**LEYDINALDO FERREIRA MIRANDA**

# ULTILIZAÇÃO DE UM MOTOR GRÁFICO PARA O DESENVOLVIMENTO DE UM SISTEMA COMERCIAL MÓVEL DESTINADO AO RAMO DE ALIMENTAÇÃO

**PALMAS / TO**

**2016**

**LEYDINALDO FERREIRA MIRANDA**

# ULTILIZAÇÃO DE UM MOTOR GRÁFICO PARA O DESENVOLVIMENTO DE UM SISTEMA COMERCIAL MÓVEL DESTINADO AO RAMO DE ALIMENTAÇÃO

Trabalho de Conclusão de Curso de Sistemas de Informação da Fundação Universidade do Tocantins, apresentado como parte dos requisitos para obtenção do título de Bacharel em Sistemas de Informação.

Orientador: Msc. Silvano Malfatti

**PALMAS / TO**

**2016**

## DEDICATÓRIA

Dedico este trabalho a todos os professores
que se dedica a ensinar da melhor
forma seus alunos, de forma a contribuir
para uma sociedade melhor.

## AGRADECIMENTOS

Agradeço a Deus por me acompanhar e ajudar a superar os obstáculos.

Agradeço a minha família: aos meus pais Leonora Bandeira Miranda Silva e Durval Ferreira da Silva, que me incentivaram desde cedo aos estudos. Aos meus irmãos Leydiane, Leydivaldo, Leydiluz e Leydmara, pelo amor, amizade e por todo o incentivo.

Agradeço os meus amigos de forma especial Andréia Gualberto Pereira e Paulo Henrique Guimarães por todo o companheirismo e contribuição para este trabalho.

Agradeço de forma muito especial minha companheira Adriana Alves de Oliveira por todo o carinho, paciência e toda a contribuição na realização da pesquisa.

Agradeço também, ao meu orientador Silvano Maneck Malfatti pela ajuda e confiança. E aos demais professores que de alguma forma contribuíram para o meu desenvolvimento profissional.

E por fim, e não menos importante agradeço a Fundação Universidade do Tocantins, por dá a oportunidade de concluirmos uma graduação gratuita e proporcionar um ensino de qualidade.

Que vossos esforços desafiem as
impossibilidades, lembrai-vos de que as grandes coisas
do homem foram conquistadas do que parecia impossível

(Charles Chaplin)

## RESUMO

Os dispositivos móveis evoluíram tanto no hardware quando na interface, assim surgiu à necessidade de aplicações aprimorada que atendam de maneira satisfatória as necessidades dos seus usuários. Atualmente a plataforma Android representa boa parte dos sistemas operacionais dos dispositivos, sendo responsável por disponibilizar aos desenvolvedores componentes visuais tradicionais e outros recursos, como OpenGl que permite criar interfaces gráficas personalizadas que pode atender a demanda dos usuários. Porém, desenvolver aplicações com OpenGl demanda muito tempo, para isso faz-se o uso de motores gráficos, que disponibiliza um conjunto de bibliotecas facilitando no desenvolvimento. Assim, o presente trabalho tem por objetivo avaliar o impacto e o esforço necessário para o desenvolvimento de sistemas comerciais móveis, utilizando um motor gráfico ao invés das bibliotecas de componentes convencionais.



**Palavras-chaves:** Android; Motores gráficos; Componentes visuais tradicionais.

## ABSTRACT

Mobile devices have evolved on both hardware and interface. Therefore, it pushes for the necessity of apps that would suitably meet the demands of their users. Currently, the Android platform represents a huge part of the operating systems of the mobile devices, giving to the developer's traditional visual components and other features, such as OpenGl, which allow users to create customized graphical interfaces that respond to their demands. Developing applications such as OpenGl, however, takes a lot of time. Then, using graphic engines provides a set of libraries and makes development easier. The goal of this research is to evaluate the impact and the effort to develop commercial mobile systems using graphic engine instead of conventional component libraries. Mobile devices have evolved on both hardware and interface. Therefore, it pushes for the necessity of apps that would suitably meet the demands of their users. Currently, the Android platform represents a huge part of the operating systems of the mobile devices, giving to the developer's traditional visual components and other features, such as OpenGl, which allow users to create customized graphical interfaces that respond to their demands. Developing applications such as OpenGl, however, takes a lot of time. Then, using graphic engines provides a set of libraries and makes development easier. The goal of this research is to evaluate the impact and the effort to develop commercial mobile systems using graphic engine instead of conventional component libraries.



**Keywords**: Android; Graphics engines; Traditional visual components.

## Lista de Abreviaturas

***ART -*** *Android Runtime*

***API*** *- Application Programming Interface*

***IDE -*** *Integrated Development Environment*

***iOS*** *- iPhone Operating System*

***HAL –*** *Hardware Abstraction Layer*

***MVC*** *- Model View Controller*

***MVP -*** *Model View Presenter*

***SDK -*** *Software Development Kit*

***URI –*** *Unified Resource Identification*

***XML*** *– eXtensible Markup Language*

## Lista de Figuras

## Lista de Quadros

## Lista de Tabelas

**SUMÁRIO**

# 1. INTRODUÇÃO

As interfaces têm evoluído nas últimas décadas. É possível perceber e constatar que pessoas tiveram contato com as primeiras interfaces, que eram por meio de prompt de comandos, e naquela época nem se imaginaria uma tecnologia avançada, baseada em toques (padrão *mobiles*). Entre aquelas, duas realidades citadas, ocorreram evoluções bastantes significativas; sendo, o wimp (*window, icon, menu, pointer*), as interfaces baseadas em gestos, realidade virtual, e os comandos de voz (Garbin, 2010).

Houve também uma evolução nos dispositivos móveis que aumentou em capacidade de: armazenamento, processamento, comunicação e, interatividade. Com isso, evoluiu as interfaces e a quantidade de consumidores.

Os dispositivos móveis, com aquelas características avançadas, são chamados de *smartphones*. O uso dessa tecnologia tem sido utilizada cada vez mais entre pessoas e empresas. Uma vez que permite maior flexibilidade, mobilidade e, principalmente, facilidade para se comunicar com outras pessoas. Dessa forma, os usuários *mobile* têm disponibilidade para acessar diversos sistemas em seu cotidiano, como por exemplo: sistema de vendas, jogos, rede sociais e outros (Lecheta, 2013, p.23). Com isso surge a necessidade de aplicações aprimoradas que consigam atender as demandas de seus usuários.

Os sistemas, em sua maioria, são desenvolvidos com recursos tradicionais dos sistemas operacionais como, por exemplo, o uso de: botões, painéis, textos: entre outros, que já estão praticamente prontos. Esse tipo de desenvolvimento, por sua vez, pode não oferecer meios de interação intuitivos, baseados em metáforas que remetem o usuário a situações de seu cotidiano.

A computação gráfica admite que os desenvolvedores projetem interfaces diferenciadas que explorem, através de animações gráficas, os sentidos dos usuários, as quais permitem uma maior comunicação com os dispositivos.

Por outro lado desenvolver um sistema utilizando a computação gráfica demanda tempo e trabalho, pois é necessário gerenciar um conjunto de subsistemas como a renderização, detecção de colisão, tratamento de eventos, dentre outros. Assim faz-se necessário o uso de motores gráficos que gerencie esses subsistemas.

O presente trabalho tem por objetivo geral avaliar o impacto e o esforço necessário para o desenvolvimento de sistemas comerciais móveis utilizando um motor gráfico ao invés

da biblioteca de componentes convencional oferecida pela tecnologia Android. Como objetivos específicos o projeto busca: diferenciar a organização e estrutura do modelo de desenvolvimento tradicional e do modelo com motores gráficos, avaliar: o impacto na modelagem de telas, o esforço para fazer o tratamento de eventos e troca de contexto, o impacto para ligar o projeto a um modelo de negócios e o grau de satisfação do usuário quando interage com interfaces personalizadas, fazendo uma comparação com um aplicativo que utiliza componentes tradicionais.

Para avaliar o impacto causado pela troca de tecnologias, foi desenvolvido como estudo de caso AnimePizza, software destinado a pizzarias que utilizará recursos da plataforma Android junto com a computação gráfica.

O aplicativo AnimePizza tem um *layout* personalizado, contendo animações gráficas, onde o cliente tem a oportunidade de montar sua pizza escolhendo sabor, tamanho, massa e borda, além de bebidas e extras.

A organização desse trabalho está dividida da seguinte forma. No capítulo 2 foi feito a revisão de literatura, isto é, apresentado os conceitos que serão desenvolvidos ao longo do trabalho. No capítulo 3 é apresentada a metodologia do trabalho. No capítulo 4 e 5, e certamente a aparte mais importante desse trabalho, é exibida a análise dos resultados da pesquisa, parte esta que é descrita como cada parte do trabalho foi desenvolvido. No capítulo 6 é apresentada a conclusão deste trabalho.

# 2 REFERENCIAL TEÓRICO

Esta seção apresenta o embasamento teórico necessário para a melhor compreensão da proposta deste trabalho.

## 2.1 ANDROID

O Android é uma plataforma de desenvolvimento baseada em conceitos do Sistema Operacional Linux, o que permite que suas aplicações sejam livres e de código aberto (*open source*). Esse fator é de fundamental importância muito atrativa, pois permite a evolução rápida de seus componentes, devido ao fato de que programadores em qualquer lugar do mundo possam contribuir para melhorar a plataforma (LECHETA, 2013, p.26).

Considerado uma curiosidade por alguns no começo, o Android cresceu em poucos anos para se tornar um ator capaz de mudar o mercado, ganhando tanto respeito como escárnio dos seus colegas da indústria (ABLESON, W.F. et al., 2012, p.3)

Segundo recente estudo realizado pelo O Estado da Nação de Desenvolvedores, referente à primeira metade deste ano, 47% dos desenvolvedores profissionais de aplicativos consideram o Android a plataforma prioritária. Na pesquisa anterior havia empate técnico, com 40% mencionando o Android, e 39%, o *iPhone Operating System* (iOS) da Apple. Agora, o iOS tem 31% da preferência. Esse estudo também a presenta dois motivos para essa mudança. O primeiro motivo, é devido ao fato do Android ser a maior plataforma móvel usada por mais de 80% das pessoas mundo afora. O outro motivo é porque os desenvolvedores estão reduzindo o foco em *apps* que gerem receita por aquisição, assinatura, publicidade, e focando mais em serviços que beneficiem outros negócios (BUCCO, 2016).

De acordo com Ableson, et al.(2012), uma das características da plataforma Android é que não existe a diferença entre aplicativos integrados e os aplicativos que você cria com o *Software Development Kit* (SDK). Isso significa que você pode escrever poderosos aplicativos que acessam os recursos disponíveis no dispositivo.

