3
SDK

Hoje temos diversas plataformas de dispositivos móveis disponíveis no mercado, entre elas, as mais populares são Android e iOS. Cada uma destas plataformas possui um SDK próprio com sua linguagen de programação e arquitetura bem definidas. Neste capítulo, daremos uma visão geral sobre as duas plataformas nas Seções 3.1 e 3.2. Na Seção 3.3 apresentaremos algumas alternativas de SDKs para o desenvolvimento multiplataforma. Por fim, na Seção 3.4 propomos uma extensão de API no SDK escolhido.

3.1
iOS

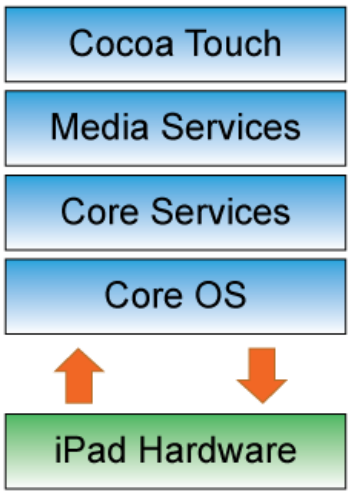


Figura 3.1: Arquitetura do iOS.

Criado pela Apple, o iOS está atualmente na versão 5.0 e possui uma arquitetura muito similar a do Mac OS X. A linguagem padrão de desenvolvimento é Objective-C, mas C/C++ pode ser usado para desenvolver algumas bibliotecas. A Figura 3.1 mostra o conjunto de camadas que compõem a plataforma.

A camada *Cocoa Touch*, acessada através de Objective-C, contém os principais serviços para construir um aplicativo iOS. Esta camada define a infraestrutura básica de um aplicativo e fornece acesso a controles de interface do usuário e tecnologias de compartilhamento de arquivos, reconhecimento de gestos, multitarefa e outros serviços.

O papel da camada *Media Services* é prover as funcionalidades de áudio, vídeo, animação e de toda a parte gráfica.

A camada *Core Services* contém os serviços do sistema da qual as camadas de cima precisam para acessar recursos e dados do sistema operacional. O acesso ao serviço desta camada é disponibilizado na sua maioria através de chamadas em C/C++.

A quarta camada é a *Core OS*, também com acesso através de chamadas em C, consiste do ambiente kernel, drivers e interfaces básicas do sistema operacional.

3.2 Android

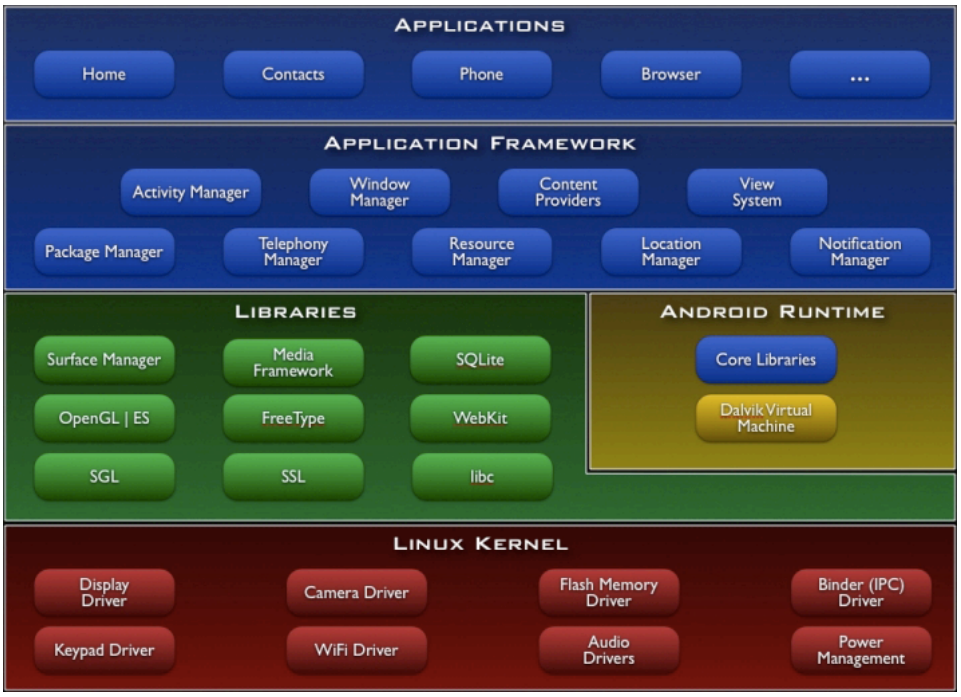


Figura 3.2: Arquitetura do Android.

