nota-se que na primeira declaração há um asterisco (*). Em Objetive-C, objetos de referência são ponteiros. Nas variáveis do tipo *strong* o asterisco vem antes do nome da variável do objeto. No tipo *id* não precisa utilizar o asterisco. Para as construções orientadas a objeto em Objetive-C, tais como valores de retorno de método, *id* substitui *int* como o tipo de dados padrão. Para construções estritamente em C, tais como valores de retorno de função, *int* continua sendo o tipo padrão. As duas barras no Código 4.1 indica comentário.

A palavra-chave *nil* é definida como um objeto nulo, um *id* com um valor de 0. *id*, *nil*, e os outros tipos básicos de Objetive-C são definidos no arquivo header objc/objc.h.

O Código 4.2 a seguir mostra a definição de *id* como ponteiro para uma estrutura de dados:

Código 4.2: Estrutura de dados id.

```
1 typedef struct objc_object{
2 Class isa;
3 }*id;
```

Cada objeto possui uma variável *isa* que lhe diz de qual classe é uma instância.

4.2.3 Tipagem Dinâmica

O tipo *id* é completamente não restritivo. Por si só, não produz informações sobre um objeto, exceto que ele é um objeto. Em algum ponto, um programa normalmente precisa encontrar informações mais específicas sobre os objetos que ele contém. Como o designador de tipo *id* não pode fornecer essas informações específicas para o compilador, cada objeto tem de ser capaz de fornecê-lo em tempo de execução.

A variável de instância *isa* identifica a qual classe o objeto pertence. Objetos com o mesmo comportamento (métodos) e os mesmos tipos de dados (variáveis de instância) são membros da mesma classe.

Os objetos são dinamicamente tipados em tempo de execução. Sempre que ele precisa, o sistema de tempo de execução pode encontrar a classe exata que um objeto pertence,

apenas perguntando ao objeto. Tipagem dinâmica em Objetive-C serve como base para a ligação dinâmica.

A variável *isa* também permite que objetos executem a introspecção para descobrir sobre si mesmos (ou outros objetos). O compilador registra informações sobre as definições de classe em estruturas de dados para o sistema de tempo de execução usar. As funções do sistema de tempo de execução utilizam *isa* para encontrar estas informações em tempo de execução. Usando o sistema de tempo de execução, pode-se, por exemplo, determinar se um objeto implementa um método particular ou descobrir o nome da sua superclasse.

4.3 Métodos, Mensagem e Propriedades

Esta seção introduz os conceitos de métodos, mensagem e propriedades em Objetive-C, analisando a forma sintática destes.

4.3.1 Métodos

Pode-se declarar dois tipos de métodos em Objective-C: métodos de instância e métodos de classe. Antes de invocar um método de instância, primeiro deve-se criar uma instância de uma classe. Já o método de classe não precisa de uma instância de classe para poder ser invocado.

A declaração de um método consiste em um identificador de tipo de método, um tipo de retorno, uma ou mais palavras de assinatura, um tipo de parâmetro e informação de nome. A Figura 4.2 mostra a declaração do método de instância insertObject:atIndex:.

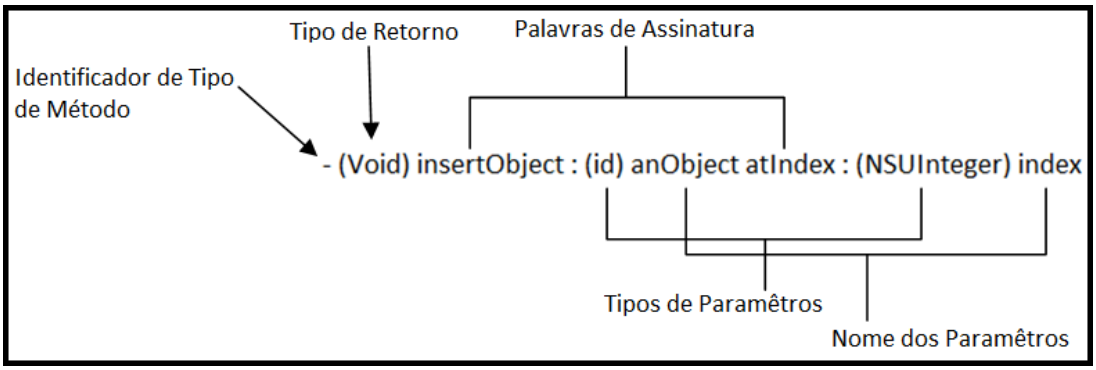

Figura 4.2: Sintaxe para declaração de um método

Esta declaração é precedida por um sinal de menos (-), que indica que é um método de instância. O nome do método é uma concatenação de todas as palavras de assinatura. O caractere dois pontos (:) declara a presença de um parâmetro. Se o método não tem parâmetro, o caractere dois pontos pode ser omitido depois da primeira palavra de assinatura. No exemplo da Figura 4.2, o método possui dois parâmetros.

4.3.2 Sintaxe da Mensagem

Para que um objeto faça algo, é necessário enviar uma mensagem invocando algum dos seus métodos. Em Objetive-C, a expressão da mensagem é colocada entre colchetes, como mostrado no Código 4.3.

Código 4.3: Expressão de uma mensagem.

```
1 [receiver message];
```

O *receiver* é um objeto, e *message* diz à ele o que fazer. No código-fonte, *message* é simplesmente o nome de um método e os parâmetros que são passados para ele. Quando uma mensagem é enviada, o sistema de tempo de execução seleciona o método apropriado no repertório do *receiver*, em seguida o método é invocado.

Por exemplo, a mensagem no Código 4.4 solicita que o objeto aluno execute o seu método obterNome, que faz com que o nome do aluno seja retornado pelo método:

Código 4.4: Método obterNome.

```
1 [aluno obterNome];
```

A mensagem é seguida por um (;) como é normal para qualquer instrução em C.

Métodos também podem possuir parâmetros. Para uma mensagem com um único parâmetro coloca-se dois pontos (:) depois do nome do método e em seguida o parâmetro, como mostrado no Código 4.5.

Código 4.5: Parâmetro de um método.

```
1 [aluno setMatricula:080111];
```

O método setMatricula recebe o número de matrícula do aluno como parâmetro, quando o método é invocado a matrícula do aluno é alterada para o valor 080111.

Para evitar o armazenamento dos resultados temporários em várias variáveis locais, Objetive-C permite o aninhamento de mensagens. O valor retornado por uma mensagem aninhada é usado como parâmetro para outra mensagem. No Código 4.6 uma variável é economizada usando-se mensagens aninhadas.

Código 4.6: Aninhamento de mensagens.

```
1 [aluno2 setNome: [aluno1 obterNome]];
```

Neste caso, o método obterNome retorna o nome do aluno1 que é passado como parâmetro para o método setNome que atualiza o nome do aluno2.

Os Códigos 4.3, 4.4, 4.5 e 4.6 enviam mensagens para uma instância de uma classe. Outra maneira de envio de mensagens é quando estas são invocadas pela própria classe, sem a necessidade de se criar uma instância. Para enviar mensagens para a própria classe,

o método deve ser especificado como sendo um método de classe ao invés de um método de instância.

Normalmente os métodos de classe são usados como construtores, para criar novas instâncias de uma classe ou para acessar alguma informação compartilhada associada com a classe. A sintaxe para a declaração de um método de classe é idêntico ao método de instância, a única diferença é que ao invés de usar o sinal de menos (-) para o identificador de tipo de método, usa-se o sinal de mais (+).

No Código 4.7 é ilustrado um método de classe. Nesse caso, o método array é um método de classe da classe NSMutableArray que aloca e inicializa uma nova instância desta classe.

Código 4.7: Método de classe.

```
1 NSMutableArray *myArray = nil;
2 myArray = [NSMutableArray array];
```

4.3.3 Declaração de propriedades

A declaração de propriedades é uma notação conveniente para substituir a implementação de métodos de acesso.

A declaração de propriedades é inclusa com a declaração de métodos na classe interface. A declaração usa a diretiva de compilador @property, seguido pelo tipo de informação e o nome da propriedade. Pode-se configurar a propriedade com opções que definem como o método de acesso vai se comportar. O Código 4.8 mostra algumas declarações de propriedades:

Código 4.8: Declarações de propriedades.

```
1 @property BOOL flag;
2 @property (copy) NSString *nomeObjeto; // Copia o objeto durante a
      atribuição
3 @property (readonly) UIView *rootView;  // Declara somente o método
      getter.
```

Na classe de implementação, usa-se a diretiva de compilador @synthesize que faz com que o compilador gere os métodos de acordo com a especificação na declaração de propriedades. No Código 4.9 é ilustrado o uso de @synthesize:

Código 4.9: Diretiva @synthesize.

```
1 @synthesize flag;
2 @synthesize nomeObjeto;
3 @synthesize rootView;
```

O uso de propriedades reduz a quantidade de códigos redundantes que se escreve, pois a maioria dos métodos de acesso são implementados da mesma maneira. Propriedades eliminam a necessidade de implementar os métodos *getter* e *setter* para cada propriedade exposta na classe. Ao invés disso, o comportamento que se deseja é especificado usando a declaração de propriedades e então os métodos *getter* e *setter* são sintetizados no tempo de compilação.

4.4 Gerenciamento de Memória, Strings e Protocolos

Esta seção introduz os conceitos de gerenciamento de memória, *strings* e protocolos em Objetive-C.

4.4.1 Gerenciamento de Memória

Em qualquer programa, é importante garantir que os objetos serão desalocados da memória quando não forem mais necessários. Também é importante garantir que os objetos não serão desalocados da memória enquanto eles ainda estão em uso.

Segundo Apple (2010b) o iOS não suporta o *Garbage collection*, assim o gerenciamento de memória deve ser feito através do mecanismo de contagem de referência que permite determinar a vida útil dos objetos.

O mecanismo de contagem de referência usa um contador para cada objeto, quando este é criado o contador é incrementado. Quando um objeto é liberado, o contador é decrementado. Quando o valor do contador chegar em 0, o objeto é desalocado da memória pelo sistema operacional.

O contador de referência também é incrementado quando um objeto invoca o método *retain*. Para decrementar o contador de referência, o objeto pode usar o método *release*. O Código 4.10 ilustra o gerenciamento de memória usando *retain* e *release*.

Código 4.10: Gerenciamento de Memória.

```
1 NSString *str = [[NSString alloc] initWithString:@"Olá"];
2 NSString *str2 = str;
3 [str2 retain];
4 [str release];
5 [str2 release];
```

Na linha 1 o objeto *str* é criado e inicializado, o seu contador de referência possui o valor 1. Na linha 2 é criado um ponteiro chamado *str*2 para o objeto *str*. Se o objeto *str* invocar o método *release*, o objeto *str*2 ficará inválido por ser apenas um ponteiro (*str* e *str*2 compartilham o mesmo objeto na memória), então é necessário que o objeto

*str*2 invoque *retain* para que ele obtenha o objeto *str*. Na linha 3 *str*2 invoca *retain*, incrementando o contador de referência, agora o contador possui o valor 2. Para que o objeto seja desalocado da memória é necessário decrementar o valor do contador de referência duas vezes através do método *release*. Nas linhas 4 e 5 isso é feito.

4.4.2 Strings

Objetive-C lida com *string* através do objeto NSString que fornece várias funcionalidades de manipulação. Uma *string* em Objetive-C é delimitada por aspas duplas e precedida pelo caractere @, como mostrado no Código 4.11.

Código 4.11: String em Objetive-C.

```
1 NSString *minhaString = @"Minha String";
```

4.4.3 Protocolos

Um protocolo declara métodos que podem ser implementados por alguma classe. Protocolos não são classes, eles simplesmente definem uma interface que outras classes devem implementar.

Uma classe implementa um protocolo colocando no bloco da interface o nome do protocolo entre "< >" depois do nome da superclasse. O Código 4.12 ilustra o uso de um protocolo:

Código 4.12: Protocolo em Objetive-C.

```
1 @interface MinhaClasse : NSObject < UIApplicationDelegate,
    AnotherProtocol >{
2
3 }
4 @end
```

A declaração de um protocolo é semelhante à declaração de uma classe interface, a única exceção é que os protocolos não têm classe pai e não definem variáveis de instância. O Código 4.13 mostra a declaração de um simples protocolo com um único método:

Código 4.13: Declaração de um protocolo.

```
1 @protocol MeuProtocolo
2   - (void) meuMetodoProtocolo;
3 @end
```

Capítulo 5

iOS SDK

Antes de desenvolver aplicativos para o iOS é necessário um computador Intel Macintosh com a versão mais atualizada do sistema operacional da Apple Inc., pois para instalar o iOS SDK são exigidas essas configurações (MARK; LAMARCHE, 2009).

A plataforma iOS foi baseada no conhecimento utilizado na criação do Mac OS X, e muitas ferramentas e tecnologias que se utiliza no desenvolvimento para essa plataforma também surgiu desse conhecimento. Apesar de sua semelhança com o Mac OS X, o iOS não exige experiência no desenvolvimento de aplicativos para o Mac OS X. O iOS SDK fornece tudo que se precisa para começar a criar aplicações para o iOS (APPLE, 2010d).