As seções subsequentes têm por objetivo apresentar como é a Arquitetura da plataforma Android, como ela está dividida e o que tem em cada camada; e quais são os componentes Android.

### 2.1.1 Arquitetura do Android

A arquitetura Android é formada por camadas divididas por componentes, conforme exposto na Figura 1. A camada mais baixa 1, é composta pelo *Kernel (Linux Kernel*), onde existem diversos sistemas importantes, dentre eles *drives*, processos,

memórias, segurança e outros serviços. Segundo Android Developers (2016), usar um *kernel* do Linux permite que o Android aproveite os recursos de segurança principais e que os fabricantes dos dispositivos desenvolvam *drivers* de hardware para um *kernel* conhecido.

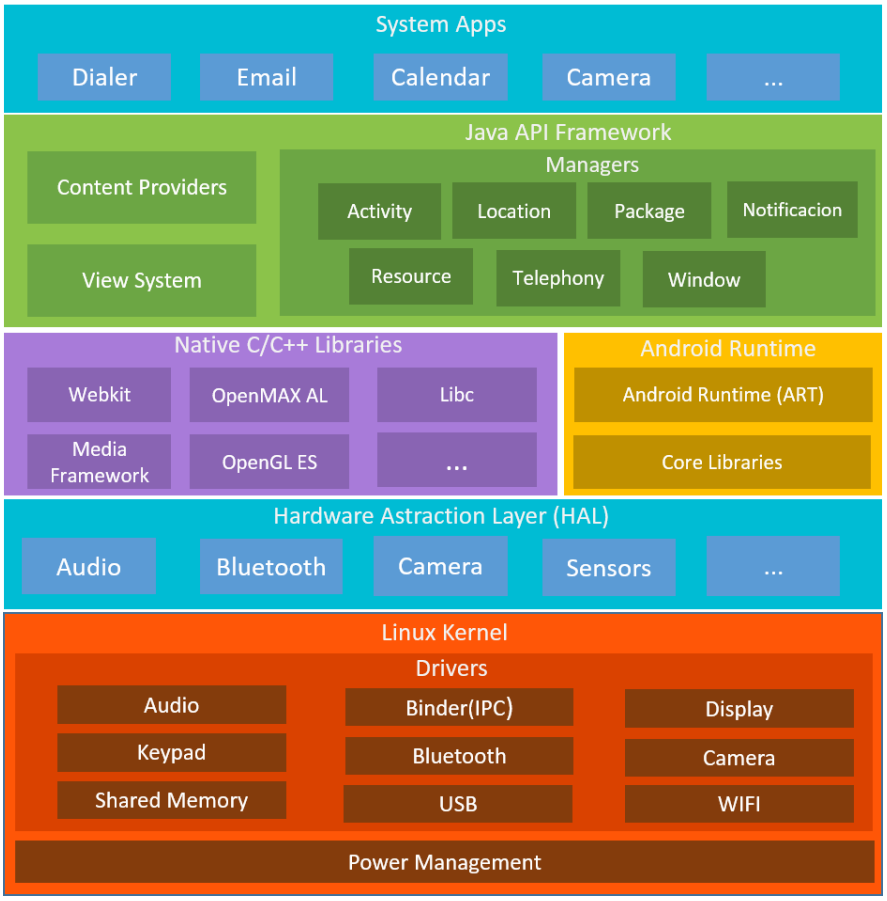


Figura 1- A pilha de software do Android (Adaptado do ANDROID DEVELOPER, 2016).

Na Figura1, acima do *Kernel*, existe a *Hardware Abstraction Layer* (HAL) que camada de abstração de hardware, esta por sua vez fornece interface padrão que expõem as capacidades de hardware do dispositivo para a estrutura da Java API de maior nível. A HAL consiste em módulos de biblioteca, que implementam uma interface para um tipo específico de componente de hardware, como o módulo de câmera ou bluetooth (ANDROID DEVELOPER, 2016).

Na camada superiora camada de abstração do hardware estão localizados o *Runtime Android* (ART) e as Bibliotecas C/C++ nativas. ART são bibliotecas que quem as funcionalidades da linguagem de programação Java, e que foram projetados para executar várias máquinas virtuais em dispositivos de baixa memória executando arquivos *Dalvik Executable* (DEX), um formato de *bytecode* projetado especialmente para Android,

otimizado para oferecer consumo mínimo de memória. As Bibliotecas C/C++ nativas permitem manipular banco de dados, renderizar imagens e fontes, manipular objetos 2D e 3D. Essas são de fundamentais importância, pois existem componentes e serviços do sistema Android, tais como ART e HAL, que são implementados em código nativo que exige bibliotecas nativa programadas em C e C ++ (ANDROID DEVELOPER, 2016).

Logo acima da Bibliotecas C/C++ e o ART, está a camada que contém a estrutura da API Java (Java API Framework). Segundo ANDROID DEVELOPER (2016) essas APIs formam os blocos de programação que você precisa para criar os aplicativos Android simplificando a reutilização de componentes e serviços de sistema modulares e principais.

Ainda na Figura 1, mais precisamente no topo da pilha estão localizados os aplicativos do sistema (*System Apps*), é a interface de usuário, local onde estão as funções básicas de telefone. De acordo com ANDROID DEVELOPER (2016) os aplicativos do sistema funcionam como aplicativos para os usuários e fornecem capacidade principais que os desenvolvedores podem acessar pelos próprios aplicativos.

Esta seção ajudará a entender o funcionamento de uma aplicação Android e os componentes convencionais.

2.1.2 Componentes de uma aplicação Android

Quando se desenvolve aplicações para plataforma Android é necessário estruturar alguns componentes essenciais para que a aplicação seja executada. A Figura 2 mostra esses componentes, sendo eles: *Activities, Services, AndroidManifest.xml, ContentProviders* e *BrodcastReceivers.*


Figura 2 – Componentes de uma aplicação Android (CÁRDENAS, 2016).

Como mostra a Figura 2 a união desses componentes formam o Android *Core* que segundo Cárdenas (2016), não é um componente especifico, mas sim a plataforma Android propriamente dita. É através dele que ocorre a interação entre os componentes, o que torna possível a execução do código.

Podemos descrever os componentes da seguinte forma:

> *Activities* são as representantes das telas da aplicação. Associada a uma activity normalmente existe uma *view*, que define como será feita a exibição visual para o usuário. As *activities* são responsáveis por gerenciar os eventos de tela e também coordenam o fluxo da aplicação.
>
> Services, são códigos que executam em segundo plano. Normalmente são utilizados para tarefas que demandam um grande tempo de execução.
>
> *ContentProviders* (provedores de conteúdos), são a maneira utilizada pela plataforma para compartilhar dados entre as aplicações que executam no dispositivo. Um exemplo bem claro disto é a aplicação de gerenciamento de contatos do Android, que é nativa. Aplicações desenvolvidas por terceiros podem utilizar um contente *provider* a fim de ler os contatos armazenados no dispositivo de forma simples.
>
> *BroadcastReceivers* são componentes que ficam "escutando" a ocorrência de determinados eventos, que podem ser nativos ou disparados por aplicações. Uma aplicação pode, por exemplo, utilizar um broadcast retiver para ser avisada quando o dispositivo estiver recebendo uma ligação e, com base nessa informação, realizar algum tipo de processamento. (Cárdenas,2016, p.4)

Unido a esses componentes tem o *AndroidManifest* que é obrigatório e único para a aplicação, nele está contido as configurações gerais da aplicação e dos componentes que fazem parte dela.

#### 2.1.2.1 *Activities*

Uma *Activity* é uma classe que geralmente representa uma tela da aplicação, sendo responsável por gerenciar os eventos gerados na tela, como, por exemplo, a ação de quando um botão é pressionado ou quando um item do menu é escolhido. Uma vez que a plataforma já possui métodos predefinidos para o tratamento desses eventos.

Obrigatoriamente a activity deve implementar o método *onCreate(bundle*), o qual é o responsável por inicializar a aplicação chamando o método *setContentView(view)* que recebe como parâmetro um objeto do tipo *android.app.view* que representa os elementos gráficos da tela (Lecheta, 2013).

Comumente usada nas interfaces tradicionais, a classe *android.app.view*é a classe-mãe de todos os objetos visuais do Android, que podemos dividi-los em dois tipos.*View*que herda diretamente da classe-mãe *View* e os gerenciadores de *layout* que são subclasses de *android.view.viewgroup*os quais podem conter *views*-filhas ou até mesmo outras *viewgroup* (LECHETA, 2013).

Segundo Ribeiro (2013), podemos definir um objeto *View* como uma estrutura de dados que representa uma área retangular limitada da tela do dispositivo, onde é possível obter informações como medidas de largura e altura, desenho, mudança de foco, capacidade de rolagem e captura de comandos percebidos pela área específica. Sendo que a plataforma Android disponibiliza ao desenvolvedor *widgets* já pré-definidoscomo*TextView,EditText, AutoCompleteTextView, Button,ImageButton, CheckBox, ToggleButton, RadioButtom, Spinner, ProgressDialog, ProgressBar, Toast*e outros.

Já os objetos do tipo *android.view.Viewgroup* funcionam como *containers* e serve como base para subclasses chamadas *layouts*, que geralmente são utilizadas em estruturas mais complexas. As principais*ViewGroups* predefinidas e disponibilizadas pela plataforma são:*AbsolutLayout, FrameLayoutLinearLayout, RelativeLayoute e TableLayout* (LECHETA 2013).

Na Figura 3 podemos visualizar a árvore hierárquica que pode ser simples ou complexa, dependendo da necessidade da aplicação.

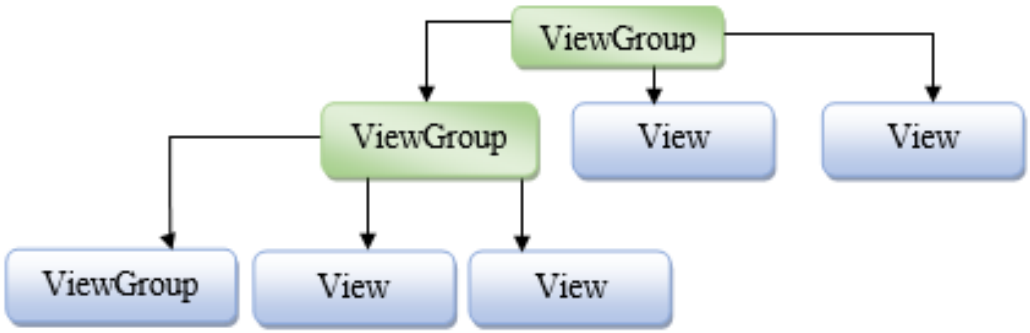


Figura 3 – Hierarquia de Views e ViewGroups (Adaptado de RIBEIRO, 2013)

O método *setContentView()* é o responsável por fazer a ligação entre a *activity* e a *view* recebendo como parâmetro o nó raiz da arvore, com a finalidade que ela seja anexada a tela. Assim o sistema operacional faz automaticamente a renderização desses componentes visuais não havendo necessidade que o desenvolvedor faça o tratamento.