O Android é um sistema operacional de código aberto construído para dispositivos móveis. Desenvolvido com base no linux, o Android usa Java como linguagem padrão de desenvolvimento, mas C/C++ pode ser usado para desenvolver aplicativos nativos usando o NDK (*Native Development Kit*).

A arquitetura é composta de cinco camadas como mostra a Figura 3.2. A camada de Aplicação contém todos os aplicativos qua são acessados pelo usuário. Estes aplicativos são escritos em Java e rodam na máquina virtual Dalvik, criada para rodar as aplicações de forma otimizada nos dispositivos móveis.

Na camada de framework estão disponíveis todas as APIs e recursos utilizados pelos aplicativos. Dentre eles, podemos citar os controles de telas, provedores de serviço (*Content Providers*), os elementos visuais como botões, caixas de texto, etc. Outra importante serviço desta camada é o gerenciador de atividades que controla o ciclo de vida das aplicações.

A camada de Bibliotecas contém todas as bibliotecas C/C++ que são utilizadas pelo sistema, entre elas a biblioteca C padrão, bibliotecas de multimídia, navegação web, acesso a banco de dados SQLite, renderização 3D (*OpenGL ES*) e outras.

Usando a versão 2.6 do kernel do linux, a camada *Linux Kernel* é responsável pelos principais serviços do sistema operacional, tais como, comunicação com o hardware e gerenciamento de memória, processos e energia.

O Android possui algumas variantes de versões. A versão mais recente é a versão 4.1 (*Jelly Bean*). Antes da versão 4.0, havia distinção entre versões para tablets e celulares com diferenças significantes no layout visual dos aplicativos de cada uma das versões.

Dada a diferença de linguagem, arquitetura e versionamento entre as duas plataformas, desenvolver um aplicativo para cada plataforma demandaria muito esforço, então optou-se por utilizar um SDK multiplataforma. Uma análise das opções existentes é apresentada na seção seguinte.

3.3 SDKs multiplataforma

Os SDKs (*Software Development Kit*) surgem como uma alternativa que permite o uso de uma linguagem em comum entre diversas plataformas. Os principais fatores que influenciaram na escolha do SDK foram:

1. As plataformas móveis que são suportadas.
2. A linguagem de programação que será usada para desenvolvimento.
3. Flexibilidade de estender e criar novas funcionalidades.
4. Comunidade de desenvolvedores para discussão.
5. Custo de licença do SDK.

Como o objetivo do framework é permitir que cada aplicação científica defina a sua própria interface, Lua se destaca como uma excelente opção de linguagem de programação não só pela sua portabilidade, mas também pela extensibilidade e eficiência. Dos SDKs que usam Lua como linguagem de desenvolvimento, foram selecionados três opções que serão abordadas a seguir.

Corona

Corona SDK (2) foi lançado em 2009 por Water Luh e Carlos Icaza e permite a criação de aplicativos para as plataformas iOS, Android, Kindle Fire e NOOK. O Corona dispõe de uma API de alto nível, fácil de usar com bibliotecas para simulação de física com suporte a gravidade e colisões, tratamento e execução de áudio, controle de mapas, integração com redes sociais, etc. Apesar de oferecer vários recursos e facilidade de uso, o Corona é uma solução fechada, ou seja, não é possível estender ou criar novas funcionalidades. O Corona possui um simulador que roda no Windows ou MacOS para testar os aplicativos antes de enviar para o dispositivo móvel. Para testar no dispositivo móvel, é necessário gerar uma *build*, onde todo o código é enviado aos servidores da CoronaLabs e um instalador do aplicativo é gerado.

Por ser mais maduro, já possui uma comunidade bastante grande que dá suporte e troca idéias com frequência no forum do site. Por razões de seguranca, as funções relativas à execução dinâmica de Lua (*dofile, load, loadfile e loadstring*) tem o acesso restringido. Essas funções são necessárias para carregar dinamicamente as definições de interface do servidor. Mais detalhes sobre esta interface são dados no Capítulo 4.

Gideros

Gideros Studio (5) foi lançado em meados de 2011 apenas com suporte ao iOS e somente em 2012 adicionou suporte ao Android. Por ser um SDK relativamente novo, não possui ainda tantos recursos quanto o Corona, mas os tem adicionando ao longo dos últimos meses. A falta de algums recursos foi compensada recentemente com a idéia de usar plug-ins. Os plug-ins são bibliotecas que podem ser desenvolvidos na linguagem nativa de cada plataforma e anexados ao SDK para serem acessados através de Lua. Outro ponto positivo do SDK é que permite execução dinâmica de código Lua. Apesar de não ter código aberto, seu uso é livre com a condição de aparecer a tela do Gideros no início do aplicativo.