5.1 iOS SDK

O iOS SDK contém as ferramentas necessárias para projetar, criar, depurar e aperfeiçoar o software para o iOS. Ele também contém arquivos *header*, amostras de código e documentação para as tecnologias dessa plataforma. O iOS SDK pode ser baixado no iOS *Dev Center*, que está localizada em:

http://developer.apple.com/devcenter/ios

Os principais componentes inclusos no iOS SDK são:

- **Xcode Tools** - é um pacote de ferramentas que suportam o desenvolvimento de aplicações para o iOS, as principais aplicações desse pacote são:
  - **Xcode** - é um ambiente de desenvolvimento integrado que gerencia projetos de aplicação e permite editar, compilar, executar e depurar o código-fonte. O *Xcode* integra-se com muitas outras ferramentas e é a principal aplicação usada durante o desenvolvimento dos aplicativos para o iOS.

  - **Interface Builder** - é uma ferramenta usada para construir interface de usuário visualmente. Os objetos de interface que se cria são salvos em um tipo especial de arquivo de recurso e são carregados na aplicação em tempo de execução.
  - **Instruments** - é uma ferramenta de análise de desempenho de execução e de depuração. Pode-se usar o *Instruments* para juntar informações sobre o comportamento do aplicativo em tempo de execução e identificar problemas potenciais.

- **iOS Simulator** - é um aplicativo do Mac OS X que simula a tecnologia iOS, o que permite testar aplicativos iOS localmente em um computador Intel Macintosh.

- **iOS Developer Library** - Contêm a referência e a documentação conceitual que explica toda tecnologia necessária para o desenvolvimento de aplicativos iOS.

Embora seja possível executar aplicações no *iOS Simulator*, o *Xcode* e o *Instruments* também permitem que as aplicações sejam executadas e depuradas diretamente em um dispositivo conectado ao computador. O *iOS Simulator* é ideal para construir e testar aplicações rapidamente, mas não substitui um dispositivo real para os testes dos aplicativos. Se for necessário testar um aplicativo que utiliza o acelerômetro ou a câmera do dispositivo, o teste deve ser feito no próprio dispositivo, pois o *iOS Simulator* não suporta esses recursos de *hardware* (MARK; LAMARCHE, 2009). Para realizar testes em um dispositivo real é necessário uma inscrição paga na *Apple Developer Program*, também é necessário configurar o dispositivo para fins de desenvolvimento (APPLE, 2010d).

Nas seções a seguir serão detalhadas as principais ferramentas usadas no desenvolvimento de aplicativos para o iOS.

5.2 Xcode

As ferramentas do *Xcode Tools* oferecem suporte para gerenciamento de projetos, edição de código, construção de executáveis, depuração, ajuste de desempenho, etc. A principal ferramenta desse pacote é o *Xcode* em si, que fornece o ambiente de desenvolvimento básico do código-fonte. Essa ferramenta incorpora uma série de recursos, o que tornar mais fácil o desenvolvimento de aplicativos para o iOS, incluindo o seguinte:

- Um sistema de gerenciamento de projetos para a definição de produtos de *software*.

- Um ambiente de edição de código que inclui características tais como coloração de sintaxe, indexação de símbolo e função que completa o código.

- Um visualizador de documentação avançado para visualização e busca da documentação da Apple Inc..
- Um *inspector* sensível ao contexto para visualizar informações sobre o código de símbolos selecionados.
- Um sistema de construção avançado com regras de verificação de construção.
- Compiladores GCC que suportam C, C++, Objetive-C, Objetive-C++ e outras linguagens.
- LLVM e Clang que suportam C, C ++ e Objetive-C.
- Depuração integrada usando GDB.
- Computação distribuída, permitindo distribuir grandes projetos em várias máquinas em rede.
- Previsão de compilação, que acelera a compilação dos arquivos.
- Recursos de depuração avançada, como corrigir e continuar e formatadores de dados personalizados.
- Ferramentas de refatoração avançada que permitem fazer modificações globais no código-fonte sem alterar seu comportamento.
- Suporte a execução de ferramentas de desempenho para analisar o *software*.
- Suporte a gerenciador de código-fonte integrado.
- Suporte a *AppleScript* para automatizar o processo de construção.

Para criar um novo aplicativo iOS, primeiro cria-se um novo projeto no *Xcode*. Um projeto gerencia todas as informações associadas com o aplicativo, incluindo os arquivos de recursos, configurações de construção e regras necessárias para juntar todos os arquivos. A Figura 5.1 mostra a janela de um projeto no *Xcode*. Esta janela fornece acesso rápido a todos os elementos-chave da aplicação. Nos grupos e lista de arquivos, pode-se gerenciar os arquivos do projeto, incluindo os arquivos de recursos e alvos de construção. Na barra de ferramentas, pode-se acessar ferramentas e comandos comumente usados. No painel de detalhes, pode-se visualizar o conteúdo dos arquivos do projeto.

Quando se constrói uma aplicação no *Xcode* tem-se a opção de construí-la para *iOS Simulator* ou para um dispositivo. O *iOS Simulator* fornece um ambiente local para testar os aplicativos e certificar-se que eles se comportam de maneira desejada. Após se sentir satisfeito com o comportamento básico do aplicativo, pode-se construir através do *Xcode* o

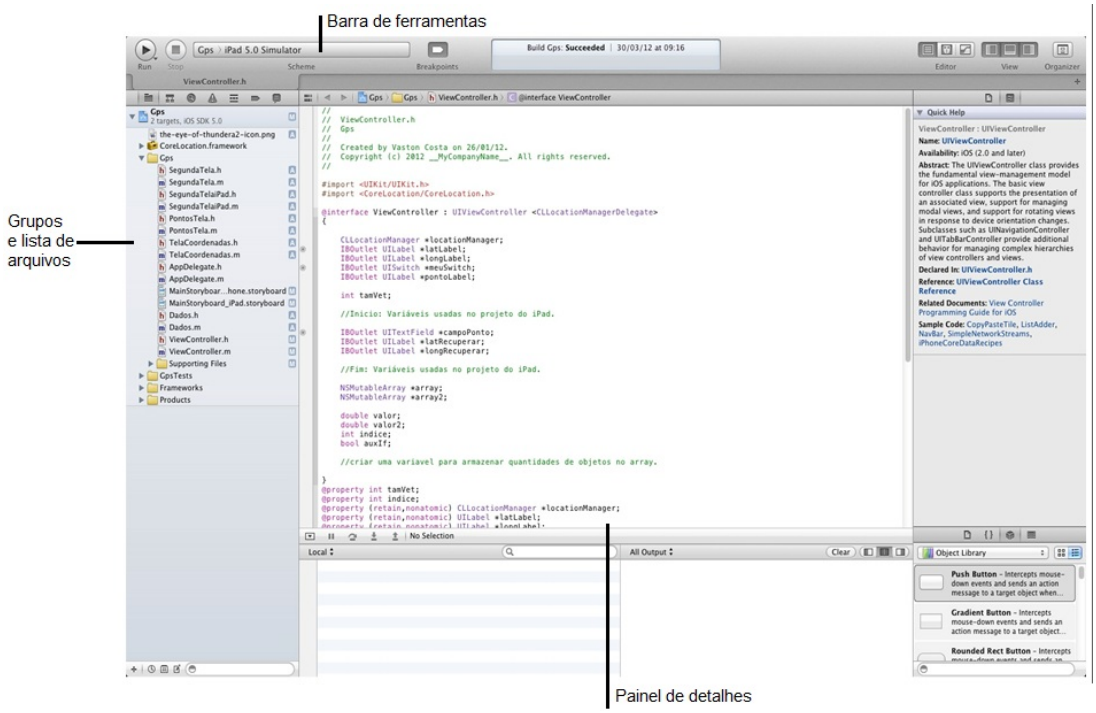

Figura 5.1: Janela de um projeto no *Xcode*

aplicativo e executá-lo em um dispositivo móvel da Apple Inc. conectado ao computador. Os testes finais podem ser feitos executando o aplicativo no próprio dispositivo. A Figura 5.2 ilustra a execução de um projeto no *Xcode*.

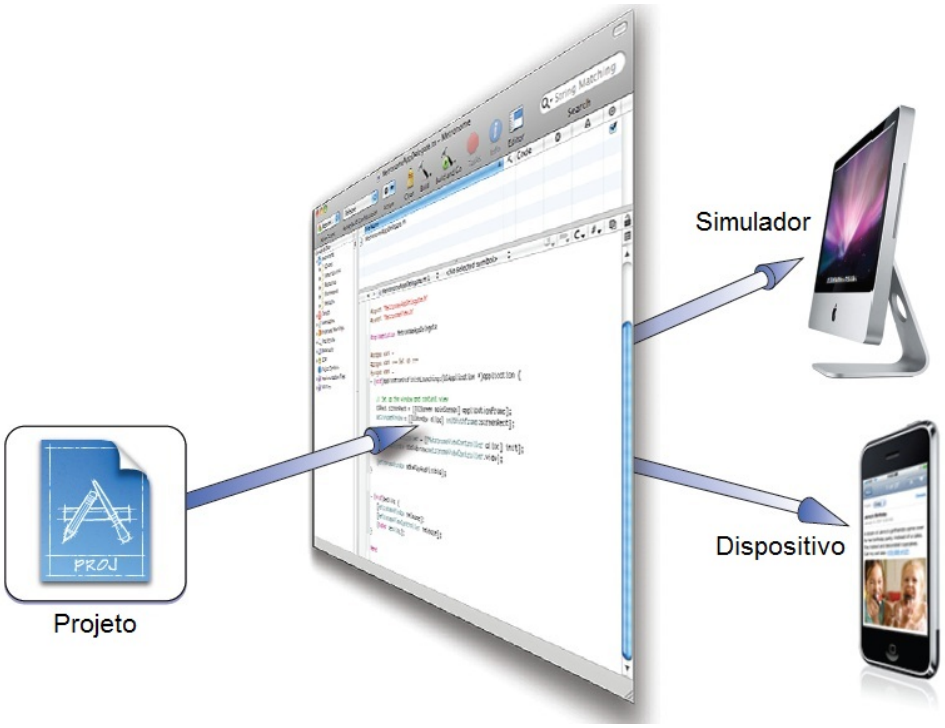

Figura 5.2: Executando um projeto no *Xcode* (APPLE, 2010d)

5.3 Interface Builder

O *Interface Builder* é uma ferramenta usada para construir interface do usuário visualmente. Com essa ferramenta, pode-se construir a janela do aplicativo, arrastando os componentes gráficos pré-configurados, como mostrado na Figura 5.3. Entre os componentes encontram-se controles do sistema padrão, como *switches*, campos de texto e botões, além de exibições personalizadas.

Depois que os componentes estão na superfície da janela, é possível posicioná-los, arrastando-os pela janela, seus atributos podem ser configurados usando o *inspetor*, estabelecendo as relações entre os objetos da interface gráfica e o código-fonte. Quando a interface está pronta, seu conteúdo pode ser salvo em um arquivo *nib*, que é um formato de arquivo personalizado de recursos.

Figura 5.3: Construindo uma interface gráfica usando o *Interface Builder* (APPLE, 2010d)

Os arquivos *nib* criados no *Interface Builder* contêm todas as informações que o UIKit precisa para recriar os mesmos objetos na aplicação em tempo de execução. Quando um arquivo *nib* é carregado, versões de todos os objetos armazenados nesse arquivo são criadas em tempo de execução, elas são configuradas exatamente como eram no *Interface Builder*. Segundo Piper (2009), a aparência da interface de usuário no *Interface Builder* é uma aproximação muito boa da que será apresentada no programa em execução.

No geral, usando o *Interface Builder* ganha-se muito tempo quando se trata de criar interface de usuário, pois este elimina o código necessário para criar, configurar e posicionar os objetos que compõem a interface.

5.4 Instruments

O ambiente *Instruments* permite analisar o desempenho dos aplicativos iOS durante a execução no simulador ou em um dispositivo. *Instruments* junta os dados do aplicativo em execução e apresenta esses dados em uma exibição gráfica chamada linha do tempo. Pode-se coletar dados sobre o uso de memória, atividade do disco, a atividade da rede e desempenho de gráficos do aplicativo. A linha de tempo pode mostrar todos os tipos de informações lado a lado, permitindo assim, relacionar o comportamento global da aplicação, não apenas o comportamento específico. A Figura 5.4 mostra o ambiente do *Instruments*.

Figura 5.4: Análise da aplicação usando *Instruments* (APPLE, 2010d)

Além de fornecer a linha do tempo, *Instruments* fornece ferramentas que pode ser utilizada para analisar o comportamento da aplicação ao longo do tempo. Por exemplo, a janela *Instruments* permite armazenar dados de várias execuções de modo que seja possível ver se o comportamento do aplicativo está realmente melhorando ou se ainda precisa ser modificado.

Capítulo 6

Aplicativo Proposto

Este capítulo descreve o protótipo do aplicativo proposto nesse trabalho, abordando o modelo de projeto *Model View Controller*, o algoritmo que calcula a área do terreno, o código-fonte das principais funcionalidades do protótipo e os diagramas de classes. No final desse capítulo também serão apresentados os resultados e testes.