Para que seja possível compreender melhor o conceito de *actitvity* e sua estrutura no capítulo a seguir será mostrado o ciclo de vida de uma *activity*.

#### 2.1.2.1.1 Ciclo de vida

O ciclo de vida de uma *activity* é bem definido. Sempre que uma *activity* é iniciada ela é inserida no topo de uma pilha chamada *ActivityStack* logo, a anterior passara a ficar abaixo dela, dessa forma o sistema precisa gerenciar os estados das *activity* os quais são, em execução, temporariamente interrompido em segundo plano e completamente destruído.

Existem três subníveis do ciclo de vida de uma *activity*, que por vez ficam se repetindo durante a execução da aplicação. Cada um desses ciclos inicia-se durante a chamada de um dos métodos e termina quando outro método é chamado.

São eles:

> *Entire lifetime*: Este é o ciclo completo, ocorre uma única vez, definindo o tempo de vida da *activity.* Acontece entre as chamadas do método *onCreate( )*e *onDestroy( ).*
>
> *Visible lifetime: Activity* pode estar no topo da pilha, porém interagindo com o usuário em segundo plano ou temporariamente parada em segundo plano. Acontece entre a execução dos métodos *onStart()* e *onStop().*
>
> *Foreground lifetime: Activity* está no topo da pilha interagindo com o usuário. Acontece entre a chamada dos métodos *onResume()* e *onPause()*(LECHETA, 2013, p.117).

Na Figura 4 pode-se ver o ciclo completo e os métodos que podem ser usados para controlar os estados da aplicação.

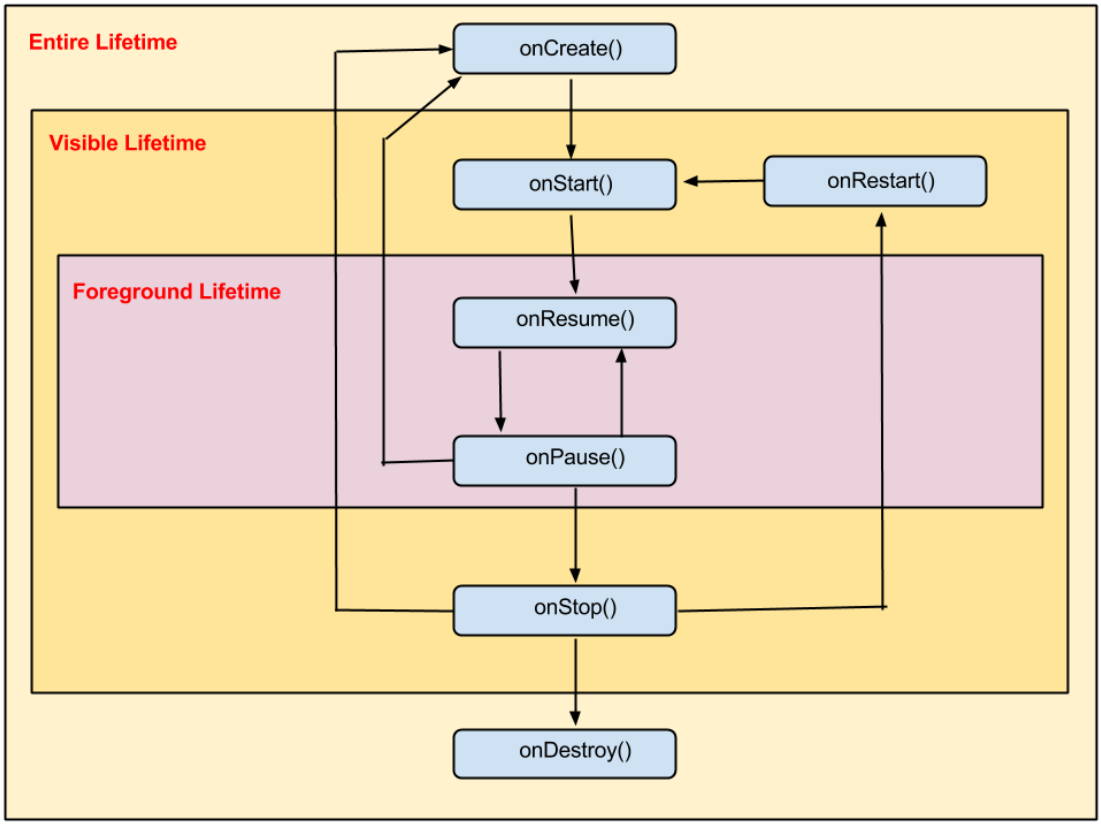


Figura 4-Ciclo de vida de uma activity (THIENGO, 2013).

(LECHETA, 2013, p.119) descreve os métodos da seguinte forma:

a) *onCreate* (): é obrigatório e é chamado uma única vez. Nesse método deve-se criar uma *view* e chamar o *método setContentView(view)* para exibi-la na tela.

b) *onStart*(): é chamado quando a *activity* está ficando visível ao usuário e já tem uma *view*. Pode ser chamado depois dos métodos todos *onCreate*() ou *onRestart*(), dependendo do estado da aplicação.

c) *onRestart*(): é chamado quando uma *activity* foi parada temporariamente e está sendo iniciada outra vez. O método *onRestart*() chama o método *onStart*() de forma automática.

d) *onResume*(): é chamado quando a *activity* está no topo da pilha "*activitystack*", e dessa forma já está executando como a activity principal pronta para interagir com o usuário.

e) *onPause*() : se um evento ocorrer, como o celular entrar em modo espera para economizar energia, a *activity* do topo pilha que está executando pode ser temporariamente interrompida. Para isso o *onPause*() é chamado para salvar o estado da aplicação, para que posteriormente, quando a *activity* voltar a executar, tudo possa ser recuperado, se necessário, no método *onResume*().

f) *onStop*() : é chamado quando a *activity* está sendo encerrada, e não está mais visível ao usuário, o que pode ocorrer quando outra *activity* é iniciada.

g) *onDestroy*() : literalmente encerra a execução de uma *activity*. O método pode ser chamado automaticamente pelo sistema operacional para liberar recursos ou pode ser chamado pela aplicação pelo método *finish*() da

> classe *Activity*. Depois do método *onDestroy*() ter executado a activity é removida completamente da pilha e o seu processo no sistema operacional também é completamente encerrado.

Manipular uma Activity é um processo simples e importante para o desenvolvimento de aplicações Android, pois quando os métodos citados anteriormente são aplicados de forma correta permite que a aplicação fique bem mais robusta.

#### 2.1.2.2 *Services*

A classe *Service* é utilizada para executar um processamento em segundo plano por tempo indeterminado, chamado popularmente de serviço. Um Serviço geralmente faz um alto consumo de recursos, memória e CPU. Não precisa interagir com o usuário e consequentemente não precisa de interface gráfica (LECHETA, 2013, p. 348).

Segundo Lecheta (2013), há duas formas de se iniciar um serviço em segundo plano, por meio dos métodos *startService(),*e *bindService()*. O método *startService()* é utilizado para iniciar um serviço que fica executando por tempo indeterminado, até quando o método *stopService()* é acionado ou até que o próprio serviço termine a sua própria execução chamando e método *stopSelf()*. O segundo método é o *bindService(),* o mesmo pode iniciar um serviço se ele ainda não está executando ou simplesmente conectar-se a ele. Vale ressaltar que a realizar a conexão é possível recuperar uma referência de uma classe ou interface que pode manipular o serviço.

#### 2.1.2.3 *ContentProvider*

Como já foi citado anteriormente, o *ContentProvider* é recurso do Android que permite o compartilhamento de dados, permitindo dessa forma que informações possam se tornar pública para as aplicações.

De acordo com Reis (2014), os provedores de conteúdo é uma abstração das fontes de dados das camadas inferiores, a exemplo, a camada de banco de dados. Esse recurso possibilita disponibilizar mecanismos de segurança de dados, como controle de permissão de leitura e escrita de dados, e disponibiliza uma interface padrão para compartilhar dados entre aplicações.

A classe *ContentProvider* necessita do auxílio de uma interface para seu funcionamento. Geralmente é usada a interface *ContentResolver*, conforme apresentado na Figura 5.

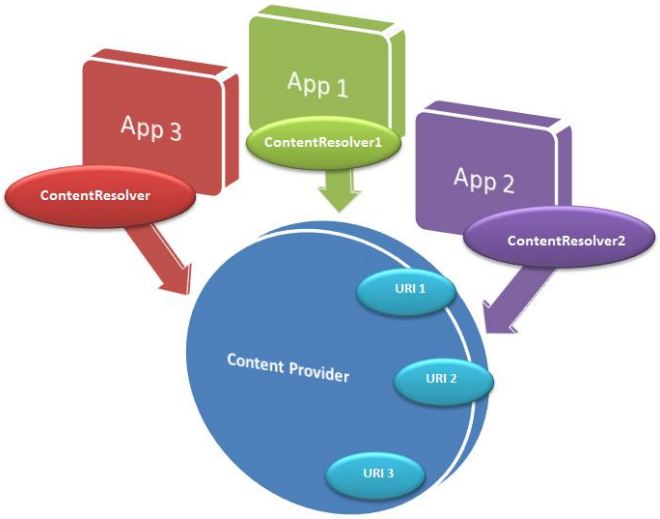


Figura 5 - *ContentProvider* (REIS, 2014).

Segundo Reis (2014), a função da interface *ContentResolver* é receber as requisições dos clientes, e "resolvê-las", ou seja, envia a requisição para um *ContentProvider* distinto. Para realizar isso, a interface faz uso dos seus métodos CRUD (*create, read, update, delete*) correspondentes aos métodos abstratos do *ContentProvider* (*insert, delete, query, update*). Cada método recebe um *UnifiedResourceIdentification* (URI) especificando o endereço do *ContentProvider* no qual se deve interagir.

A classe URI da plataforma Android é utilizada em conjunto com um provedor de conteúdo com a finalidade de identificar um registro ou informação com *string* única e um formato amigável (Lecheta, 2013).

#### 2.1.2.4 *BroadcastReceivers*

Segundo Lecheta(2013), a classe *BroadcastReceiver* é utilizada para responder determinados eventos enviados por uma *intent*. A *intent* representa uma "ação" que a aplicação deseja executar. Na prática essa intenção é enviada ao sistema operacional como uma mensagem, chamada de broadcast. Ao receber a mensagem, o sistema operacional tomará as decisões necessárias, dependendo do conteúdo da mensagem, isto é, de sua intenção.