Seu uso foi descartado inicialmente, pois quando estávamos fazendo a avaliação dos SDKs, o Gideros ainda não tinha suporte para a biblioteca *lua-sockets*, requisito essencial para fazer a comunicação de rede.

Moai

Criado pela Zipline Games, o Moai SDK (9) teve suas primeiras versões betas disponibilizadas em meados de 2011, destacando-se por ter o código

aberto e por atender mais plataformas que seus concorrentes. O MOAI SDK gera aplicativos para as plataformas iOS, Android, Chrome, Mac OS X, Windows e Linux. Assim como os outros SDKs, o Moai também oferece várias serviços e bibliotecas integradas (e.g. renderização, animação, simulação de física, integração com redes sociais, monetização).

Com o código fonte aberto, o Moai atrai cada vez mais desenvolvedores que contribuem para a evolução do SDK. Assim como o Gideros, o Moai permite rodar código dinâmico Lua. Outra vantagem é que com *builds* geradas para Windows ou Mac, podemos rapidamente testar e depurar código antes de enviar para os dispostivos móveis.

A Figura 3.3 mostra a arquitetura interna do SDK. O *binding* de Lua dá acesso direto a várias classes C++ implementadas no Runtime. Dessa forma, o desenvolvedor tem controle sobre operações gráficas de baixo nível, tais como carregar texturas na memória ou utilizar *framebuffers*.

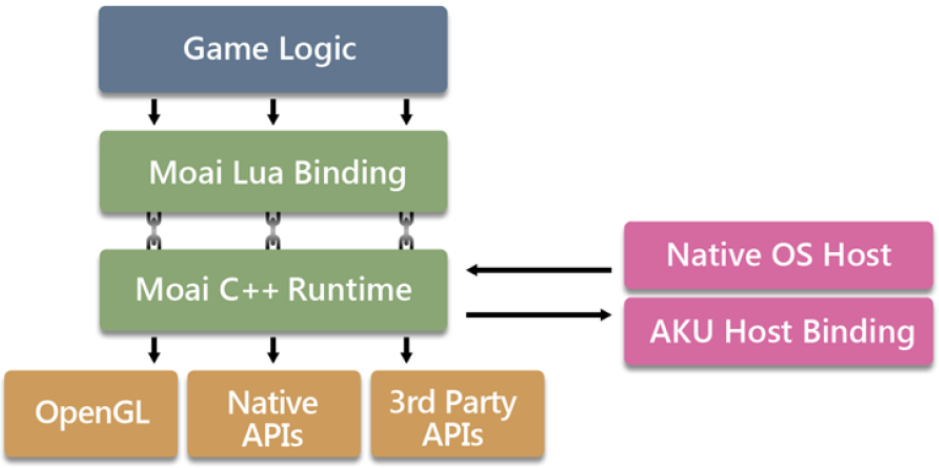


Figura 3.3: Arquitetura do Moai SDK.

Dada as vantagens e a flexibilidade de customização do SDK, resolvemos utilizar ele como plataforma para desenvolver o cliente no framework proposto. Otimizações no carregamento de texturas em OpenGL e a inclusão do descompressor LZO (10) foram as customizações feitas no SDK. Com isso foi possível interpretar, além de imagens no formato JPEG e PNG, as imagens comprimidas que são enviadas do servidor.

3.4 Proposta de Extensão da API do Moai em Lua

Diferentemente dos outros dois SDKs, o MOAI fornece APIs em Lua como abstrações de baixo nível, o que dá ao programador mais controle sobre