Optou-se por desenvolver o aplicativo na plataforma iOS pelo alto nível de precisão do GPS dos dispositivos que executam esse sistema operacional, além de possuírem uma boa autonomia da bateria. Outro motivo importante é a rica documentação disponibilizada pela própria Apple Inc. que explica toda tecnologia necessária para se desenvolver aplicativos para o iOS. Esta documentação é livre e pode ser encontrada no site da Apple Inc.[1].

Os *frameworks* desta plataforma oferecem recursos de manipulação do GPS e de gráficos simples e intuitivos. Estes foram utilizados na implementação do protótipo do aplicativo proposto. Os *frameworks* garantem a portabilidade do aplicativo entre os dispositivos móveis da Apple Inc., pode-se escrever um aplicativo para o iPhone utilizando esses *frameworks*, e depois executar o mesmo no iPad sem problemas.

6.1 Objetivos do aplicativo

O protótipo do aplicativo proposto nesse trabalho visa suprir uma carência de pequenos e médios agricultores, no que envolve medidas de áreas em campo aberto e quantidade de sementes de milho necessárias para o plantio.

Tal aplicativo utiliza o recurso GPS do dispositivo móvel para obter coordenadas geográficas de pontos limítrofes do terreno. Com esses pontos o algoritmo forma triângulos, calculando a área de cada um destes separadamente. A área do terreno é obtida somando a área de todos os triângulos.

[1] `http://developer.apple.com/library/ios/navigation/` Acessado em 27 de janeiro de 2012

De posse da medida da área, o aplicativo gera a quantidade de sementes necessárias para o plantio, contribuindo para redução dos gastos do produtor.

6.2 Model View Controller

Esta seção tem como objetivo apresentar o modelo de projeto utilizado no desenvolvimento de aplicativos para o iOS, o *Model View Controller*. A comunicação entre a tela e o código-fonte do aplicativo se dá através dos objetos *Model*, *View* e *Controller*. Cada um dos três tipos de objetos é separado um dos outros por meio de fronteiras abstratas e se comunicam através dessas fronteiras , como mostrado na Figura 6.1(APPLE, 2010c).

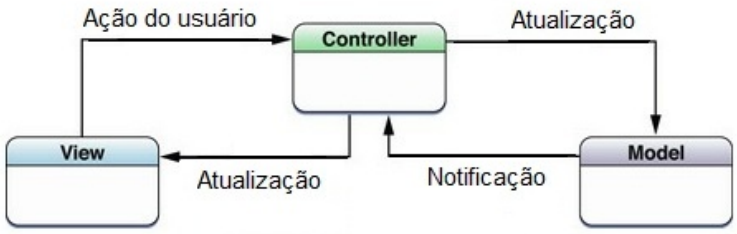

Figura 6.1: Modelo de projeto *Model View Controller* (APPLE, 2010a)

6.2.1 Model

Objetos *model* encapsulam os dados específicos em um aplicativo e definem a lógica e a computação que manipulam e processam esses dados. Por exemplo, um objeto *model* pode representar um personagem em um jogo ou um contato em uma agenda. Grande parte dos dados que fazem parte do estado persistente da aplicação (se esse estado persistente é armazenado em arquivos ou bancos de dados) deve residir nos objetos *model* após os dados serem carregados na aplicação.

Como os objetos *model* reproduzem conhecimentos e experiências relacionadas a um domínio de um problema específico, eles podem ser reutilizados em problemas de domínios semelhantes. Um objeto *model* deve ter nenhuma conexão explícita com objetos *view* que apresentam seus dados e permitem aos usuários editá-los. Os objetos *model* não têm ligação com a interface de usuário e temas de apresentação.

As ações de usuário na camada de visualização do aplicativo que criam ou modificam dados são comunicadas aos objetos *model* por meio de um objeto *controller*, essas ações resultam na criação ou atualização de dados nos objetos *model*. Quando um objeto *model*

é alterado (por exemplo, novos dados recebidos através de uma conexão de rede), ele notifica a um objeto *controller*, que atualiza os objetos *view* apropriados.

6.2.2 View

Um objeto *view* é um objeto em um aplicativo que os usuários podem ver. Um objeto *view* desenha a si próprio e pode responder às ações do usuário. Um dos objetivos principais dos objetos *view* é exibir dados a partir de objetos *model* da aplicação e permitir a edição dos dados. Apesar disso, objetos *view* são normalmente dissociados de objetos *model* em um aplicativo.

Os objetos *view* asseguram a coerência entre as aplicações. Ambos os *frameworks* UIKit e AppKit fornecem coleções de classes de exibição que podem ser utilizados nas interfaces de usuário. O *Interface Builder* também oferece dezenas de objetos de exibição em sua biblioteca.

Objetos *view* aprendem sobre mudanças no objeto *model* através de objetos *controller*. As mudanças iniciadas pelo usuário nos objetos *view* são comunicadas aos objetos *model* pelos objetos *controller*, ou seja, os objetos *controller* estão entre os objetos *view* e *model*.

6.2.3 Controller

Um objeto *controller* atua como um intermediário entre um ou mais objetos *view* e um ou mais objetos *model*. Objetos *controller* são, portanto, um canal através do qual os objetos *view* recebe informações sobre as mudanças nos objetos *model* e vice-versa. Objetos *controller* também podem executar tarefas de coordenação para uma aplicação e gerenciar os ciclos de vida de outros objetos.

Um objeto *controller* interpreta as ações feitas pelos usuários em objetos *view* e comunica as alterações ou criação de novos dados a camada *model*. Quando um objeto *model* é alterado ou criado, um objeto *controller* comunica essas ações aos objetos *view* para que estes possam exibi-lo na tela do aplicativo.

6.2.4 Ligação dos Objetos da Interface Gráfica de Usuário ao Código-Fonte

Para que os objetos da tela interajam com a parte lógica do aplicativo, eles precisam de pontos de conexões definidos. Essas conexões podem ser declaradas por meio de diretivas: IBOutlets e IBActions. Um IBOutlets define o caminho entre o código e um objeto na tela que é usado para ler ou escrever valores. Um IBAction define um método na aplicação que pode ser disparado via um evento, como um toque em um botão na tela do aplicativo.

Os passos necessários para ligar os componentes da interface gráfica de usuário ao código-fonte são mostrados a seguir:

1. Criar uma classe no projeto do Xcode, a qual será vinculada a uma tela do aplicativo onde estão os componentes gráficos. Para criar a classe, clique no primeiro ícone no lado esquerdo do Xcode como mostra a seta na Figura 6.2, e com o botão direito do mouse clique na pasta com o nome do projeto, no caso desse exemplo a pasta chama "GPS" e está circulada na Figura 6.2. Selecione a opção "new file", escolha as opções *Cocoa touch* e UIViewController subclass como mostrado na Figura 6.3. Escolha um nome para a classe.

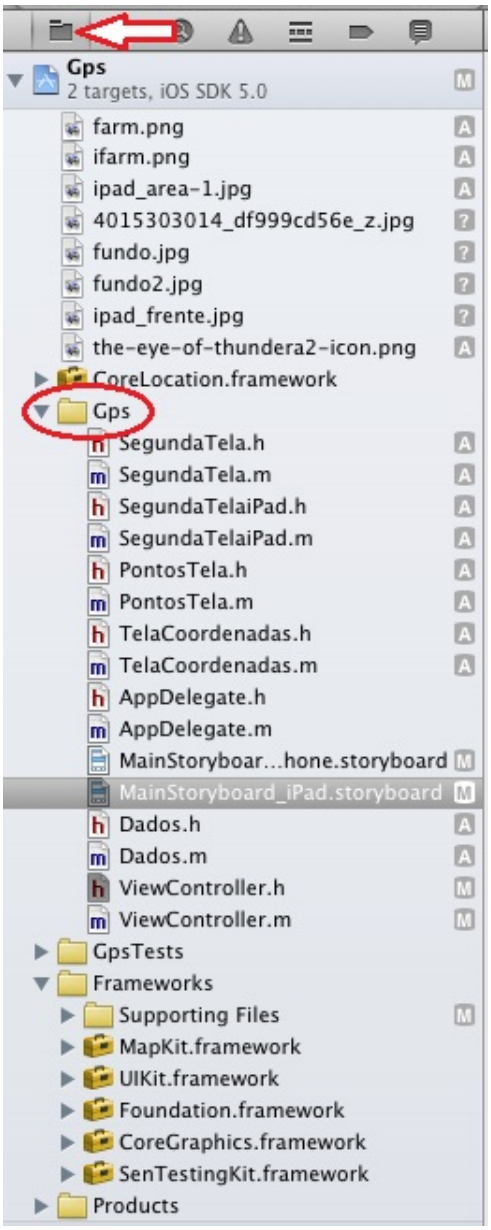

Figura 6.2: Criando um novo arquivo no Xcode

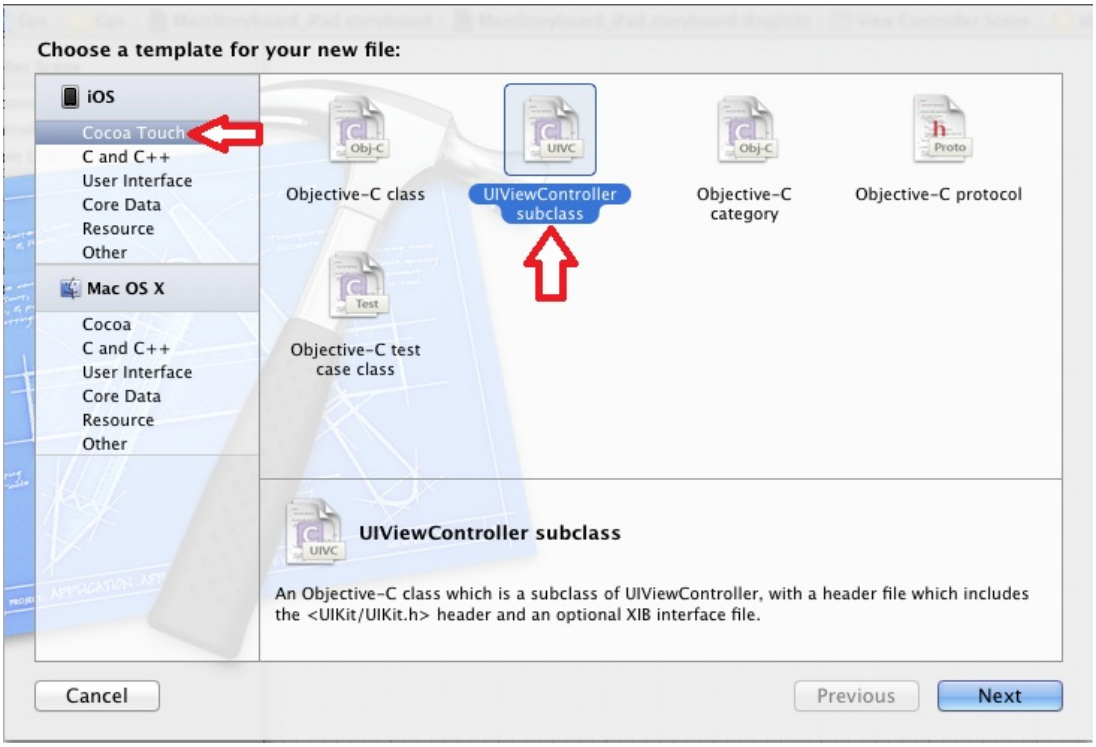

Figura 6.3: Criando uma classe ViewController

2. Declarar pontos de conexões IBOutlets e IBAction no arquivo de interface da classe criada no passo 1, este arquivo está circulado na Figura 6.4.

Figura 6.4: Arquivo de inteface da classe

3. A Figura 6.5 mostra alguns exemplos de declarações de componentes gráficos usando IBOutlets.

```
IBOutlet UILabel *latLabel;
IBOutlet UILabel *longLabel;
IBOutlet UISwitch *meuSwitch;
IBOutlet UILabel *pontoLabel;

IBOutlet UITextField *campoLatitude;
IBOutlet UITextField *campoLongitude;
IBOutlet UILabel *latRecuperar;
IBOutlet UILabel *longRecuperar;
IBOutlet UILabel  *resultadoArea;
```

Figura 6.5: Declarações de IBOutlets

4. A Figura 6.6 mostra alguns exemplos de declarações de métodos usando IBAction.

```
-(IBAction) salvarPontoGps;
-(IBAction) calculaAreaSementes;
-(IBAction) salvaPontoManual;
```

Figura 6.6: Declarações de IBActions

5. No lado esquerdo do *Interface Builder* onde a interface gráfica de usuário foi criada, clique na opção *View Controller*, como mostrado na Figura 6.7.