Ainda de acordo com Lecheta (2013) um *BroadcastReceiver* não utiliza interface gráfica e não se comunica diretamente com o usuário, pelo contrário, ele é executado em segundo plano sem que o usuário perceba, isso é muito importante principalmente quando

se trata de integração de aplicações, uma vez que elas podem trocar mensagens em segundo plano sem atrapalhar o usuário.

Na aplicação tradicional uma *intent* geralmente é utilizada para fazer a troca de tela, chamando outra activity, utilizando o conceito de pilha, assim a activity que estava sendo utilizada passa para segundo plano e a nova activity será apresentada na tela.

## 2.2 MODELO DE PADRÃO *MODEL VIEW CONTROLLER* (MVC) E *MODEL VIEW PRESENTER* (MVP)

MVC é um padrão de apresentação de interface do usuário que se concentra em separar a UI(*View*) de sua camada de negócios(*Model*). Nesse padrão, o sistema é dividido em três camadas, sendo elas: o *Model*, que é responsável pelo comportamento dos negócios e gerenciamento dos estados; a *View* responsável pela interface do utilizador; e o *Controller* é quem responde pelo processamento dos dados transmitidos entre a *View* e o *Model* (GUALHARDO, 2016).

Além disso, Gualhardo (2016) ainda afirma que na maioria das implementações todos os três podem interagir diretamente uns com os outros e em algumas implementações o controlador é responsável por determinar qual visualização vai ser exibida. Na Figura 6 é possível visualizar essa interação.

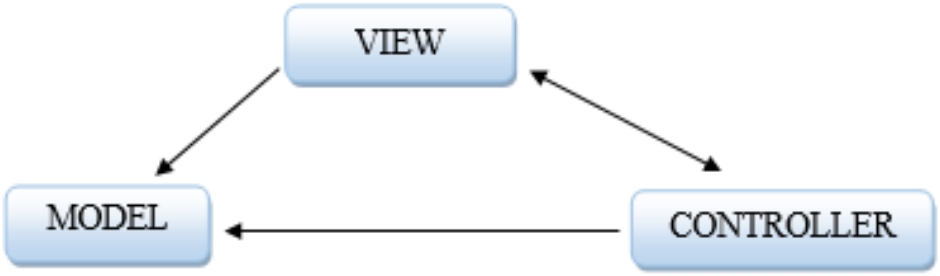


Figura 6 - Model View Controller (Elaborada pelo autor)

O padrão MVP é considerado por muitos uma evolução dos conceitos de MVC. Ele tem a finalidade de separa a camada de apresentação das camadas de dados e as regras de negócios. Esse padrão é dividido em três partes que possuem responsabilidade específicas, são elas: o *Model*, que é responsável pelo comportamento dos negócios e gerenciamento dos estados; a *View* que é responsável pela interface do utilizador; e o Presenter que é a ligação entre *View* e *Model*, e possui um papel de mediador entre eles. (GUALHARDO, 2016). A Figura 7, apresenta a estrutura de um MVP.

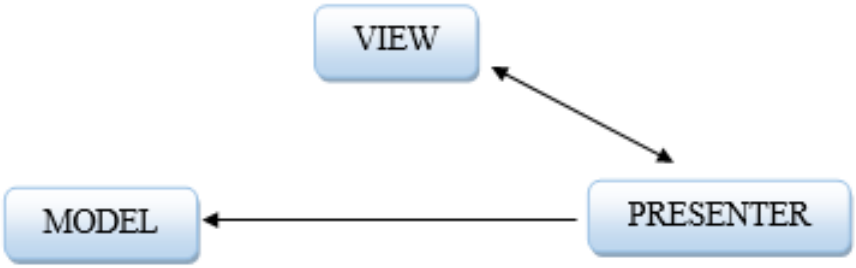


Figura 7– Model View Presenter (Elaborada pelo autor)

Como foi visto na anteriormente uma *activity* geralmente representa uma tela da aplicação sendo que através do método *setContentView* informa qual layout vai desenhar os elementos na tela. Assim no modelo MVC ou MVP, a *activity* seria o *Controller* ou *Presenter*, a *view* como o próprio nome sugere é são elementos que herdam da classe mãe *view* e os *Models* classes modeladas de acorda a necessidade da aplicação.

Visto os conceitos apresentado na  seção 2.1 é possível ver que a plataforma Android é bem organizada, pois tem um padrão de desenvolvimento bem definido, o qual diminui o trabalho do desenvolvedor pois já vem com componentes visuais e métodos pré-pronto, sendo também que o sistema operacional faz o controle da troca de contexto e renderização.

No capítulo a seguir será apresentado conceitos para a construção de um layout personalizado.

## 2.3 MOTOR GRÁFICO

Um motor gráfico tem como objetivo facilitar no desenvolvimento de aplicações com interfaces mais aprimoradas que não pode ser feita pelos componentes convencionais das linguagens. Ele pode ser definido como um conjunto de bibliotecas que permitem a abstração da programação em baixo nível de maneira a oferecer uma interface de desenvolvimento mais amigável ao programador (LEWIS, 2002 citado por MALFATTI ,2009).

Outro fator importante a considerar sobre motores gráficos ou *game engine* é que surgiram com a necessidade reduzir tempo e custo no desenvolvimento de jogos, visto que a cada novo projeto desenvolvido não havia reaproveitamento de recursos utilizados em jogos anteriores, ou seja, recriavam-se tudo a parir do zero (LEWIS, 2002 citado por MALFATTI, 2009).

Segundo Maia (2005), para facilitar a compreensão de um motor gráfico podemos dividi-lo em três partes, sendo elas:

a) Núcleo, ou módulo principal, que é responsável pela gerência do motor em alto nível e contém módulos fundamentais utilizados tanto pelos subsistemas quanto pela camada de aplicação;
b) Subsistemas, que abrange cada subsistema provendo as principais funcionalidades do motor gráfico; e
c) Carregamento de Elementos do Mundo Virtual, módulo responsável pela mediação necessária ao carregamento das mídias utilizadas pelo motor gráfico para especificar o conteúdo do mundo virtual, como, por exemplo, arquivos de som e modelos tridimensionais pré-moldados.

A seguir será apresentado as principais características de cada parte do motor gráfico.

### 2.3.1Núcleo

Segundo Maia (2005), o núcleo do motor gráfico tem a função de intermediar interações entre a camada de aplicação e os subsistemas disponíveis no motor, e realizar interações entre os subsistemas. O núcleo é responsável por registrar, identificar, iniciar e coordenar o funcionamento de todos os módulos e subsistemas que compõe o motor gráfico. Além dos subsistemas, no núcleo também se encontra um conjunto de sub-módulos básicos que são utilizados por esses subsistemas, e que as vezes também são utilizados pela camada de aplicação.

O núcleo oferece as seguintes funcionalidades (Maia, 2005):

a) Serviços comumente utilizados pelos subsistemas;

b) Políticas aplicadas ao carregamento de mídias;

c) Políticas aplicadas ao gerenciamento de memória;

d) Políticas aplicadas ao tratamento de particularidade da plataforma corrente;

e) Políticas aplicadas ao escalonamento das tarefas executadas pelo motor gráfico;

f) Políticas aplicadas ao sistema de eventos (interação do usuário com os dispositivos de entrada disponíveis);

g) Manutenção do registro de atividade e erros do motor gráfico (log);

h)Descrição da interface para a especificação e manipulação de cena;

Na Figura 8, são apresentadas as principais funcionalidade do núcleo em um motor gráfico de acordo com o autor.

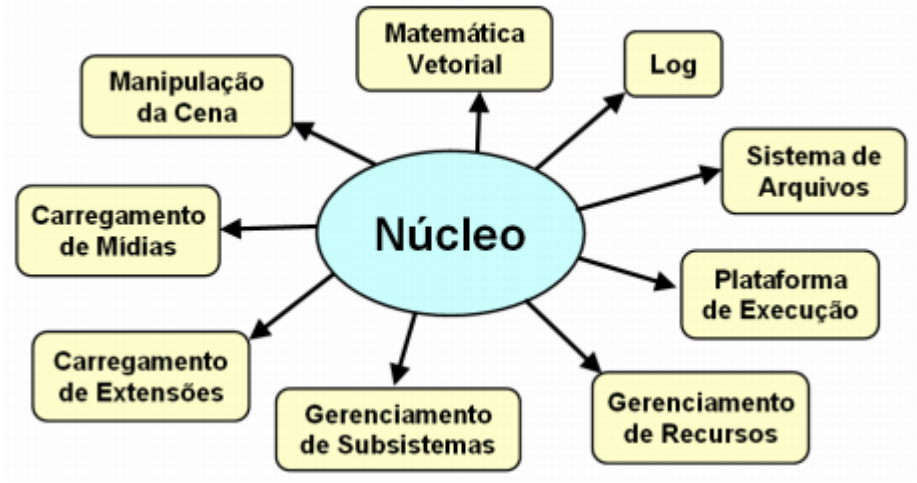


Figura 8 - As principais funcionalidades do núcleo em um motor gráfico. (Maia, 2003 citado por Maia et al., 2005).

Alguns motores podem ser expandidos em tempo de execução através de *plugins*, que são carregados como bibliotecas de vínculo dinâmico e registram módulos adicionais no motor gráfico (OGRE3D, 2004; Fly3D, 2004; Epic Games, 2004; Id Software, 2004, citado por Maia, 2005). Segundo Maia (2005), a utilização desses *plugins* permite isolar o código de subsistemas e módulos semelhantes em diferentes *plugin*, de maneira a permitir que a camada de aplicação escolha o mais conveniente.

Além disso, o autor ainda cita que pode-se corrigir falha em um módulo pela simples substituição do *plugin* correspondente, sem a necessidade de refazer o código executável do motor gráfico ou de aplicações. Se o motor suporta o carregamento de *plugins*, o núcleo define as políticas adotadas durante a localização, o carregamento, a identificação e a inserção dos *plugins* na estrutura de execução adotada pelo motor gráfico.

Para melhor entender o funcionamento do núcleo é importante também saber conceitos e o funcionamento dos subsistemas. Veremos isso no próximo tópico.

### 2.3.2.Subsistemas

De acordo com Maia (2005) subsistema representa um módulo específico do motor gráfico que é utilizado pela camada de aplicação para algumas funcionalidades. Dessa forma, cada subsistema é caracterizado pelo domínio de problemas para o qual é destinado e pelas

tarefas que desempenha como módulo do motor gráfico, além das tecnologias que utiliza para desempenhar essas tarefas. O subsistema de renderização em tempo real, por exemplo, caracteriza-se por sintetizar imagens a uma taxa de exibição interativa com base numa cena tridimensional e, dentre as tarefas que desempenha, estão: a determinação da visibilidade dos objetos na cena, a geração texturas e a especificação sistemas de coordenadas através de matrizes.