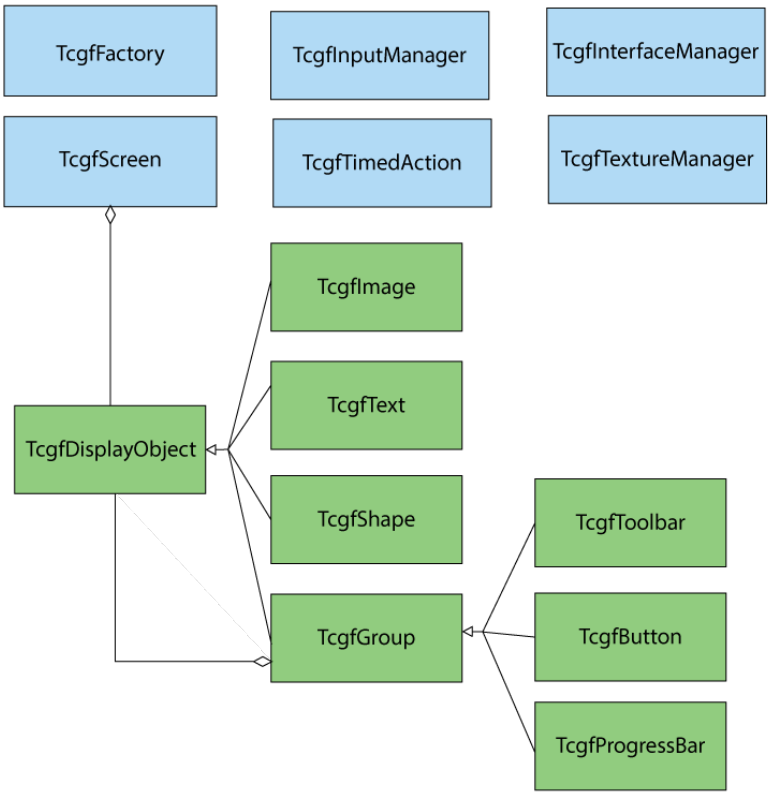


Figura 3.4: Diagrama de classes da extensão de API criada para o Moai.

o comportamento do aplicativo mas requer um esforço maior de programação. Pensando nisso, a Ymobe (15) começou o desenvolvimento de uma API de alto nível chamada Rapanui, que se assemelhava a API do Corona, facilitando a escrita de código e permitindo aos desenvolvedores do Corona portar de forma mais tranquila seus produtos para o Moai. Como o conjunto de APIs do Rapanui ainda estava nos estágios iniciais e continha muitos bugs, resolvemos criar nossa própria extensão da API do Moai em Lua.

Além de facilitar o desenvolvimento ao reduzir o volume de código escrito, a API também é responsável por controlar transições de telas, gerenciar a conexão TCP e tratar os eventos de toque na tela, detectando quais gestos são executados.

A Figura 3.4 ilustra as classes criadas nesta proposta. A classe principal é *TcgfDisplayObject* que implementa o comportamento geral de todos os objetos que são exibidos na tela. Dela são derivados os outros objetos como Imagem (*TcgfImage*), Texto (*TcgfText*) e Formas Geometricas (*TcgfShape*). A classe TcgfGroup permite o agrupamento de objetos para criação de componentes visuais mais complexos, tais como, botões (*TcgfButton*) e barras de ferramentas

(*TcgfToolbar*).

TcgfInputManager é uma classe utilitária que tem como função abstrair o processo de análise dos eventos de toque ou de mouse e interpretá-los como gesto. A Seção 3.4 fornece mais detalhes sobre o comportamento desta classe.

Detecção de Gestos

As telas sensíveis ao toque já existem desde a década de 80, mas devido à tecnologia usada no hardware não era possível usar mais de um dedo na tela, o que tornava a interação do usuário muito similar ao mouse, limitando-se na maioria das vezes em clicar em opções e arrastar objetos. À medida que foram surgindo novos métodos para rastrear a presença do dedo na tela, foi possível obter uma precisão melhor nos movimentos e também reconhecer múltiplos pontos independentes na tela. Em consequência disso, os gestos puderam ser definidos de forma mais elaborada e com um potencial para criar mais combinações.

Tanto o iOS quanto o Android já implementam um conjunto de gestos próprios, mas que não são os mesmos entre as plataformas. Como em um ambiente multi-plataforma precisa-se de uma padronização, o Moai disponibiliza somente o acesso aos dados originais do toque. Com estes dados é possível implementar qualquer tipo de gesto.

Cada evento de toque contém as seguintes informações:

- **id**: identificador único para cada toque.
- **tipo**: indica o tipo do evento (*touch-down*, *touch-move*, *touch-up*)
- **posição na tela**: Coordenada x,y da tela onde o evento ocorreu
- **tempo**: tempo do sistema em que o evento ocorreu com precisao em milisegundos

O ciclo de vida de um toque se inicia quando um evento *touch-down* chega. Seguindo uma arquitetura parecida com a proposta de Echtler e Kinkler (3), o primeiro passo é transformar as coordenadas recebidas em coordenadas do mundo em OpenGL. A partir desta nova coordenada, obtemos todos os elementos de interface que interceptam este ponto. Para isso, usamos uma implementação de divisão espacial fornecida pelo Moai que localiza os elementos a partir de uma coordenada do mundo. Cada elemento de interface tem uma propriedade de prioridade, uma espécie de *z-index* que ajuda a controlar a ordem que os elementos estão na tela. Por ordem de prioridade, listamos os elementos do mais visível ao menos visível como mostra a Figura 3.5 e verificamos qual tem um tratamento para o evento. Se algum objeto

responder que está tratando o evento, o *id* do toque fica associado àquele objeto e todos os eventos deste mesmo *id* serão redirecionados para este objeto, até chegar o evento de *touch-up* desassociando o *id* ao objeto.