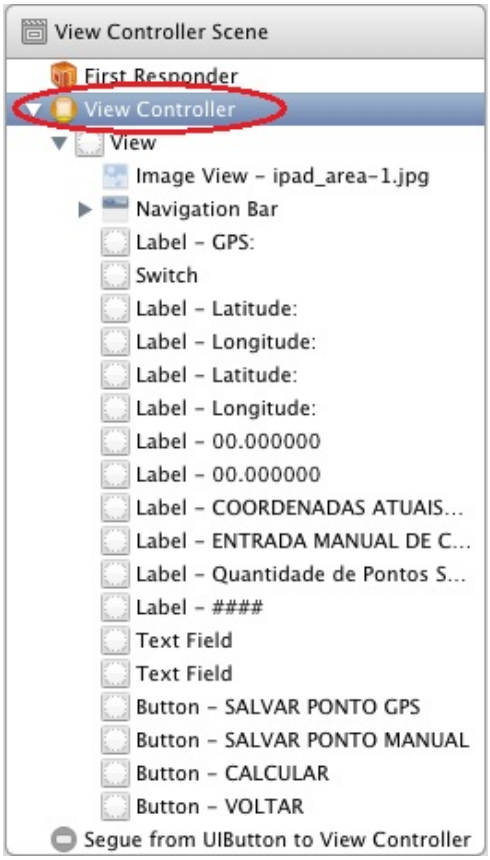

Figura 6.7: Opção View Controller

6. No lado direito da tela do *Interface Builder*, selecione o terceiro ícone da esquerda para direita, como mostra a seta na Figura 6.8. Em *Custom Class* selecione a classe criada no passo 1 que será vinculada a interface gráfica criada no *Interface Builder*.

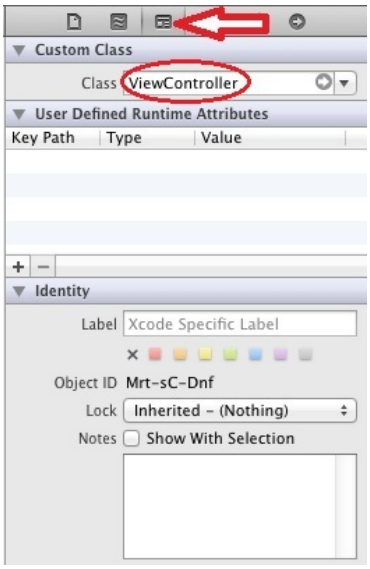

Figura 6.8: Opção *Custom Class*

7. No lado direito da tela do *Interface Builder*, selecione o sexto ícone da esquerda para direita, como mostra a seta na Figura 6.9. Em *Outlets*, aparecerá todos os componentes gráficos declarados na classe criada no passo 1, estes podem ser ligados aos componentes da interface gráfica de usuário. Em *Receiver Actions* aparecerá todos os métodos com o tipo de retorno IBAction declarados na classe criada no passo 1, estes podem ser ligados a qualquer evento provinda da interface gráfica.

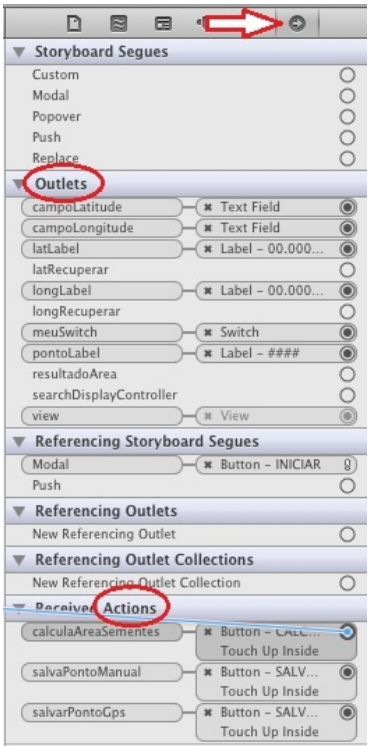

Figura 6.9: Opção Outlets e Actions

8. Na Figura 6.10, é mostrado a ligação de um componente gráfico chamado "latLabel" a interface gráfica de usuário, esse componente é responsável por mostrar um texto ao usuário. No caso do exemplo da Figura 6.10, esse componente mostra o valor atual da coordenada da latitude do GPS.

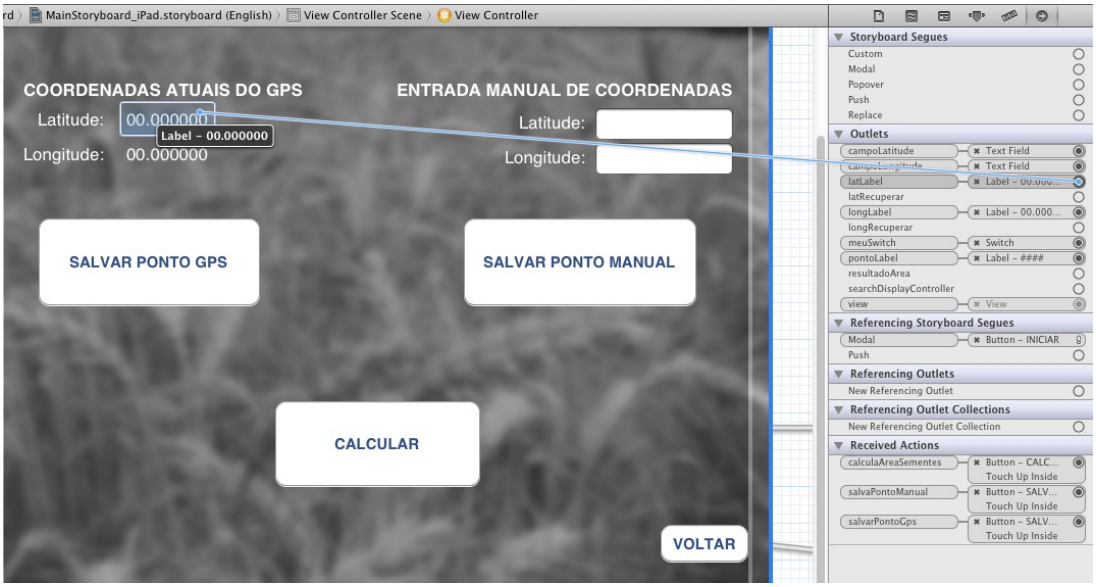

Figura 6.10: Ligação de um *label*

9. Na Figura 6.11, é mostrado a ligação de um componente gráfico chamado "campo-Latitude" a interface gráfica de usuário, esse componente é responsável por obter a

coordenada da latitude manualmente.

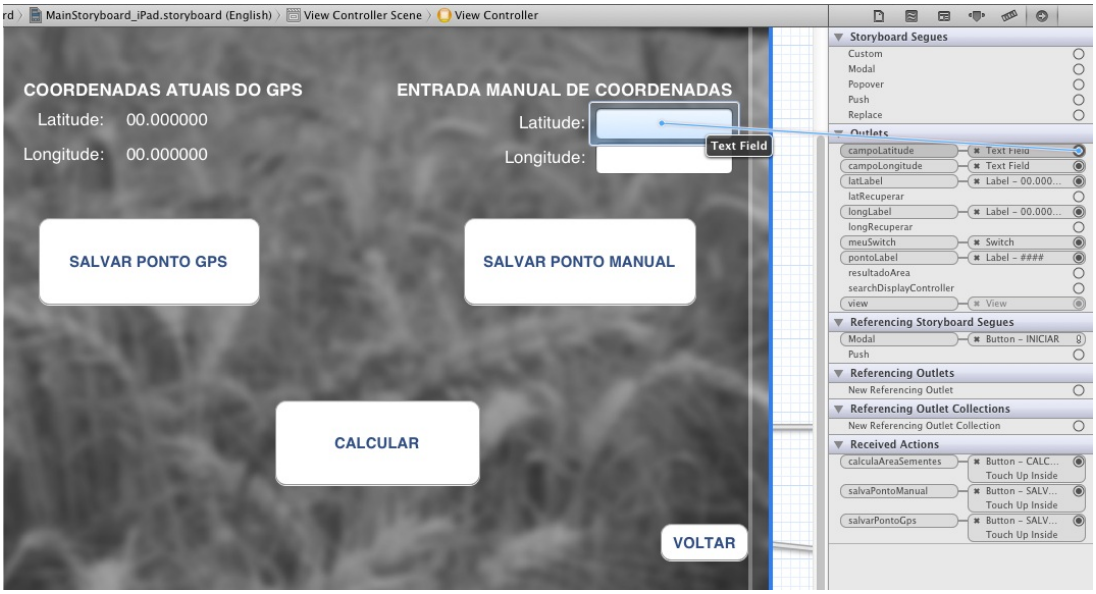

Figura 6.11: Ligação de um *Text Field*

10. Na Figura 6.12, é mostrado a ligação de um método chamado calculaAreaSementes a um botão na interface gráfica chamado "Calcular".

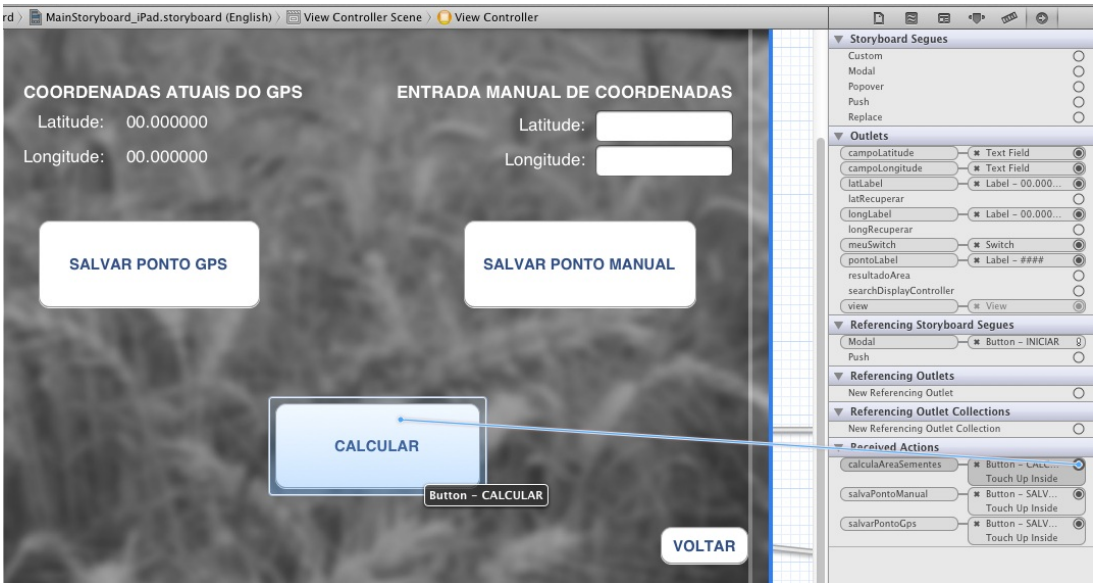

Figura 6.12: Ligação de um método IBAction a um botão na interface gráfica

6.3 Algoritmo que Calcula a Área do Terreno

Nesta seção será ilustrado como o algoritmo desenvolvido nesse trabalho calcula a área do terreno.

Suponha que o usuário coletou os pontos do terreno através do GPS do dispositivo, conforme a Figura 6.13.

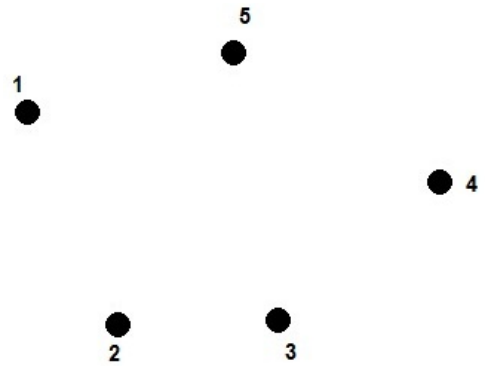

Figura 6.13: Coleta de pontos através do GPS

Os pontos são numerados conforme a ordem que o usuário os coleta, sendo o ponto 1 o primeiro e o ponto 5 o último. O algoritmo sempre pega o primeiro ponto como referência para formar triângulos com os outros pontos, as ligações entre os pontos é feito calculando a distância entre esses dois pontos. Na Figura 6.14, é mostrada a ligação do ponto 1 ao 2.

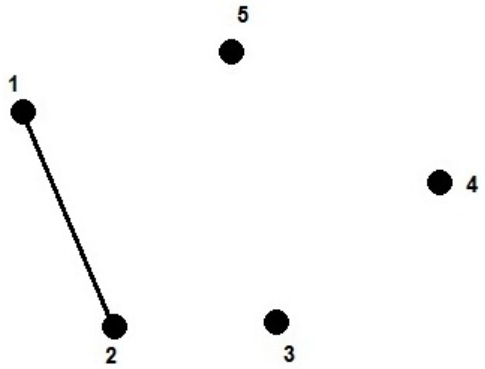

Figura 6.14: Ligação do ponto 1 ao 2.

Na Figura 6.15 é mostrado a ligação do ponto 2 ao 3.

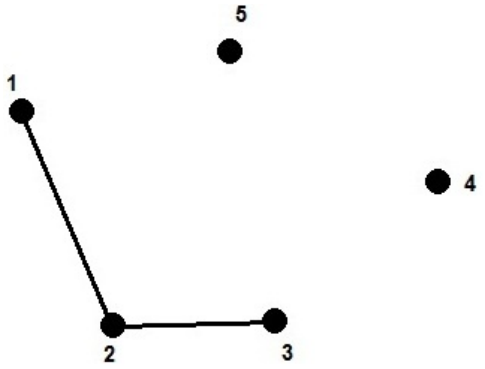

Figura 6.15: Ligação do ponto 2 ao 3.