Os subsistemas necessitam de uma variedade de tecnologias, técnicas, algoritmos e bibliotecas adequados à sua construção. No caso do subsistema de renderização em tempo real, por exemplo, existem várias APIs disponíveis para acessar o hardware gráfico, como OpenGL (Segal&Akeley, 2004) citado por Maia(2005).

Para Bernardes et al (BERNARDES, 2004 citado por Malfatti, 2009) os subsistemas são conhecidos um conjunto de módulos, e os comumente encontrados em motores são:

a) Modulo gráfico para renderização 2D ou 3D: Também conhecido como renderizador, é um dos componentes mais complexos de um motor, pois tem a possibilidade de desenhar, com velocidade suficiente, os componentes visuais da aplicação formados por objetos 2D e 3D (estáticos ou dinâmicos). Geralmente esse módulo faz uso de diversas técnicas de otimização tais como diminuir o detalhamento dos objetos à medida que sua distância aumenta em relação ao observador (*Levelof Detail*) e não desenhar polígonos ocultos ou objetos que estejam fora do volume de visualização (*Culling*).

b) Módulo de Física: O sistema de física tem por objetivo identificar entidades virtuais estão em colisão, simular corpos rígidos, deformáveis e movimentos com aceleração, resistência e gravidade, que são fatores essenciais para que o realismo seja mantido.

c) Módulo de Dispositivos: Esse módulo monitora e capta dados e eventos disparados pelos equipamentos utilizados pelo usuário para interagir com a aplicação, como mouse, teclado e joystick.

d) Módulo de Som: A renderização sonora é importante para determinar parâmetros como atenuação, volume e frequência de um som base na posição e orientação do usuário no ambiente virtual.

e) Módulo de Inteligência Artificial: Esse módulo deve oferecer mecanismos para que o desenvolvedor possa atribuir comportamentos a entidades controladas pela aplicação.

f) Módulo de Rede: O módulo de rede deve oferecer recursos que seja possível realizar a comunicação entre computadores que participam de uma mesma simulação simultaneamente. Através desse módulo é possível manter a consistência do mundo virtual utilizado em aplicações colaborativas.

### 2.3.3 Carregamento de Elementos do Mundo Virtual

Numa aplicação com motores gráficos, diversos tipos de mídias são manipulados em tempo de execução, como, por exemplo, imagens e sons. De acordo com Maia(2005), antes das mídias virtuais se integrarem a aplicação, as mesmas devem ser carregadas no formato de arquivo, que pode ser proprietário do motor gráfico ou nativo das ferramentas de edição utilizadas durante a confecção dessas mídias.

Quando uma mídia está armazenada nas estruturas de dados adequadas, o núcleo ou um subsistemas do motor é capaz de manipular essa mídia de acordo com as necessidades da aplicação. Um exemplo disso, é uma imagem poder ser carregada na memória principal, possibilitando sua manipulação através de um subsistema de processamento digital de imagens para a criação de uma textura bidimensional (MAIA, 2005).

### 2.3.4 Estrutura de um Motor Gráfico

Geralmente, nas aplicações com motores gráficos uma tela é representada por uma cena onde são desenhados os objetos visuais chamados de *sprites* que não contém layout que possa receber ou agrupar esses componentes. Não existem componentes pré-definidos pelo motor ou pelo sistema operacional que possa ser usado na tela, logo, é necessário que o desenvolvedor crie esses elementos informando o tamanho e qual posição deve ser desenhado na tela.

Um *sprite* representa um arquivo de imagem que será mostrado na cena. Assim, é possível criar a partir de editores de imagem, componentes visuais de forma personalizada buscando aumentar a interatividade da aplicação. Sendo essa uma das vantagens em desenvolver aplicação com motor gráfico.

A estrutura da cena de uma aplicação com motor gráfico se baseia em um *loop* de renderização, onde ocorre o tratamento de eventos, atualização dos objetos desenhados,

renderização e no final do *loop* apaga tudo que foi desenhado e o ciclo é reiniciado como mostra na Figura 9.

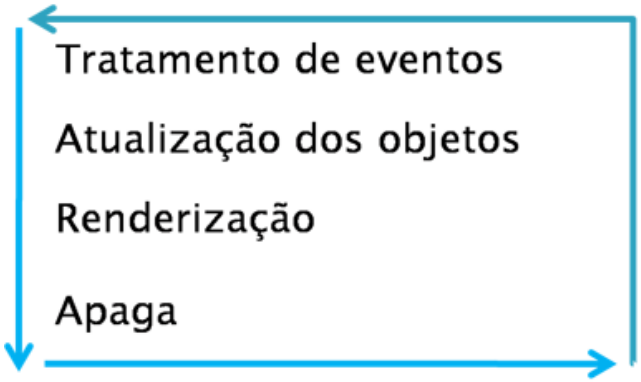


Figura 9 – Loop de Renderização (Elaborada pelo autor)

Dessa forma, se em um ciclo do loop o objeto mudou de quadro, ele será apagado no final do loop e redesenhado na nova posição no loop seguinte.

É importante salientar que o tratamento de eventos nos motores é feito utilizando os módulos de entrada e física, que são responsáveis por verificar as colisões entre dois objetos ou quando algum dispositivo de entrada é chamado.

### 2.3.5 Motor Gráfico AndGraphics

O motor gráfico AndGraphics, desenvolvido por Silvano Maneck Malfatti, tem uma estrutura simples e de fácil entendimento, além de possui diversos exemplos que auxilia o seu estudo. Por isso, ele foi escolhido para o desenvolvimento do estudo de caso.

No Android, o motor AndGraphics utiliza uma *activity* para renderiza as cenas da aplicação, assim não existe o conceito de pilha que existe nas aplicações tradicional, o motor permite que seja chamado qualquer outra cena sem ter que voltar para a anterior. A Figura 10 ajuda a entender como é organizado a troca de contexto no motor.

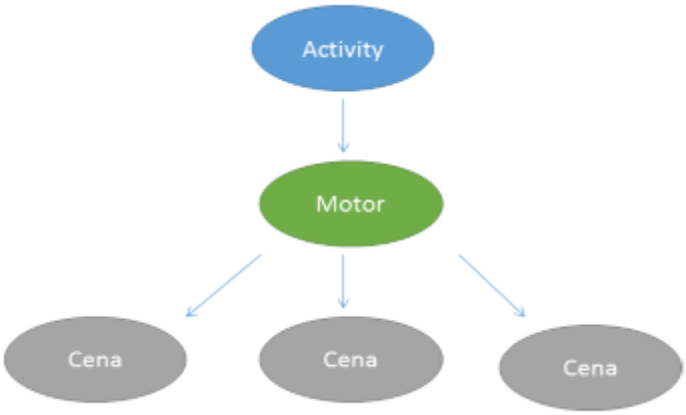


Figura 10 – Estrutura de um motor gráfico (Elaborada pelo autor).

De tal modo, se for requisitado ao motor a troca de cenas, ele irá carregara a próxima cena e finalizar a cena anterior.

A Tabela 1 mostra o que representa cada classe utilizada pelo motor AndGraphics nas camadas de Aplicação, Núcleo, Subsistemas.

| Camadas | Classes |
|---|---|
| **Camada de Aplicação** | AGScene: Responsável pela abstração de uma cena ou contexto da aplicação.<br>AGSprite: Utilizada para criar elementos gráficos independentes e animados.<br>AGAnimation: Classe responsável pelo gerenciamento de uma animação associada a um Sprite.<br>AGVector2D: Responsável armazenamento de um par de valores x e y. |
| **Núcleo** | AGGameManager: Responsável pelo gerenciamento de estados e cenas, utilizadas na aplicação.<br>AGSoundManager: Classe responsável pelo gerenciamento geral de sons tendo em vista que contém as referências para as classes AGMusic e AGSoudEffect. |

| | |
|---|---|
| | AGInputManager: Classe responsável pela centralização dos objetos responsáveis pelo tratamento de eventos do tipo *touch* e acelerômetro.<br>AGScreenManager: Classe responsável pelo armazenamento de informações da tela. |
| **Subsistemas.** | AGMusic. Classe responsável pelo gerenciamento e reprodução de sons utilizando *streaming*.<br>AGNioBuffer: Responsável por carregar os elementos visuais.<br>AGAccelerometer: Utilizada para fornecer dados referentes ao acelerômetro do dispositivo.<br>AGTouchScreen: Classe responsável pelo gerenciamento de eventos do tipo *touch*.<br>AGSoundEffect: Classe responsável pelo carregamento e reprodução de efeitos sonoros rápidos.<br>AGTimer: Classe utilizada para medir intervalos de tempo durante a execução da aplicação. |

Tabela 1– Classes por camadas do motor AndGraphics

### 2.3.6 Comparativo entre desenvolvimento de forma tradicional e o desenvolvimento utilizado motor gráfico AndGraphics

Como citado na seção 2.2, é possível perceber que a estrutura de uma aplicação tradicional é bem organizada, pois tem um padrão de desenvolvimento bem definidos, com componentes visuais e métodos pré-prontos, sendo também que o sistema operacional faz o controle da troca de contexto e renderização facilitando o trabalho do desenvolvedor.

No motor gráfico exigem maior trabalho do desenvolvedor, já que ele terá que construir os componentes visuais, programar uma lógica para o tratamento de eventos e atualização das imagens.

Para esclarecer melhor a diferença de recurso entre os dois tipos de programação, será apresentada a Tabela 2, analisando o motor AndGraphics e os componentes tradicionais do Android.

| | Motor Gráfico AndGraphics | Componentes Tradicionais |
|---|---|---|
| Tela da Aplicação | Representado por uma cena, onde o desenho dos elementos visuais e a lógica do modelo de negócios ficam na mesma classe. | Representado por uma activity e uma arquivo XML de forma separada seguindo o padrão MVC. |
| Layout | Fica na responsabilidade do desenvolvedor efetuar o cálculo informando em qual posição deseja desenha o objeto com base nas coordenadas X e Y. | Gerenciadores de layout predefinidos que organiza os componentes de forma automática. |
| Componentes Visuais | Não existe componentes predefinidos. Deve ser tratado pelo desenvolvedor. | Pode ser usado componentes predefinidos pela plataforma (quais são os componentes). |
| Troca de Contexto | Gerenciado pelo motor. | Funciona como uma pilha, que é gerenciada pelo Sistema Operacional. |
| Tratamento de Eventos | Deve ser criada uma lógica utilizando o módulo de entrada e de física. | Possui métodos predefinidos. |
| Renderização | Gerenciado pelo motor, porém dependo da complexidade da aplicação | Controlado pelo Sistema operacional. |

| | | |
|---|---|---|
| | é necessário que seja tratado pelo desenvolvedor. | |
| Estados | Controlado pelo desenvolvedor. | Controlado pelo Sistema operacional. |
| Entrada | Deve ser tratado pelo desenvolvedor. | Métodos já predefinidos. |

Tabela 2- Tabela 2 – Comparação entre desenvolvimento de aplicação tradicional e aplicação utilizando motor gráfico AndGraphics. (Elaborada pelo autor).