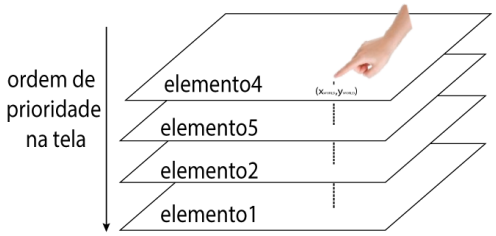


Figura 3.5: Rastreando os objetos por ordem de prioridade.

Uma vez identificado o objeto que tratará o evento, seguimos para a camada de interpretação. Esta camada é responsável por entender uma sequência de eventos de toque e reconhecer qual gesto está sendo executado. Os gestos podem ser reconhecidos durante o movimento ou após ele ter finalizado. Uma análise do comportamento espacial e/ou temporal determina qual gesto foi executado.

O grande desafio nesta etapa é lidar com possíveis conflitos entre os gestos. Para isso é necessário adicionar restrições de limite máximo ou mínimo para o tempo do gesto ou movimento sendo executado. Adicionalmente é monitorado somente os gestos indicados para cada elemento de interface, reduzindo a chance de conflitos.

Até a presente data, os seguintes gestos foram implementados:

Gesto		Descrição
 1	**Tap**	O dedo toca rapidamente na tela porém não executa nenhum deslocamento. *tap(objectName, x, y)*
 1	**Double Tap**	Dois toques consecutivos na mesma posição da tela. Seguindo a mesma regra do Tap, mas com a restrição do segundo toque ser na mesma região que o primeiro. *doubleTap(objectName, x, y)*
 1	**Hold**	Manter o dedo pressionado na mesma posição por um certo tempo. *hold(objectName, x, y)*

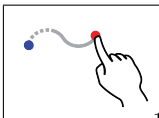
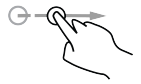
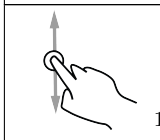
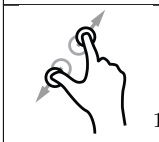
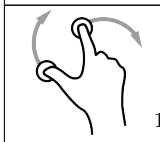
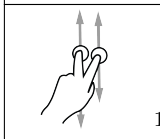
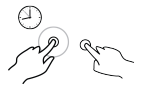
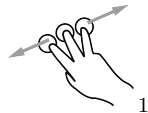
 1	**Free move**	Sem restrições, todos os eventos (Hold/Moving/Finished) são repassados para o servidor. *freeMove(objectName, type, x, y)*
 1	**Flick**	Movimento rápido de um dedo em uma direção (Left/Right/Up/Down). A velocidade do gesto é calculada dividindo a distância percorrida pelo tempo total do gesto. *flick(objectName, speed)*
 1	**Slide**	Movimento contínuo em um eixo (Horizontal/-Vertical). *slide(objectName, axis, displacement)*
 1	**PinchZoom**	Gesto de aproximar ou distanciar dois dedos. *pinchZoom(objectName, zoom)*
 1	**Rotate**	Gesto de rotação com dois dedos. *rotate(objectName, angle)*
 1	**Slide 2Fingers**	Similar ao gesto Slide, mas com dois dedos. *slide2Fingers(objectName, axis, displacement)*
 1	**Flick 2Fingers**	Similar ao gesto Flick, mas com dois dedos. *flick2Fingers(objectName, speed)*
 1	**Hold and Tap**	Segurando um dedo na tela, outro toca rapidamente na tela. *holdAndTap(objectName, holdX, holdY, tapX, tapY)*
 1	**Hold and DoubleTap**	Segurando um dedo na tela, outro toca duas vezes na tela. *holdAndDoubleTap(objectName, holdX, holdY, tapX, tapY)*
 1	**Free move - 3Fingers**	Movimento livre usando três dedos. *freeMove3Fingers(objectName, type, x, y)*

Tabela 3.1: Lista de gestos implementados.

[1]Ícones providos pela Gestureworks (`http://gestureworks.com`), licença cc-by-sa