Para formar um triângulo o ponto 3 é ligado ao ponto 1, conforme a Figura 6.16. Assim a área do primeiro triângulo é obtida.

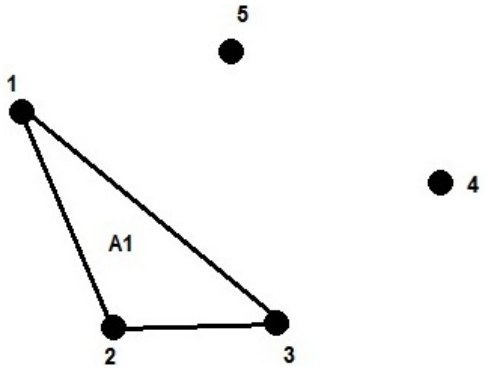

Figura 6.16: Ligação do ponto 3 ao 1.

Depois de calculada a área do primeiro triângulo o algoritmo liga o primeiro ponto ao ponto 3, conforme a Figura 6.17.

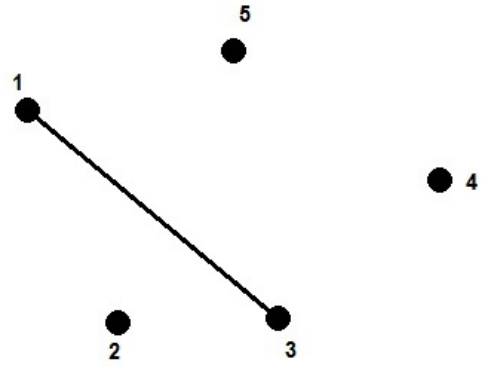

Figura 6.17: Ligação do ponto 1 ao 3.

Então é usada a mesma lógica para calcular a área do segundo triângulo, conforme as Figuras 6.18 e 6.19.

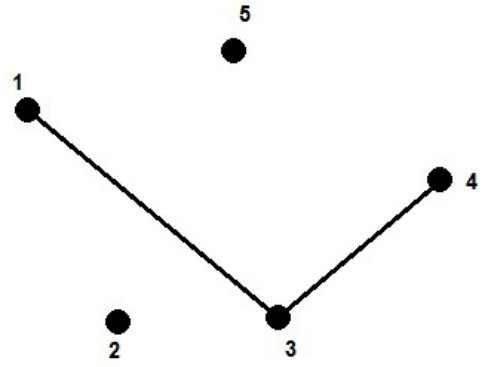

Figura 6.18: Ligação do ponto 3 ao 4.

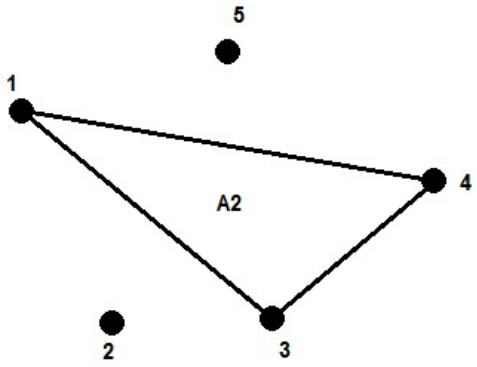

Figura 6.19: Ligação do ponto 4 ao 1.

O algoritmo repete a mesma lógica até calcular a área de todos os triângulos formados através dos pontos do terreno. As Figuras 6.20, 6.21 e 6.22 mostram o cálculo do terceiro triângulo.

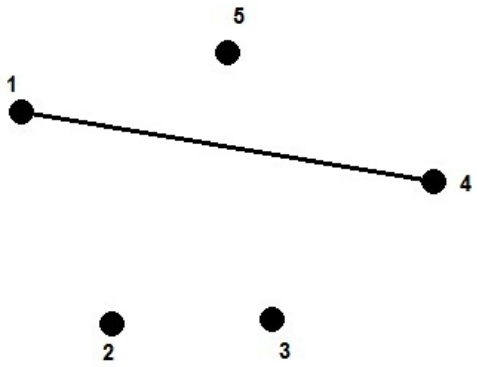

Figura 6.20: Ligação do ponto 1 ao 4.

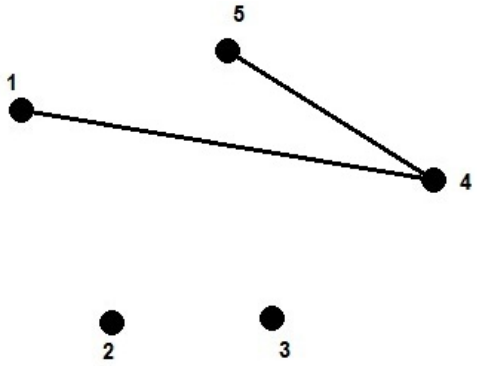

Figura 6.21: Ligação do ponto 4 ao 5.

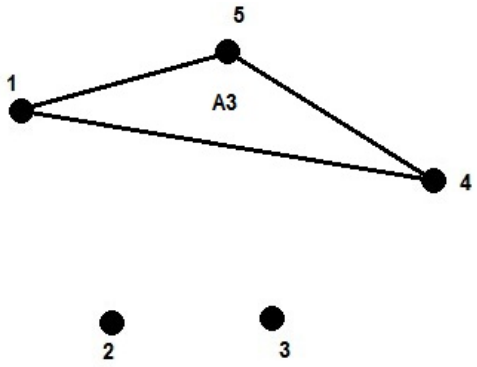

Figura 6.22: Ligação do ponto 5 ao 1.

A área do terreno é obtida somando as áreas de todos os triângulos, conforme a Figura 6.23.

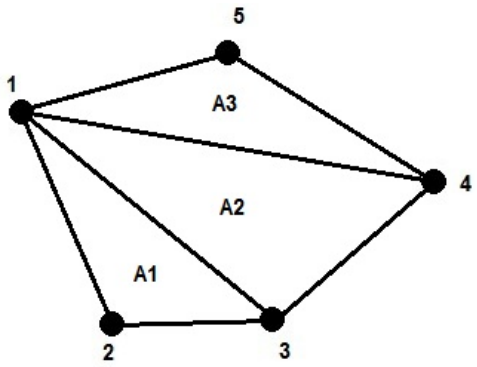

Figura 6.23: Área do terreno

6.4 Código do Aplicativo

Nessa seção serão abordados os principais métodos do protótipo do aplicativo proposto nesse trabalho. Será apresentada a implementação do aplicativo para o iPad, pois a tela deste é maior que a do iPhone, assim foi necessário a implementação de apenas duas classes, uma para cada tela do aplicativo. Como não há funcionalidades na tela inicial do aplicativo, nessa seção será explicado apenas a classe SegundaTelaiPad.m que está vinculada a tela onde estão as funcionalidades do aplicativo.

O mesmo aplicativo foi implementado para iPhone sem dificuldades, a única diferença foi a quantidade de classes implementadas. Como a tela do iPhone é menor que a do iPad, foi necessário implementar mais classes para o primeiro, pois o aplicativo possui várias funcionalidades e estas foram implementadas em classes separadas devido a falta de espaço na tela. Lembrando que para cada tela há uma classe vinculada. Os métodos explicados nessa seção são os mesmos implementados para o iPad e o iPhone.

6.4.1 SegundaTelaiPad.m

Os principais métodos da classe SegundaTelaiPad.m são: locationManager, onRoff, salvarPontoGps, salvarPontoManual, grausRadiano, calculaDistancia e calculaAreaSementes.

locationManager

Esse método é responsável por obter as coordenadas do usuário por meio do GPS do dispositivo, atualizando-as automaticamente conforme o usuário se desloca no terreno.

Na linha 1 do Código 6.1, o objeto *newLocation* do tipo CLLocation que é responsável por obter as coordenadas geográficas é passado como argumento ao método locationManager. A variável *graus* na linha 3, recebe a coordenada da latitude em ponto flutuante. Na linha 4, o conteúdo da variável *graus* é armazenado na variável global *latAux* que será utilizada por outros métodos. Na linha 5, é instanciado um objeto do tipo NSString chamado *lat* que é inicializado com o valor da latitude que está armazenado na variável *graus*. Nessa linha, o valor da variável *graus* em ponto flutuante é convertida para o formato de *string*. Na linha 6, o objeto *latLabel* recebe o valor da latitude. Esse objeto está vinculado a um componente na interface gráfica de usuário que é responsável por mostrar a latitude atual do GPS na tela do dispositivo. Nas linhas 7, 8, 9 e 11, é feito o mesmo, com o valor da longitude.

Código 6.1: Método locationManager.

```
1  - (void)locationManager: didUpdateToLocation:(CLLocation *)
       newLocation
2  {
3      double graus = newLocation.coordinate.latitude;
4      latAux = graus;
5      NSString *lat = [NSString stringWithFormat:@"%f", graus];
6      latLabel.text = lat;
7      graus = newLocation.coordinate.longitude;
8      longAux = graus;
9      NSString *longt = [NSString stringWithFormat:@"%f",
10                        graus];
11     longLabel.text = longt;
12     }
```

onRoff

Esse método é responsável por ligar/desligar o GPS do dispositivo, pois este consome muita bateria quando está ligado.

Na linha 3 do Código 6.2, há uma condição onde é verificado se o componente *switch* da interface gráfica que está vinculado ao objeto *meuSwitch* no código-fonte do aplicativo está ligado. Se isso for verdade, na linha 5 o método startUpdatingLocation do objeto *locationManager* liga o GPS do dispositivo. Caso a condição na linha 3 for falsa, o método stopUpdatingLocation do objeto *locationManager* na linha 8 desliga o GPS do dispositivo.

Código 6.2: Método onRoff.

```
1  -(void) onRoff
2  {
3      if(meuSwitch.on)
4      {
5          [locationManager startUpdatingLocation];
6      }else
7      {
8          [locationManager stopUpdatingLocation];
9      }
10 }
```

salvarPontoGps

Esse método é responsável por salvar os pontos que se obtêm por meio do GPS do dispositivo em dois vetores. Esses dados podem ser utilizados por outros métodos para calcular a área do terreno.

O tipo de retorno -(IBAction) na linha 1 do Código 6.3, indica que o método salvarPontoGps será vinculado a alguma ação provinda da interface gráfica de usuário. No caso do aplicativo proposto nesse trabalho o método salvarPontoGps está vinculado a um botão na interface gráfica de usuário chamado "Salvar Ponto GPS".

Na linha 3, a variável *tamVet* é incrementada. Essa variável global armazena a quantidade de pontos armazenados nos vetores, outros métodos utilizam essa variável. Na linha 4, é instanciado um objeto do tipo NSString chamado *latitude* que é inicializado com o valor da latitude que está armazenado na variável global *latAux*. Nessa linha, o valor da variável *latAux* em ponto flutuante é convertida para o formato de *string*.

Na linha 5, o método addObject do objeto *array* adiciona o valor da latitude no vetor *array*. Na linha 6 e 7, é feito o mesmo com a longitude, só que seu valor é armazenado no vetor *array*2. Na linha 8, a variável global *indice* é incrementada, por padrão o valor dessa variável é 0. Na linha 9, é instanciado um objeto do tipo NSString chamado *aux* que é inicializado com o valor da variável *indice*. Nessa linha o valor da variável *indice* do tipo inteiro é convertido para o formato de *string*. Na linha 10, o objeto *pontoLabel*

recebe o valor da variável *aux*. Esse objeto está vinculado a um componente na interface gráfica de usuário que é responsável por mostrar o número do ponto salvo.

Na linha 11, é instanciado um objeto do tipo UIAlertView chamado *alerta* que é responsável por abrir uma janela informando a mensagem: "As coordenadas foram salvas". Na linha 12, o método *show* do objeto *alerta* é invocado para mostrar a mensagem ao usuário.

Código 6.3: Método salvarPontoGps.

```
1  -(IBAction) salvarPontoGps
2  {
3        tamVet++;
4        NSString *latitude = [NSString stringWithFormat:@"%f", latAux];
5        [array addObject:latitude];
6        NSString *longitude = [NSString stringWithFormat:@"%f", longAux
             ];
7       [array2 addObject:longitude];
8        indice++;
9        NSString *aux = [NSString stringWithFormat:@"%d",indice];
10       pontoLabel.text = aux;
11      UIAlertView *alerta = [[UIAlertView alloc] initWithTitle:@"Aviso"
            message:@"As coordenadas foram salvas" delegate:nil
          cancelButtonTitle:@"O.K" otherButtonTitles:nil];
12      [alerta show];
13 }
```

salvarPontoManual

Esse método é responsável por salvar os pontos que se obtêm através das coordenadas fornecidas pelo o usuário em dois vetores. Esses dados podem ser utilizados por outros métodos para calcular a área do terreno.

O tipo de retorno -(IBAction) na linha 1 do Código 6.4, indica que o método salvarPontoManual será vinculado a alguma ação provinda da interface gráfica de usuário. No caso do aplicativo proposto nesse trabalho o método salvarPontoManual está vinculado a um botão na interface gráfica de usuário chamado "Salvar Ponto Manual".