# 3 METODOLOGIA

Para elaboração do presente trabalho foi adotada a seguinte metodologia:

I. Primeiramente foi realizado o estudo do problema, sendo realizada uma pesquisa sobre os temas relacionadas, tais como Android, sua arquitetura, componentes de aplicação e padrão de projeto e posteriormente o conceito de motores gráficos. O resultado da pesquisa foi apresentada anteriormente no referencial teórico.
II. Após entender os conceitos, foi feito o mapeamento e definição de uma arquitetura de aplicação baseada em renderização de baixo nível. Dessa forma, foi definido o motor gráfico a ser utilizado, o estudo de caso e as ferramentas utilizadas no desenvolvimento.
III. Posteriormente foi definido o estudo de caso desse trabalho que é o desenvolvimento do aplicativo AnimePizza, sendo desenvolvido um projeto de telas com objetivo de avaliar a interface, e quais as próximas etapas para o desenvolvimento. Sendo elas: Prototipação da tela, Tratamento de eventos, Troca de Contexto, Ligação entre a parte gráfica e o modelo de negócios, teste e, por fim avaliação com usuário.

Vale salientar que as ferramentas utilizadas foram:

- IDE Eclipse: utilizado para o desenvolvimento da estudo de caso, utilizando a linguagem Java.
- Editor de imagem CorelDrawX7, para fazer a edição e personalização das imagens utilizadas.
- *Smartphone* Samsung *Galaxy* Core II, com sistema operacional Android.
- Notebook Sony vaio, com sistema operacional Windows 10.

# 4 PROJETO ANIME PIZZA

Este capítulo apresenta as partes do desenvolvimento do estudo de caso AnimePizza. Nele é descrito o processo de construção, as dificuldades, e soluções propostas para a resolução de cada situação comparando o modelo tradicional.

O projeto foi divido nas seguintes etapas: construção do projeto de telas, prototipação da tela, tratamento de eventos, modelagem da tela, implementação do modelo de negócios, teste e avaliação do usuário.

## 4.1 PROJETO DE TELA

A construção de projeto de tela foi o primeiro passo no desenvolvimento do sistema, pois é por meio dele pode-se ter uma prévia de como seria modelado as etapas para o desenvolvimento do estudo de caso. A Figura 11 mostra o primeiro projeto desenvolvido.

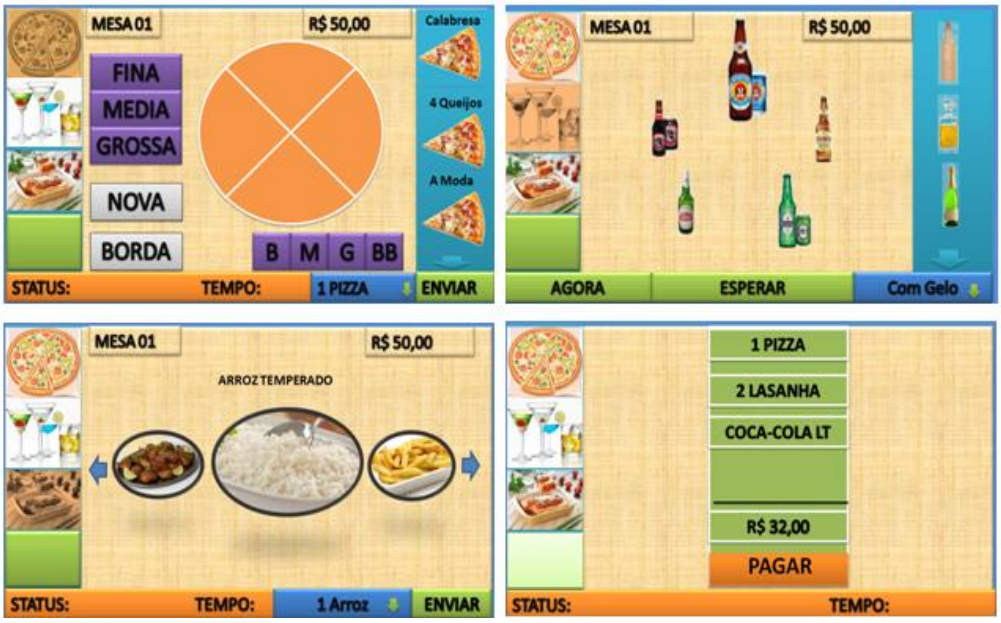


Figura 11 – Projeto de Telas AnimePizza (Elaborado pelo autor)

O projeto de telas se baseava em quatro telas, a de pizza onde o cliente poderia montar a pizza escolher o tipo de massa, tamanho e borda, bebidas no qual seria apresentada uma lista de forma circula, extras onde seria apresentado uma tela de forma linear e menu apresenta os pedidos realizados.

## 4.2 PROTOTIPAÇÃO DA TELA

Nesta seção será apresentado os principais dificuldades e passos para prototipação das telas utilizando motores gráficos e fazendo uma comparação com o modelo tradicional plataforma Android.

### 4.2.1 Tamanho e Posicionamento dos objetos

Como apresentado na Tabela 2, o motor gráfico não tem componentes visuais pré-definidos, assim o primeiro passo para a construção da prototipação foi criar as imagem que compõe a tela da aplicação utilizando editores de imagens.

Devido à grande diversidade no tamanho de telas dos dispositivos móveis que utiliza a plataforma Android, todas as imagens utilizadas no estudo de caso AnimePizza, tiveram que ser criada com a dimensões de *pixel* em potência de 2, pois evita que o ao alocar a imagem para desenhar na tela, o sistema faça o arredondamento errado e a imagem não apareça.

Após criar as imagens para a tela a ser desenvolvida, o próximo passo é desenhar o objeto na cena. Como mostra no Quadro 1.

```
vrFundo_Solido_Marron = createSprite(R.drawable.gr_fundo_solido_marrom, 1024, 512, 1024, 512, "", "", null);
vrFundo_Solido_Marron.setScreenPercent(100.0F, 100.0F);
vrFundo_Solido_Marron.vrPosition.setXY(AGScreenManager.iScreenWidth / 2,
AGScreenManager.iScreenHeight / 2);
```

Quadro 1- Desenhando um sprite.

Ao chamar o método *createSprite()* para desenhar o s*prite*, o sistema aloca essa imagem na memória do dispositivo, e o motor adapta a imagem ao tamanho da tela de acordo os percentuais passado como parâmetro no método *setScreenPercent()*, após definir o tamanho, a imagem é posicionada passando as coordenadas para método *setXY()*. Esse código é repetido para todos os *sprites* que serão desenhados na cena.

Uma das dificuldades encontrada para desenvolver a interface do estudo de caso foi deixá-lo adaptável ao diversos tamanhos de telas. Pois as coordenadas onde será desenhada o objeto deve esta proporcional ao tamanho da tela. Para resolver essa situação é necessário passar como parâmetro no método *setXY()*, um cálculo envolvendo o tamanho da largura e altura da tela ou pegar a posição de outro objeto que já foi desenhado utilizando a largura e altura da tela que é disponibilizada pelo motor gráfico.

Levando em conta que o motor gráfico AndGraphics posiciona os *sprites* pelo centro da imagem, um objeto pode ser desenhado no centro da tela passando como coordenadas a metade da largura e altura da tela, como mostra a Figura 12.

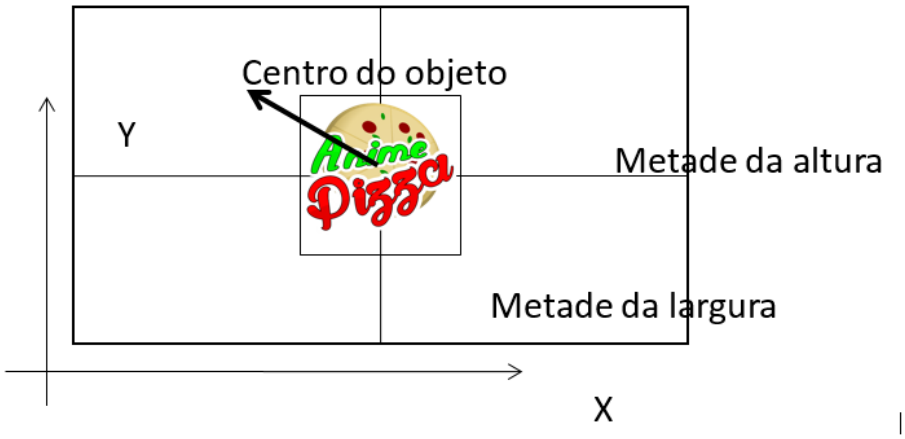


Figura 12- Forma com é desenhado os objetos no motor gráfico AndGraphics.

No desenvolvimento de uma aplicação tradicional, o programador, pode simplesmente através do recurso de sua IDE, apenas arrastar o botão e soltar em um layout gráfico que corresponde à tela do dispositivo, como mostra a Figura 13. Desse modo é possível ter uma previa da interface sem precisar compilar a aplicação em um dispositivo ou emulador e sem precisar fazer um cálculo para o posicionamento da imagem.

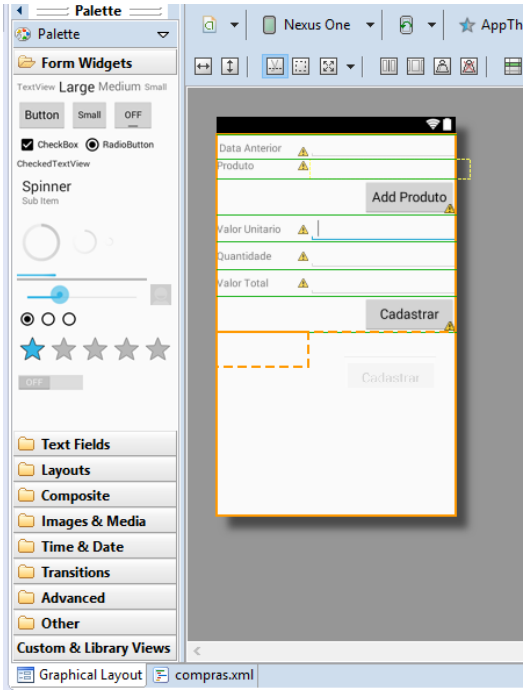


Figura 13– Componente sendo arrastado.

Dessa forma a aplicação com motor gráfico tende a ter maior tempo no processo de construção, pois é necessário desenvolver a imagem e implementar uma lógica para que as objetos fiquem adaptáveis a qualquer tipo de tela. Além disso, a aplicação exigi mais processamento do dispositivo móvel, pois ele terá que alocar a imagem na memória e sempre efetuar um cálculo para que o objeto seja desenhado na posição correta.