Nas linhas 3 e 4, são declaradas as variáveis x e y do tipo $float$ que recebem os valores da latitude e longitude em ponto flutuante respectivamentes. Há uma condição na linha 5 que verifica os valores de x e y, isso evita que o usuário entre com qualquer valor que não seja numérico. Caso o(s) valor(es) de x e/ou y seja(m) 0, na linha 7 é instanciado um objeto do tipo UIAlertView chamado *alerta*6 que é responsável por abrir uma janela com

uma mensagem informando ao usuário que o(s) valor(es) da latitude e/ou da longitude está(ão) incorreto(s). Na linha 8, o método *show* do objeto *alerta*6 é invocado para mostrar a mensagem ao usuário. Nas linhas 9 e 10, os campos na interface gráfica de usuário responsáveis por obter os valores da latitude e longitude informados pelo usuário manualmente, recebem uma *string* vazia, isso faz com que os campos fiquem limpos para que o usuário possa entrar com outros valores.

Caso a condição na linha 5 seja falsa, então o programa executa a condição da linha 13 que verifica se o valor da latitude é menor do que -90 ou maior do que 90 , e /ou valor da longitude é menor do que -180 ou maior do que 180. Se isso for verdade, na linha 15 é instanciado um objeto do tipo UIAlertView chamado *alerta*7 que é responsável por abrir uma janela informando a mensagem: "Valor da latitude menor do que -90 ou maior do que 90 , e /ou valor da longitude menor do que -180 ou maior do que 180.". Na linha 16, o método *show* do objeto *alerta*7 é invocado para mostrar a mensagem ao usuário. Nas linhas 17 e 18, os campos na interface gráfica de usuário responsáveis por obter os valores da latitude e longitude são limpos.

Se a condição da linha 13 for falsa, então na linha 21, a variável $tamVet$ é incrementada. Essa variável global armazena a quantidade de pontos armazenados nos vetores, outros métodos utilizam essa variável. Na linha 22, o método addObject do objeto *array* adiciona o valor da latitude armazenada no objeto campoLatitude no vetor *array*. O objeto campoLatitude está vinculado a um componente na interface gráfica de usuário responsável por obter o valor da latitude informada pelo usuário manualmente. Na linha 23 é feito o mesmo com a longitude, só que seu valor é armazenado no vetor *array*2. Na linha 24, a variável global *indice* é incrementada, por padrão o valor dessa variável é 0. Na linha 25, é instanciado um objeto do tipo NSString chamado *aux* que é inicializado com o valor da variável *indice*. Nessa linha, o valor da variável *indice* do tipo inteiro é convertida para o formato de *string*. Na linha 26, o objeto *pontoLabel* recebe o valor da variável *aux*. Esse objeto está vinculado a um componente na interface gráfica de usuário que é responsável por mostrar o número do ponto salvo.

Na linha 27, é instanciado um objeto do tipo UIAlertView chamado *alerta*4 que é responsável por abrir uma janela informando a mensagem: “As coordenadas foram salvas”. Na linha 28, o método *show* do objeto *alerta*4 é invocado para mostrar a mensagem ao usuário. Nas linhas 29 e 30, os campos na interface gráfica de usuário responsáveis por obter os valores da latitude e longitude são limpos.

Código 6.4: Método salvarPontoManual.

```
1 -(IBAction) salvaPontoManual
2 {
3     float x = [campoLatitude.text doubleValue];
```

```
4       float y = [campoLongitude.text doubleValue];
5       if(x==0 || y==0)
6       {
7           UIAlertView *alerta6 = [[UIAlertView alloc] initWithTitle:@"
                Aviso" message:@"Valor(es) da latitude e/ou da longitude
                incorreto(s)." delegate:nil cancelButtonTitle:@"O.K"
                otherButtonTitles:nil];
8           [alerta6 show];
9           campoLatitude.text = @"";
10          campoLongitude.text = @"";
11      }else
12      {
13          if(x < -90 || x > 90 || y < -180 || y > 180)
14          {
15              UIAlertView *alerta7 = [[UIAlertView alloc] initWithTitle
                    :@"Aviso" message:@"Valor da latitude menor do que -90
                     ou maior do que 90,  e/ou valor da longitude menor do
                     que -180 ou maior do que 180." delegate:nil
                    cancelButtonTitle:@"O.K" otherButtonTitles:nil];
16              [alerta7 show];
17              campoLatitude.text = @"";
18              campoLongitude.text = @"";
19          }else
20          {
21              tamVet++;
22              [array addObject:campoLatitude.text];
23              [array2 addObject:campoLongitude.text];
24              indice++;
25              NSString *aux = [NSString stringWithFormat:@"%d",indice];
26              pontoLabel.text = aux;
27              UIAlertView *alerta4 = [[UIAlertView alloc] initWithTitle
                    :@"Aviso" message:@"As coordenadas foram salvas"
                    delegate:nil cancelButtonTitle:@"O.K"
                    otherButtonTitles:nil];
28              [alerta4 show];
29              campoLatitude.text = @"";
30              campoLongitude.text = @"";
31          }
32      }
33  }
```

grausRadiano

Esse método é responsável por transformar graus em radiano. Outros métodos que auxiliam no cálculo de área utilizam esse método.

Na linha 1 do Código 6.5, o método recebe os graus como parâmetro, na linha 3 é feito o cálculo que transforma graus em radiano, depois é retornado o valor em radiano.

Código 6.5: Método grausRadiano.

```
1 -(double) grausRadiano:(double) graus
2 {
3     return graus * (3.14159265/180);
4 }
```

calculaDistancia

Esse método é responsável por calcular a distância entre dois pontos.

Na linha 1 do Código 6.6, o método recebe quatro parâmetros do tipo *float* que são as latitudes e as longitudes dos dois pontos que se deseja calcular a distância entre eles. Na linha 3, é declarado uma variável chamada *raioTerra* do tipo *double* que recebe o raio aproximado da Terra em quilômetros. Na linha 4, é declarado a variável *deltaLat* do tipo *double* que recebe a diferença da latitude do ponto 2 pela a latitude do ponto 1 em radianos. Na linha 5, é feito o mesmo com a longitude.

Na linha 6, a variável *lat*1 provinda do parâmetro, recebe a latitude do ponto 1 que está armazenada nessa mesma variável em radianos. Na linha 7, é feito o mesmo com a variável *lat*2 que armazena a latitude do ponto 2. Na linha 8, 9 e 10 é feito o cálculo que encontra a distância entre os dois pontos. Na linha 11, é retornado a distância entre os dois pontos em ponto flutuante.

Código 6.6: Método calculaDistancia.

```
1 -(float) calculaDistancia: (float) lat1 : (float) lon1 : (float) lat2
      : (float) lon2
2 {
3     double raioTerra = 6367;
4     double deltaLat =  [self grausRadiano:(lat2 - lat1)];
5     double deltaLon = [self grausRadiano:(lon2 - lon1)];
6     lat1 =  [self grausRadiano: lat1];
7     lat2 =  [self grausRadiano: lat2];
8     double A =  pow (sin(deltaLat/2), 2) + cos(lat1) * cos(lat2) *
          pow (sin(deltaLon/2), 2);
9     double C = 2 * atan2(sqrt(A), sqrt(1 - A));
```

```
10      double D = raioTerra * C;
11      return D;
12  }
```

calculaAreaSementes

Esse método é responsável por calcular a área da figura formada pelos pontos, a quantidade de covas e o número de sementes necessárias para o plantio. Segundo Costa e Campos (1997), é necessário obedecer a um espaçamento de 100 cm entre fileiras e 40 cm entre covas, com 2 sementes por cova.

O tipo de retorno -(IBAction) na linha 1 do Código 6.7, indica que o método calculaAreaSementes será vinculado a alguma ação provinda da interface gráfica de usuário. No caso do aplicativo proposto nesse trabalho, o método calculaAreaSementes está vinculado a um botão na interface gráfica de usuário chamado "Calcular".

Na linha 3, são declaradas as variáveis p, $area$, a, b e c do tipo *float* que auxiliam no cálculo de área. Na linha 4, é declarado a variável $areaTotal$ do tipo *float* que recebe o valor 0. Essa variável armazena a área total do terreno.

Na linha 5, há uma condição que verifica se a quantidade de pontos armazenada pelo usuário é menor do que três. Caso seja verdade, então na linha 7, é instanciado um objeto do tipo UIAlertView chamado $alerta2$ que é responsável por abrir uma janela informando a mensagem: "Os pontos não são suficientes para calcular a área, entre com 3 ou mais pontos". Na linha 8, o método *show* do objeto $alerta2$ é invocado para mostrar a mensagem ao usuário.

Caso a condição da linha 5 seja falsa, os comandos que estão dentro do laço for na linha 11 serão executados n - 2 vezes, onde n é a quantidade de pontos. Na linha 13, a variável a recebe a distância entre o ponto que está armazenado no índice 0 e o ponto que está armazenado no índice i + 1 dos vetores $array$ e $array2$. Na linha 14, a variável b recebe a distância entre o ponto que está armazenado no índice i + 1 e o ponto que está armazenado no índice i + 2 dos vetores $array$ e $array2$. Na linha 15, a variável c recebe a distância entre o ponto que está armazenado no índice i + 2 e o ponto que está armazenado no índice 0 dos vetores $array$ e $array2$. Assim essas distâncias quando ligadas pelos pontos formam um triângulo. A área do triângulo é calculada utilizando o teorema de Herão que está implementado na linha 17, onde a, b e c são os lados do triangulo e p o semiperímetro que é obtido na linha 16 pela a soma de a, b, e c divido por 2. Na linha 18, a variável $areaTotal$ é somada com a variável $area$ obtida em cada iteração do laço, o resultado é armazenado na própria variável $areaTotal$. Isso é feito até que o comando chegue ao fim.

Na linha 20, a área total armazenada na variável $areaTotal$ é multiplicado por 1000000,

transformando o resultado de quilômetros quadrados para metros quadrados. Na linha 21, é declarado a variável l do tipo *double* que recebe o valor do espaçamento entre fileiras de plantio em centímetros. Na linha 22, é declarado a variável w do tipo *double* que recebe o valor do espaçamento entre covas em centímetros. Na linha 23, é declarado a variável x do tipo *double* que recebe o cálculo que encontra a quantidade de covas necessárias para o plantio. O cálculo consiste em transformar a área que está em metros quadrados para centímetros quadrados, esse valor é dividido pelo o produto de l por w.

Na linha 24, é declarado a variável *quantidadeCovas* que recebe a quantidade de covas armazenada na variável x. Na linha 25, é declarado a variável *quantidadeSementes* que recebe o resultado do cálculo que encontra a quantidade de sementes necessárias para o plantio. O cálculo consiste em multiplicar a quantidade de covas por 2, pois para cada cova joga-se duas sementes de milho. Na linha 26, é instanciado um objeto do tipo NSString chamado *resultadoFinal* que recebe as informações que serão mostradas ao usuário, nesse caso, a área do terreno e a quantidade de covas e sementes. Na linha 27, é instanciado um objeto do tipo UIAlertView chamado *alerta*3 que é responsável por abrir uma janela informando ao usuário a área do terreno e a quantidade de covas e sementes necessárias para o plantio. Na linha 28, o método *show* do objeto *alerta*3 é invocado para mostrar a mensagem ao usuário.

Código 6.7: Método calculaAreaSementes.

```
1  -(IBAction) calculaAreaSementes
2  {
3      float p,area,a,b,c;
4      float areaTotal = 0;
5      if(tamVet < 3)
6      {
7          UIAlertView *alerta2 = [[UIAlertView alloc] initWithTitle:@"
               Aviso" message:@"Os pontos não são suficientes para
               calcular o área, entre com 3 ou mais pontos" delegate:nil
               cancelButtonTitle:@"O.K" otherButtonTitles:nil];
8          [alerta2 show];
9      }else
10     {
11         for(int i = 0; i < tamVet-2; i++)
12         {
13             a = [self calculaDistancia:[[array objectAtIndex:0]
                   doubleValue] :[[array2 objectAtIndex:0] doubleValue]
                   :[[array objectAtIndex:i+1]doubleValue] :[[array2
                   objectAtIndex:i+1] doubleValue]];
```

```
14              b = [self calculaDistancia:[[array objectAtIndex:i+1]
                    doubleValue] :[[array2 objectAtIndex:i+1] doubleValue]
                     :[[array objectAtIndex:i+2]doubleValue] :[[array2
                    objectAtIndex:i+2] doubleValue]];
15              c = [self calculaDistancia:[[array objectAtIndex:i+2]
                    doubleValue] :[[array2 objectAtIndex:i+2] doubleValue]
                     :[[array objectAtIndex:0]doubleValue] :[[array2
                    objectAtIndex:0] doubleValue]];
16              p = (a + b + c)/2;
17              area = sqrt(p * (p - a) * (p - b) * (p - c));
18              areaTotal = areaTotal + area;
19          }
20          areaTotal = areaTotal * 1000000;
21          double l = 100; // espaçamento entre fileiras em cm.
22          double w = 40;  // espaçamento entre covas em cm.
23          double x = (areaTotal * 100) / (l * w);
24          int quantidadeCovas = x;
25          int quantidadeSementes = quantidadeCovas * 2;
26          NSString *resultadoFinal = [[NSString alloc] initWithFormat:@
                "       Área:   %f.\n       Covas: %d.\nSementes: %d.",
                areaTotal, quantidadeCovas, quantidadeSementes];
27          UIAlertView *alerta3 = [[UIAlertView alloc] initWithTitle:@"
                Aviso" message:resultadoFinal delegate:nil
                cancelButtonTitle:@"O.K" otherButtonTitles:nil];
28          [alerta3 show];
29      }
30  }
```

6.5 Diagramas de Classe

Nesta seção serão mostrados os diagramas de classe da implementação do protótipo do aplicativo para o iPad e iPhone. Na Figura 6.1 é ilustrado o diagrama de classe do protótipo do aplicativo para o iPad, onde a classe AppDelegate é responsável por controlar as telas do aplicativo. A classe ViewController está vinculada com a tela inicial do aplicativo, nesta classe há apenas um botão que é utilizado para ir para segunda tela. A classe SegundaTelaiPad está vinculada a segunda tela do aplicativo, onde estão todas as funcionalidades deste.