### 4.2.2 Sistema de Camadas

A cena do motor é responsável por desenhar os *sprites* e armazena-los em uma pilha onde o último objeto desenhado ira aparecer acima dos demais. Porém na tela onde são apresentadas as bebidas em alguns momentos, uma determinada imagem deverá aparecer acima de um determinado objeto e em outra parte da tela essa mesma bebida deverá aparecer abaixo desse objeto, como mostrar na Figura 14. Para isso foi preciso desenvolver um conjunto de imagem que pudesse passar ao usuário a sensação de uma única imagem e implementar uma lógica de renderização sempre que uma a lista de bebidas é carregada.


Figura 14– Sprite desenhado entre duas camadas

Na Figura 14 o retângulo em vermelho mostra um *Sprite* que aparentemente, está com metade da imagem está acima do fundo e a outra metade abaixo. Isso acontece porque a imagem do fundo que é desenhada na base da pilha foi duplicada e recortada em 25% no lado direito e 25% no lado esquerdo através de editores de imagens, essas novas imagens são desenhadas nas laterais da tela, sendo que elas estão uma camada acima dos *sprite* que representa as bebidas, assim para criar a sensação de uma única tela, foram necessários desenhar três imagens como mostra na Figura 15.

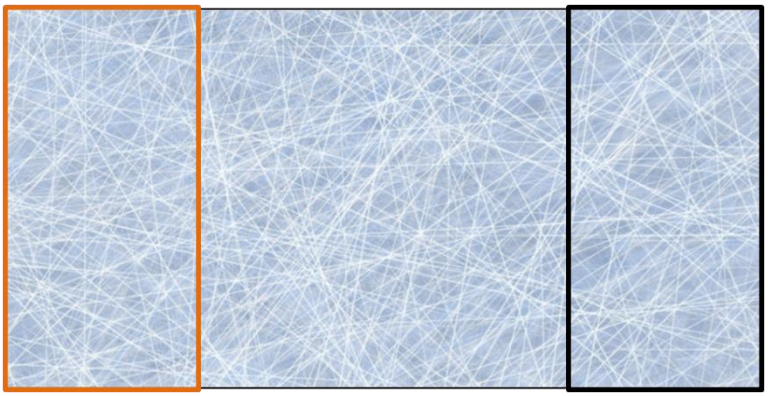

Figura 15 – Fundo da tela de bebidas.

Apesar de todo o trabalho com a edição das imagens, não foi o suficiente para resolver o problema, pois sempre que o usuário seleciona um tipo de bebida, os *sprites* da lista de bebidas são removidos e uma nova lista é criada, sendo adicionada no topo da pilha, assim a nova lista é desenhada acima de todas as três imagens.

Para tratar essa situação, foi preciso sobrescrever o método *render()* na cena, no qual é renderizado primeiro lista de bebidas e depois os fundo direito e esquerdo como mostra o Quadro 2.

```
        public void render() {
            super.render();


            if(vetListaDeBebidas != null) {
               for(int indice = 0; indice < vetListaDeBebidas.size(); indice ++) {

                        vetListaDeBebidas.get(indice).render();

               }
            }
            vrFundo_esquerdo.render();

        vrFundo_direito.render();
    }
```

Quadro 2– Sobrescrita do método render().

Essa mesma lógica foi utilizada em diversas partes do sistema, como na tela onde usuário confirma os pedidos de bebidas e extras, e na tela de pedidos onde é apresenta a lista de pedidos enviados e não enviados.

Nos componentes convencionais do Android a implementação de um sistema de camada, pode ser feito utilizando o gerenciador de layout, *FrameLayout* que funciona como uma pilha, no entanto alterar dinamicamente a sequência das camadas não é um processo

simples. Nesse contexto o motor gráfico apresenta ser mais vantajoso, pois permite que o desenvolvedor tenha maior controle das camadas.

A prototipação é uma fase importante no desenvolvimento de uma aplicação com motor gráfico já que ele não separa o *controller* da *view*. Assim o desenvolvedor deve implementar a parte lógica dos componentes visuais que são apresentados na tela e a lógica do modelo de negócio na mesma classe. Com isso a prototipação correspondem a uma grande parte do código do projeto.

### 4.3 TRATAMENTO DE EVENTOS

Como visto na seção 2.3.4 a estrutura da aplicação com motores gráficos se baseia no *loop* de renderização, assim dentro desse método definimos uma estrutura como mostrado no Quadro 3 para capturar as entradas do sistema e realizar a saídas para o usuário. Sendo esta etapa responsável por grande parte da interação com o usuário.

```
@Override
 public void loop() {
       // TODO Auto-generated method stub

                    if (AGInputManager.vrTouchEvents.backButtonClicked())
                    {
                    }
                    if (AGInputManager.vrTouchEvents.screenDown())
                    {
                    }
                    if (AGInputManager.vrTouchEvents.screenDragged())
                    {
                    }
                    if (AGInputManager.vrTouchEvents.backButtonClicked())
                    {
                    }

}
```

Quadro 3-–Loop de renderização.

Como foi visto na tabela 1, a classe AGInputManager é responsável pelo tratamento de eventos do tipo touch, assim, através do atributo vrTouchEvents da classe AGTouchScreen ele fornece as funções de entrada que pode ser manipulada pelo desenvolvedor. Abaixo é descrito a função de cada método.

screenClicked():Verifica se  o botão voltar do Android for clicado.

screenDown(): Verifica quando a tela é pressionada.

screenDragged(): Verifica se houve um arrasto na tela.

backButtonClicked():Verifica quando o botão voltar do Android foi pressiona.

Através das entradas disponibilizadas é possível fazer o tratamento de eventos, verificando se na área pressionada, deve ser executada alguma ação. Como por exemplo, quando essa área corresponde a um botão na cena, assim o sistema responde efetuando alterações na interface e executando a ação solicitada, o Quadro 4 mostra os passos executados dentro do *loop* quando um botão é pressionado e solto.

```
...
if (AGInputManager.vrTouchEvents.screenDown())
  {
                if (this.vrPedidos.collideCircular((int)AGInputManager.vrTouchEvents.fPosX,
(int)AGInputManager.vrTouchEvents.fPosY))
        {

          this.aCaoBotao = 1;
          this.vrPedidos.bVisible = false;
          this.vrPedidosPress.bVisible = true;
        }
        else if (vrPizza.collideCircular((int) AGInputManager.vrTouchEvents.fPosX,
(int)AGInputManager.vrTouchEvents.fPosY))
        {
          this.aCaoBotao = 2;
          this.vrPizza.bVisible = false;
          this.vrPizzaPress.bVisible = true;

        }
        else if (this.vrBebida.collideCircular((int)AGInputManager.vrTouchEvents.fPosX,
(int)AGInputManager.vrTouchEvents.fPosY))
        {
              this.aCaoBotao = 3;
           this.vrBebida.bVisible = false;
           this.vrBebidaPress.bVisible = true;

        }
              else if(this.vrExtra.collideCircular((int)AGInputManager.vrTouchEvents.fPosX,
(int)AGInputManager.vrTouchEvents.fPosY))
              {
                    this.aCaoBotao = 4;
          this.vrExtra.bVisible = false;
          this.vrExtraPress.bVisible = true;

        }

    }


  if(AGInputManager.vrTouchEvents.screenClicked())
  {
        voltarEstadoIniciaBotao();
        if (aCaoBotao == 1)
        {
              aCaoBotao=0;
          this.vrGameManager.setCurrentScene(4);
          return;
        }
        else if (aCaoBotao == 2)
        {
```

```
                    this.aCaoBotao = 0;
                  this.vrGameManager.setCurrentScene(1);
                  return;

                }
                else if (aCaoBotao == 3)
                {
                        this.aCaoBotao = 0;
                       this.vrGameManager.setCurrentScene(2);
                        return;
                }

                aCaoBotao=0;
                atualizaFase();


        }

    ...
```

Quadro 4- Tratamento de eventos no loop.

O Quadro 4 mostra que a variável aCaoBotao é responsável por receber e passar qual ação deve ser executada dentro do *loop*, Assim quando a tela é pressionada, e esse ponto colide com um *sprite* que representa um botão, é passada a variável, o código da ação que deve ser executada quando a tela for solta. Além disso, para que o usuário perceba que o botão foi pressionado é efetuado uma troca na visibilidade do botão. Com isso quando a tela é pressionada, o *sprite* que representa a imagem de um botão solto passa a não ter visibilidade e o *sprite* que representa o botão pressionado passa a ser desenhado, quando a tela é solta acontece o processo contrário. Essa lógica é utilizada em todas as cenas da aplicação proposta.

Na aplicação tradicional, o desenvolvedor não precisa, fazer o tratamento de quando a tela é pressionada, solta ou arrastada. Isso é feito de forma automática pelo sistema operacional.

Uma das principais dificuldades encontrada relacionada ao tratamento de eventos foi realizar a movimentação da lista que contém as fatias de pizza. A solução para esse situação dividiu-se em duas etapas, a primeira consistiu em desenvolver um método que movimenta a lista passando uma posição Y para o primeiro item da lista e atualizando a posição Y dos demais itens da lista como mostra o Quadro 5. No entanto a lista só deverá ser movimentada caso obedeça aos limites estabelecidos na função.

```
public void moverLista(float py) {
        float tamanho_lista = 0;
        for(int index = 0; index < vetListaDePizza.length; index++) {
```

```
                    tamanho_lista = tamanho_lista + vetListaDePizza[index].getSpriteHeight();
                }
                if(py<AGScreenManager.iScreenHeight - vetListaDePizza[0].getSpriteHeight()/2){

                    vetListaDePizza[0].vrPosition.setY(AGScreenManager.iScreenHeight -
vetListaDePizza[0].getSpriteHeight()/2);
                    for(int index = 1; index < vetListaDePizza.length; index++) {
                        vetListaDePizza[index].vrPosition.fY =vetListaDePizza[index-
1].vrPosition.fY
                                - vetListaDePizza[index-1].getSpriteHeight()/2 -
vetListaDePizza[index].getSpriteHeight()/2;
                }
                }
                else if(py > bt_ad_pedido.vrPosition.getY()+bt_ad_pedido.getSpriteHeight()/2 +
tamanho_lista - vetListaDePizza[0].getSpriteHeight()/2){

        vetListaDePizza[0].vrPosition.setY((bt_ad_pedido.vrPosition.getY()+bt_ad_pedido.getSpriteHeight
()/2)+tamanho_lista - vetListaDePizza[0].getSpriteHeight()/2);
                    for(int index = 1; index < vetListaDePizza.length; index++) {

                        vetListaDePizza[index].vrPosition.fY =vetListaDePizza[index-
1].vrPosition.fY
                                - vetListaDePizza[index-1].getSpriteHeight()/2 -
vetListaDePizza[index].getSpriteHeight()/2;
                }
                }
                else{
                    vetListaDePizza[0].vrPosition.setY(py);
                    for(int index = 1; index < vetListaDePizza.length; index++) {
                        vetListaDePizza[index].vrPosition.fY =vetListaDePizza[index-
1].vrPosition.fY
                                - vetListaDePizza[index-1].getSpriteHeight()/2 -
vetListaDePizza[index].getSpriteHeight()/2;
                }
                }
            }
```

Quadro 5– Método utilizado para mover uma lista.