Figura 6.24: Diagrama de classe do aplicativo para o iPad

Na Figura 6.2 é ilustrado o diagrama de classe do protótipo do aplicativo para o iPhone, onde a classe AppDelegate é responsável por controlar as telas do aplicativo. A classe ViewController está vinculada com a tela inicial do aplicativo, onde há dois botões que são utilizados para ir para as telas onde estão as funcionalidades do aplicativo, estas telas estão vinculadas as classes PontosPorGPS e PontosManuais.

Figura 6.25: Diagrama de classe do aplicativo para o iPhone

6.6 Resultados e Testes

Durante o desenvolvimento do protótipo do aplicativo foram realizados vários testes, principalmente testes de entrada e saída de dados. Nos testes foi constatado erro ao fornecer caracteres não numéricos nos campos de entrada manual de coordenadas geográficas do aplicativo. Este erro foi corrigido transformando os textos dos campos de *string* para *double*, caso o valor dessa conversão resulte em 0, o sistema informa ao usuário que o valor de uma das coordenadas ou das duas estão incorretos, desta forma o sistema só aceita caracteres numéricos nesses campos.

Outros erros percebidos foram entradas de latitudes e longitudes fora das faixas de valores. O valor para latitude vai de -90 a 90 e para longitude vai de -180 a 180. Nos testes, quaisquer outros valores fora desses intervalos resultavam em erros no aplicativo. A solução foi implementar uma condição onde os valores das coordenadas são testadas para verificar se estes estão nos intervalos citado acima.

Foi encontrado um erro no botão "Calcular" quando este era tocado, antes que pelo menos 3 pontos fossem fornecidos ao aplicativo. Esse erro foi corrigido verificando a quantidade de pontos armazenados, caso essa quantidade fosse menor do que três, o sistema informava uma mensagem ao usuário, pedindo que entre com pelo menos 3 pontos. Nenhum erro foi encontrado nos outros botões do protótipo do aplicativo.

Os testes que envolvem a precisão do cálculo de área do protótipo mostram resultados satisfatórios. Estes testes foram realizados na UFG câmpus Catalão, onde se comparou a área do bloco J contida na planta da UFG câmpus Catalão com a área do bloco J obtida através do protótipo. Os resultados desses testes são mostrados na Tabela 6.1.

Tabela 6.1: Resultados dos testes de precisão do GPS

Testes	**Área do bloco J contida na planta da UFG câmpus Catalão**	**Área do bloco J obtida através do protótipo**	**Área do bloco J contida na planta da UFG campus Catalão menos a Área do bloco J obtida através do protótipo**	**Desvio do resultado do cálculo da área em porcentagem**
1	1056,38 metros quadrados	1097,61 metros quadrados	41,23 metros quadrados	3,9%
2	1056,38 metros quadrados	1074,50 metros quadrados	18,12 metros quadrados	1,7%
3	1056,38 metros quadrados	1084,9 metros quadrados	28,52 metros quadrados	2,4%
4	1056,38 metros quadrados	1076,45 metros quadrados	20,06 metros quadrados	1,9%
5	1056,38 metros quadrados	1093,35 metros quadrados	36,96 metros quadrados	3,5%
6	1056,38 metros quadrados	1108,72 metros quadrados	52,34 metros quadrados	4,9%
7	1056,38 metros quadrados	1100,75 metros quadrados	44,37 metros quadrados	4,2%
8	1056,38 metros quadrados	1078,56 metros quadrados	22,18 metros quadrados	2,1%

Como mostra a Tabela 6.1 os resultados obtidos nos testes apontam que a margem de erro na precisão do cálculo da área pelo GPS do dispositivo ficou entre 1,7% e 4,9% para mais. Porém, pode-se considerar como satisfatório devido à grande influência das interferências eletromagnéticas e obstáculos físicos encontrados na zona urbana.

Não foram realizados testes na zona rural devido à falta de documentação de áreas ru-

rais para comparação dos resultados que seriam obtidos pelo aplicativo, porém pretende-se realizar esses testes em trabalhos futuros e espera-se que os resultados sejam melhores que os obtidos na zona urbana, já que a zona rural apresenta menos interferências eletromagnéticas e obstáculos físicos.

Testes mostraram que o protótipo do aplicativo exige uma boa durabilidade da bateria do dispositivo, pois é necessário deixar o GPS ligado para que este faça a coleta dos dados. Quando se comparou a autonomia da bateria dos dispositivos da Apple Inc. com a de seus concorrentes, pode-se observar uma larga vantagem destes que se portaram muito bem, podendo chegar a 2 horas no iPad se o aplicativo for utilizado sem intervalos de tempo.

Capítulo 7

Manual do Aplicativo

Este capítulo tem como objetivo explicar as duas versões que foram desenvolvidas do protótipo do aplicativo proposto nesse trabalho, sendo uma para o iPad e outra para o iPhone.

7.1 Manual do aplicativo para o iPad

Nessa seção será explicado como funciona a versão do protótipo do aplicativo para o iPad.

A tela inicial do aplicativo é mostrada na Figura 7.1, onde há o nome do aplicativo chamado "iFarm" e o botão INICIAR. O aplicativo é iniciado quando o usuário toca esse botão, mudando para a segunda tela.

Figura 7.1: Tela inicial do aplicativo

Na segunda tela estão todas as funcionalidades do aplicativo conforme a Figura 7.2.

Figura 7.2: Segunda tela do aplicativo

As funcionalidades estão numeradas de 1 a 8 conforme a Figura 7.2 e serão explicadas a seguir:

1. Esse componente é responsável por ligar/desligar o GPS. Na Figura 7.2 o GPS encontra-se ligado e na Figura 7.3 desligado.

2. Marca quantos pontos foram armazenados.

3. Mostram as coordenadas atuais da latitude e da longitude do GPS. Caso o usuário se desloque no espaço esses valores são atualizados automaticamente.

4. Campos de entrada manual de coordenadas, onde há um campo para a latitude e outro para a longitude. Caso o usuário deseje entrar com as coordenadas obtidas de outras fontes, como por exemplo, o Google Maps ou outro GPS.

5. Botão que salva as coordenadas do GPS do dispositivo. Depois que as coordenadas forem salvas aparecerá uma mensagem para o usuário como mostrado na Figura 7.4.

6. Botão que salva as coordenadas que usuário fornece nos campos de entrada manual. Depois que as coordenadas forem salvas aparecerá uma mensagem para o usuário como mostrado na Figura 7.4.

Figura 7.3: GPS desligado

7. Botão que calcula a área do terreno e quantidade de covas e sementes necessárias para o plantio. Caso o usuário toque nesse botão antes de coletar pelo menos três pontos, o sistema informa a mensagem mostrada na Figura 7.5.

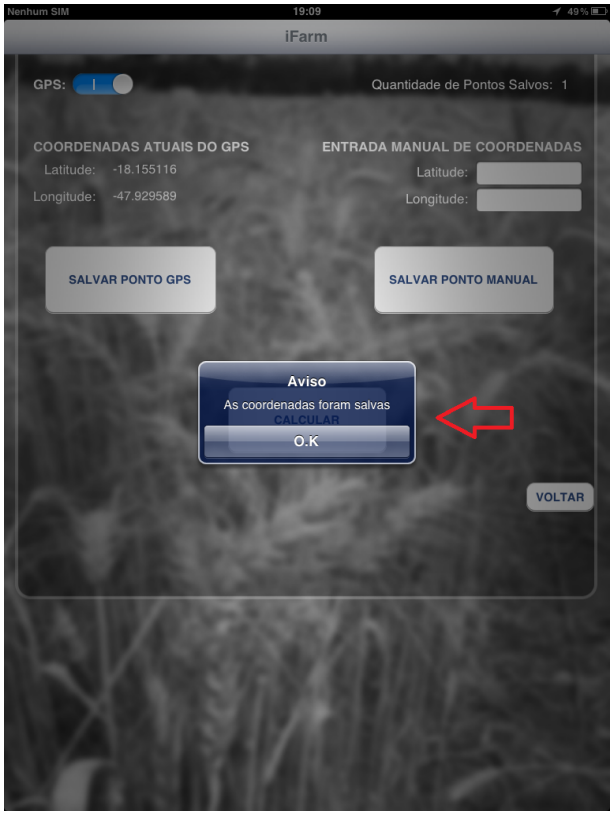

Figura 7.4: Mensagem informando ao usuário que as coordenadas foram salvas

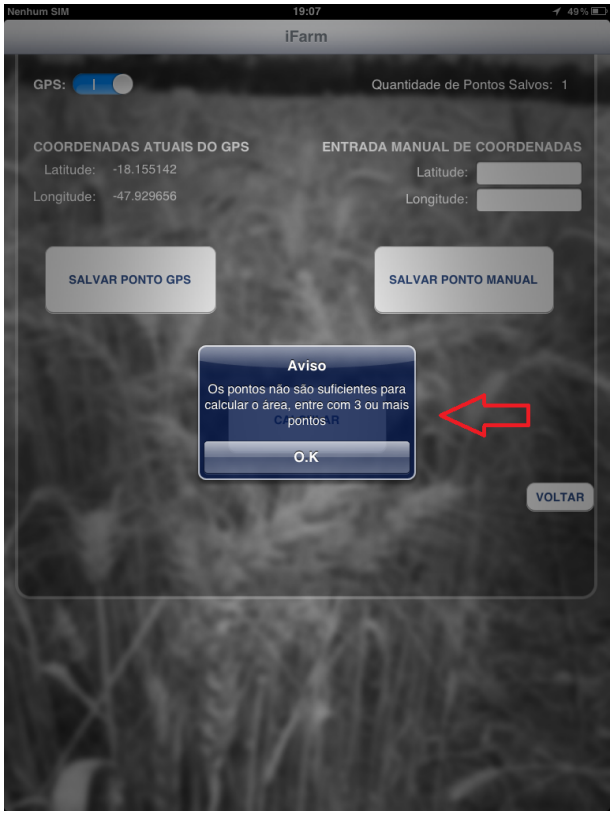

Figura 7.5: Mensagem informando ao usuário que entre com pelo menos três pontos

Se o usuário coletar três ou mais pontos, o sistema informa a área em metros quadrados e a quantidade de covas e sementes, como mostrado na Figura 7.6.

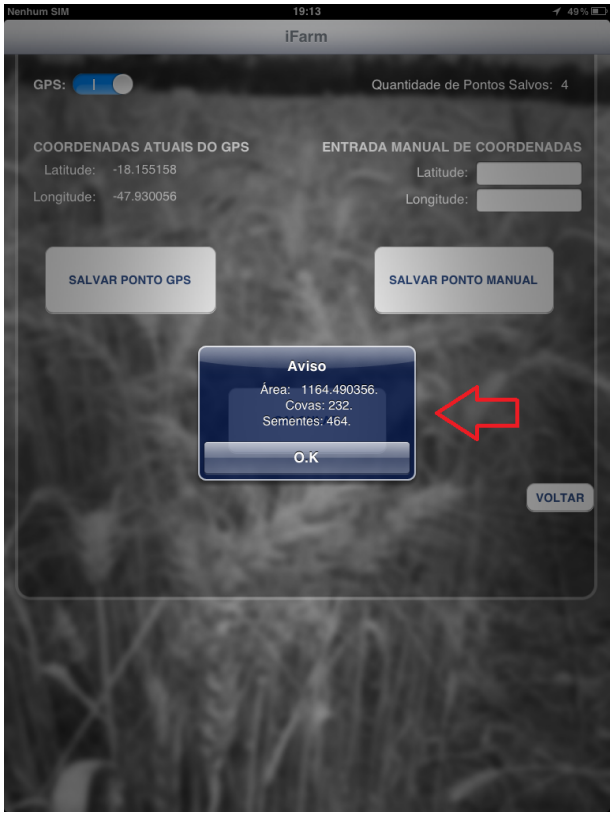

Figura 7.6: Mensagem informando ao usuário a área em metros quadrado e a quantidade de covas e sementes.

8. Botão utilizado para voltar à tela inicial do aplicativo.

7.2 Manual do aplicativo para o iPhone

Nessa seção será explicado como funciona a versão do protótipo do aplicativo para o iPhone.

A tela inicial do aplicativo é mostrada na Figura 7.7, onde há o nome do aplicativo, o botão GPS e o botão MANUAL. Nesta tela o usuário tem duas opções de fornecer as coordenadas para o aplicativo, através do GPS do dispositivo ou da entrada manual.

Figura 7.7: Tela inicial do aplicativo

Caso o usuário toque o botão GPS, será exibido à tela mostrada na Figura 7.8.

Figura 7.8: Tela Ponto GPS

As funcionalidades estão numeradas de 1 a 6 conforme a Figura 7.8 e serão explicadas a seguir:

1. Esse componente é responsável por ligar/desligar o GPS.

2. Marca quantos pontos foram armazenados.

3. Mostram as coordenadas atuais da latitude e da longitude do GPS. Caso o usuário se desloque no espaço esses valores atualizam automaticamente.

4. Botão que salva as coordenadas do GPS do dispositivo. Após as coordenadas serem salvas, aparecerá uma mensagem para o usuário semelhante à mostrada na Figura 7.4.

5. Botão que calcula a área do terreno, quantidade de covas e sementes necessárias para o plantio. Caso o usuário toque nesse botão antes de coletar pelo menos três pontos, o sistema informa a mensagem semelhante à mostrada na Figura 7.5. Se o usuário coletar três ou mais pontos, o sistema informa a área e a quantidade de covas e sementes semelhante à mostrada na Figura 7.6.

6. Botão utilizado para voltar à tela inicial do aplicativo.

Caso o usuário toque no botão MANUAL na tela inicial, será exibido à tela mostrada na Figura 7.9 que é semelhante à tela na Figura 7.8, a única diferença é que na tela da Figura 7.9, o usuário entra manualmente com as coordenadas da latitude e da longitude nos campos indicados pela seta.

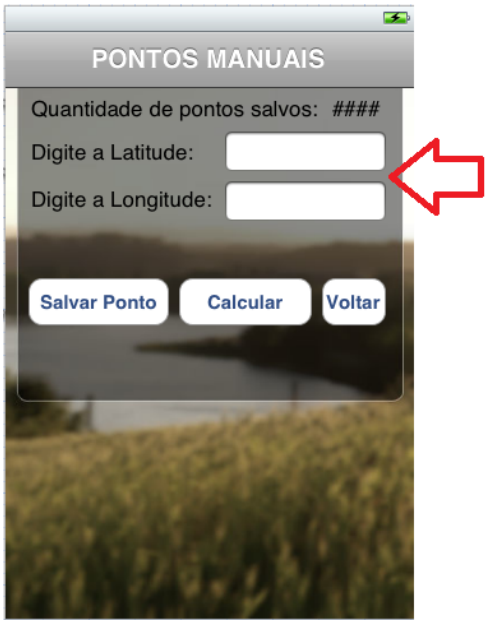

Figura 7.9: Tela de entrada manual de coordenadas

Capítulo 8

Conclusão

Nas próximas versões do aplicativo pretende-se aumentar o número de culturas que o produtor pode escolher, tendo também como opção algodão, soja, feijão e café. Além disso, serão implementadas outras funcionalidades, tais como: a quantidade aproximada de água necessária para irrigar a plantação; função que desconsidera as áreas improdutivas (represas, casas, reservas florestais, etc.); função que mostra a precisão do GPS.

Será feito um estudo para encontrar o equilíbrio entre a precisão do GPS e a autonomia da bateria dos dispositivos, pois a função que fornece a melhor precisão do GPS consome muita bateria. Espera-se encontrar uma função que fornece uma precisão que atende de forma satisfatória as necessidades do aplicativo e que consome menos bateria.

Também será melhorado o algoritmo que calcula a área do terreno, pois o algoritmo desenvolvido nesse trabalho só trabalha com polígonos convexos.

Pretende-se portar o mesmo sistema para famílias de dispositivos com o sistema operacional Android da Google, uma vez que tais dispositivos estão se tornando cada vez mais populares e já fazem parte de equipamentos com funcionalidades e autonomia similares aos da família Apple Inc..

Algumas dificuldades foram encontradas durante o desenvolvimento desse trabalho que se iniciou com quatro meses de pesquisa, que envolvia a escolha da tecnologia que mais atendia as necessidades do aplicativo e que fornecia toda a documentação para o desenvolvimento deste. Também foi necessário pesquisar sobre qual dispositivo atenderia os requisitos sobre precisão do GPS e autonomia da bateria.

Após o fim das pesquisas, foi constatado que a melhor opção seria a dos dispositivos móveis da Apple Inc.. Devido a esta escolha foi necessário obter conhecimentos sobre a linguagem de programação Objetive-C, que é uma linguagem que possui sintaxe diferente das estudadas durante o curso de Ciências da Computação da Universidade Federal de Goiás - Campus Catalão.

Outra dificuldade relevante foi o fato de que o departamento de Ciências da Computação da Universidade não possuía os recursos necessários para o desenvolvimento do

aplicativo, uma vez que para se instalar o iOS SDK precisou-se de um computador da Apple Inc.. Durante o desenvolvimento do aplicativo também foi necessário a utilização de um iPad e um iPhone para realizar os testes, até a aquisição destes recursos ocorreram atrasos durante o desenvolvimento.

O protótipo do aplicativo atendeu de forma satisfatória os objetivos propostos no projeto. Os testes realizados indicaram bons resultados no que envolve a precisão do cálculo de área, covas e sementes.

Pode-se perceber que o aplicativo poderá auxiliar os pequenos e médios produtores rurais em suas atividades agrícolas. Já que este possui recursos necessários para o cálculo da área a ser plantada, bem como o número de covas e sementes necessárias para o plantio de milho. Desta forma os produtores rurais poderão utilizar estas funcionalidades para otimizar sua produção, reduzindo os gastos com mão de obra qualificada.

Referências

ALVES, S. *A matemática do GPS*. São Paulo, SP, Brasil, 1996.

ANHOLETTO, A. T.; FERNANDES, R. A. G. *Proposta Preliminar de Arquitetura para Implementação de Soluções de Navegação Via GPS em Tempo Real*. São José dos Campos, SP, Brasil, 2005.

ANVAARI, M.; JANSEN, S. Evaluating architectural openness in mobile software platforms. In: *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*. New York, NY, USA: ACM, 2010. (ECSA '10), p. 85–92. ISBN 978-1-4503-0179-4. Disponível em: <http://doi.acm.org/10.1145/1842752.1842775>.

APPLE, I. iPhone OS programming guide. In: *iOS Developer Library*. Cupertino, CA, USA: Apple Inc., 2008. Disponível em: `http://developer.apple.com/library/ios/DOCUMENTATION/iPhone/Conceptual/iPhoneOSProgrammingGuide/iPhoneAppProgrammingGuide.pdf`. Acessado em: janeiro de 2011.

APPLE, I. Cocoa core competencies. In: *OS X Developer Library*. Cupertino, CA, USA: Apple Inc., 2010. Disponível em: `https://developer.apple.com/library/mac/#documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html`. Acessado em: junho de 2012.

APPLE, I. Cocoa fundamentals guide. In: *OS X Developer Library*. Cupertino, CA, USA: Apple Inc., 2010. Disponível em: `https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/CocoaFundamentals/CocoaFundamentals.pdf`. Acessado em: janeiro de 2011.

APPLE, I. Concepts in objective-c programming. In: *OS X Developer Library*. Cupertino, CA, USA: Apple Inc., 2010. Disponível em: `http://developer.apple.com/library/mac/documentation/General/Conceptual/CocoaEncyclopedia/CocoaEncyclopedia.pdf`. Acessado em: junho de 2012.

APPLE, I. iOS technology overview. In: *iOS Developer Library*. Cupertino, CA, USA: Apple Inc., 2010. Disponível em: `http://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/iPhoneOSTechOverview.pdf`. Acessado em: janeiro de 2011.

APPLE, I. Object-oriented programming with objective-c. In: *OS X Developer Library*. Cupertino, CA, USA: Apple Inc., 2010. Disponível em: `https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/OOP_ObjC/OOP_ObjC.pdf`. Acessado em: janeiro de 2011.

APPLE, I. The objective-c programming language. In: *OS X Developer Library*. Cupertino, CA, USA: Apple Inc., 2010. Disponível em: `http://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ObjectiveC/ObjC.pdf`. Acessado em: janeiro de 2011.

CLEMENTE, R. G. *Uma Solução de Streaming de Vídeo para Celulares: Conceitos, Protocolos e Aplicativo*. Dissertação (Mestrado) — UFRJ - Departamento de Eletrônica e de Computação, 2006.

COSTA, J. G.; CAMPOS, I. S. *Recomendações Básicas para a Produção de Sementes de Milho no Nível da Pequena Propriedade Rural*. Rio Branco, AC, Brasil, 1997.

DEITEL HARVEY M.; DEITEL, P. J. *Java: Como programar*. 6ª. ed. São Paulo, SP, Brasil: Prentice-Hall, 2005.

EVANS BILL; HOFFMAN, T. *Apple Makes Mac OS X the Default Operating System on All Macs*. 2002.

FAYAD M.E; SCHIMIDT, D. J. R. *Building Application Frameworks*. New York, NY, USA: Willey, 1999. 664 p.

FLEURY, A. Capacitação tecnológica e processo de trabalho. *Revista de Administração de Empresas*, p. 23–30, 1990.

GAÚCHO, F. *Java Básico e Intermediário*. Fortaleza, CE, Brasil, 2003.

GONÇALVES, J. Os impactos das novas tecnologias nas empresas prestadoras de serviços. *Revista de Administração de Empresas*, p. 63–81, 1994.

GONZALEZ-SANCHEZ, J.; CHAVEZ-ECHEAGARAY, M. E. iPhone application development. In: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. New York, NY, USA: ACM, 2010. (SPLASH '10), p. 321–322. ISBN 978-1-4503-0240-1. Disponível em: <http://doi.acm.org/10.1145/1869542.1869629>.

GRISSOM, S. iPhone application development across the curriculum. *J. Comput. Small Coll.*, Consortium for Computing Sciences in Colleges, USA, v. 24, p. 40–46, October 2008. ISSN 1937-4771. Disponível em: <http://dl.acm.org/citation.cfm?id=1409763.1409773>.

LEE, W.-M. *Beginning iOS 4 Application Development in Full Color*. Indianapolis, IN, USA: Wiley Publishing, Inc., 2010.

MANSFIELD, E. Microeconomia, teoria e aplicações. *Campus*, 1985.

MARK, D.; LAMARCHE, J. *Beginning iPhone Development Exploring the iPhone SDK*. Berkeley, CA, USA: Apress, 2009.

MATSUBARA, D. M. *Python na Prática: Um curso objetivo de programação em Python*. 2004. Disponível em: <http://www.async.com.br/projects/python/pnp/python.html>.

MUNDIM, L. R. et al. Emprego de técnicas geométricas em sistemas embarcados para auxiliar pequenos produtores rurais. *34º Congresso Nacional de Matemática Aplicada e Computacional*, 2012.

OLIVEIRA RÔMULO SILVA; CARISSIMI, A. d. S. T. S. S. Sistemas operacionais. In: *Revista de Informática Teórica e Aplicada - RITA*. Rio Grande do Sul, RS, Brasil: Universidade Federal do Rio Grande do Sul, 2001. III, n. 3.

ORACLE. *The Java Tutorial*. Redwood City, CA, USA, 2012.

PIPER, I. *Learn Xcode Tools for Mac OS X and iPhone Development*. Berkeley, CA, USA: Apress, 2009.

PISKE OTAVIO RODOLFO; SEIDEL, F. A. Rapid application development. 2006.

RODRIGUES MARIA ANDRÉIA FORMICO; BARBOSA, R. G. S. W. B. Desenvolvimento de aplicações 3D para dispositivos móveis utilizando as apis m3g e opengl es. In: *Revista de Informática Teórica e Aplicada - RITA*. Rio Grande do Sul, RS, Brasil: Universidade Federal do Rio Grande do Sul, 2006. XIII, n. 2.

ROGERS, M. It's for you!: An iPhone development primer for the busy college professor. *J. Comput. Small Coll.*, Consortium for Computing Sciences in Colleges, USA, v. 25, p. 94–101, October 2009. ISSN 1937-4771. Disponível em: <http://dl.acm.org/citation.cfm?id=1619221.1619240>.

SÁ, C. C.; ROCHA, J. *Treze Viagens pelo Mundo da Matemática*. 1ª. ed. Porto, Portugal: U.Porto, 2010.

TANENBAUM, A. S. *Sistemas Operacionais Modernos*. 2ª. ed. São Paulo, SP, Brasil: Pearson, 2003.

TEIXEIRA, L. Usabilidade e entretenimento na TV digital interativa. *GT Internet e Sociedade da Informação, do VIII Congresso Latino-americano de Pesquisadores da Comunicação*, 2006.

TIMBÓ, A. *Hoje é dia de geografia*. Curitiba, PR, Brasil: Editora Positivo, 2007.

YATES II, M. Practical investigations of digital forensics tools for mobile devices. In: *2010 Information Security Curriculum Development Conference*. New York, NY, USA: ACM, 2010. (InfoSecCD '10), p. 156–162. ISBN 978-1-4503-0202-9. Disponível em: <http://doi.acm.org/10.1145/1940941.1940972>.