A segunda etapa consistiu em passar qual a posição Y para essa função. Assim ao detectar que ocorreu o arraste na área da lista, o sistema guarda a posição em que a tela está sendo tocada e após 40 milissegundos efetua uma subtração entre a posição guardada e a atual posição em que a tela está sendo tocada, desse resultado é então subtraído a primeira posição da lista e passado como parâmetro para a função como mostra o Quadro 6. Essa lógica é utilizada em todas as listas do aplicativo.

```
...
if(vrfundo_direito.collide((int) AGInputManager.vrTouchEvents.fPosX, (int) AGInputManager.vrTouchEvents.fPosY))

...

//Guarda a posição antes do tempo
posYAnterior = AGInputManager.vrTouchEvents.fPosY;

//pausa o tempo 40 milesimos de segundos
try {
        Thread.sleep(40);
} catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
}

...

posYDiferenca = posYAnterior - AGInputManager.vrTouchEvents.fPosY;

...

moverLista(vetListaDePizza[0].vrPosition.fY - posYDiferenca);
. . .
```

Quadro 6- Lógica para mover uma lista dentro do loop.

Outra parte importante no tratamento de eventos utilizada na tela de pizza é permitir que o usuário arraste uma fatia e solte na bandeja. Para isso o sistema verifica se um item da lista de fatias está sendo pressionado, então ele é duplicado através da método clonar() ( mostrado no Quando 7) , permitindo que o usuário possa mover o objeto.

```
//Clonar

public AGSprite clonar(AGSprite sprite, float posx, float posy)
{
                                        AGSprite Auxiliar = null;
                        Auxiliar        =       new AGSprite(AGGameManager.vrOpenGL,sprite.getiImageCode(), sprite.iFrameWidth, sprite.iFrameHeight, sprite.getiImageWidth(), sprite.getiImageHeight(), sprite.getTipo(), sprite.getSabor(), sprite.getPreco());
```

```
                                        Auxiliar.vrScale.fX=sprite.vrScale.fX;
                                        Auxiliar.vrScale.fY=sprite.vrScale.fY;
                                        Auxiliar.iMirror = sprite.iMirror;
                                        Auxiliar.fAngle = sprite.fAngle;
                                        Auxiliar.vrPosition.setXY(posx, posy);
                                        Auxiliar.fWidthDiminuir = (sprite.getfWidthDiminuir());
                                        Auxiliar.bVisible =true;
                                        vrSprites.add(Auxiliar);
                                        return Auxiliar;

}
```

Quadro 7- Método clonar().

Quando o usuário arrasta a fatia sobre a bandeja o sistema automaticamente altera o ângulo da imagem como se ele fosse encaixar na bandeja, com a intenção que o usuário entenda onde ela deve ser solta. Isso acontece porque na camada abaixo do *sprite* que representa a bandeja, existe uma lista de fatia já montada em forma de pizza na qual, o sistema verifica se existe colisão entre a fatia arrastada e um item dessa lista, quando ocorre essa colisão o sistema altera o valor ângulo e do espelhamento do *sprite* arrastado, para o valor do item em que foi colidido como mostra no Quadro 8.

```
        …
        for (int index = 0; index < vetBandejaModeloDeOito.length; index++) {

            if (vetBandejaModeloDeOito[index].collideMetade(
                        (int) AGInputManager.vrTouchEvents.fPosX,
                        (int) AGInputManager.vrTouchEvents.fPosY)) {

                        vetMoverPizza.get(indiceFatiaPizza).fAngle =
vetBandejaModeloDeOito[index].fAngle;
                        vetMoverPizza.get(indiceFatiaPizza).iMirror =
vetBandejaModeloDeOito[index].iMirror;


            }
        }
        ....
```

Quadro 8– Alterando o ângulo e o espelhamento de um *sprite*.

Nas aplicações tradicionais do Android o tratamento de eventos não exige tanto do desenvolvedor em situações básicas como quando, a lista é arrastada, um botão é clicado, um item da lista é pressionado, entre outros, já que a o sistema operacional disponibiliza diversos métodos prontos. Diferentemente da aplicação com motores gráficos, no qual programador é responsável por tratar todas as entradas e saídas da aplicação.

## 4.4 TROCA DE CONTEXTO

Como visto na seção 2.3.4 o motor utiliza uma *activity* para realizar a renderização das cenas da aplicação, para isso é necessário registrar no gerenciador geral quais cenas ele deve gerenciar, como mostrar o Quadro 9.

```
        protected void onCreate(Bundle paramBundle)
        {
          super.onCreate(paramBundle);
          this.vrGerenciadorCenas = new AGGameManager(this, false);
          this.vrGerenciadorCenas.addScene(new Menu(this.vrGerenciadorCenas));
          this.vrGerenciadorCenas.addScene(new Pizza(this.vrGerenciadorCenas));
          this.vrGerenciadorCenas.addScene(new Bebida(this.vrGerenciadorCenas));
          this.vrGerenciadorCenas.addScene(new Extra(this.vrGerenciadorCenas));
          this.vrGerenciadorCenas.addScene(new Pedidos(this.vrGerenciadorCenas));

}
```

Quadro 9– Registrando Cenas no motor AndGraphics.

A primeira cena adicionada ao motor será a cena inicial da aplicação. Para trocar de cena, basta chamar o método setCurrentScene() e passar como parâmetro o índice da cena desejada como mostra o Quadro 10, sendo que esse índice corresponde a sequência em que a cena foi adicionada na lista de cenas do motor.

```
this.vrGameManager.setCurrentScene(2);
```

Quadro 10– Chamando outra Cena.

Na estrutura tradicional do Android geralmente um tela corresponde a uma activity mais um arquivo xml, assim para que possa ocorrer a troca de contexto, é necessário que seja registrado no *AndroidManifest* como mostra no Quadro 11.

```
..
activity
        android:name="com.tradicional.android.MainActivity"
        android:label="@string/app_name" >
        <intent-filter>
          <action android:name="android.intent.action.MAIN" />

          <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
      </activity>
      <activity
        android:name=".TelaPizza"
        android:screenOrientation="portrait"
        android:label="">

</activity>  ...
```

Quadro 11– Activities sendo registrada no AndroidManifest.

Após o registro no *AndroidManifest* para que seja efetuada a troca de tela é necessário chamar o método *startActivity*(), passado como parâmetro uma *intent*, que precisa se instanciada recebendo a activity atual e a próxima *activiy*, como mostra o Quadro 12.

```
Intent it = new Intent(this, Tela2.class);
startActivity(it);
```

Quadro 12– Exemplo Intent (LECHETA, 2013).

Por tanto, em termos de esforço do programador, no modelo tradicional, quase não existe diferença, já que as duas formas utilizam passos parecidos para fazer as trocas de telas, registrando as telas em um gerenciador e chamando dentro das classes. Porém a estrutura da aplicação fica diferente, pois como citado na seção 2.1.1.1 o Android organiza as *activities* em uma pilha e o motor organiza as cenas.

## 4.5 LIGAÇÃO ENTRE PARTE GRÁFICA E MODELO DE NEGÓCIOS

Para modelar os dados da aplicação, foi criado um pacote de classes Model que é responsável por guardar as informações que são compartilhadas na aplicação. Também foi necessário criar um gerenciador de arquivo, para poder salvar os dados e configurações da aplicação no disco interno do dispositivo. Posteriormente foi criada as imagens com o nome dos pedidos e os números de 0 a 9 que são apresentadas como informação na interface.

Para que o usuário pudesse visualizar na tela os preços dos pedidos, foi desenvolvido um método que recebe como parâmetro uma *String* e retorna uma lista de *sprite*, onde cada item dessa lista representa um caractere da *String*. Como mostra o Quadro 13.

```
public ArrayList<AGSprite> preco(String preco) {

        ArrayList<AGSprite> vetPreco= new ArrayList<AGSprite>();
        String numeros = "0123456789";
        String ponto = ".";
        vetPreco.clear();
        for (int ids = 0; ids < preco.length(); ids++) {
                for (int idn = 0; idn < 10; idn++) {

                        if (preco.charAt(ids) == ponto.charAt(0)) {

vetPreco.add(clonar(vrVirgula,vrVirgula.vrPosition.getX(),vrVirgula.vrPosition.getY() ));
                                break;
                        } else if (preco.charAt(ids) == numeros.charAt(idn)) {

vetPreco.add(clonar(vrNumeros[idn],vrNumeros[idn].vrPosition.getX(),vrNumeros[idn].vrPosition
.getY()));
                                break;
                        }
                }
        }
        return vetPreco;
}
```

Quadro 13– Métodos utilizado para apresentar os preços.

O Quadro 13 mostra que dentro da função é criada uma variável local que recebe no formato de *string* uma sequência numérica de 0 a 9 que é comparada a cada caractere do parâmetro, quando esses caracteres são iguais, o sistema adiciona ao *ArrayList* local um *sprite* que é clonado a partir de um item do vetor de *sprite* de dez posições que contém as imagens numéricas. Esse passo é repetido até que todas as posições do parâmetro sejam percorridas.

Outra dificuldade importante foi mostrar na tela de bebida uma lista de pedidos com quantidade, nome e preço. Para resolver esse problema, na classe AGSprite, foi adicionado três atributos que são: tipo, nome, preço. Assim quando um *Sprite* que corresponde a um produto é criado, é adicionado aos três atributos as informações desse produto como mostra o Quadro 14.

```
        vetCervejas = new AGSprite[12];
        vetCervejas[0] = createSprite(R.drawable.b_cervejas_brahma_umlitro, 256, 1024, 256, 1024, "Cerveja", "Brahma 1L", "6.00");
        vetCervejas[0].setTipo("Bebida");
        vetCervejas[0].setNome("Coca Cola 1L");
        vetCervejas[0].setPreco("5.00");
    ....
```

Quadro 14– Primeira posição do vetor de bebidas sendo criada.

Também foi necessário criar uma classe que recebe como atributo um nome sendo tipo AGSprite, quantidade e preço sendo ArrayList do tipo AGSprite, como mostra o Quadro 15.

```
public class List_Adapter
{
  private AGSprite nome;
  private ArrayList<AGSprite> preco;
  private ArrayList<AGSprite> qtd;
...
}
```

Quadro 15– Classe *List_Adapter*.

Na cena de Bebida, existe uma lista que guarda os *sprites* adicionados na bandeja. Assim, quando o usuário clica no botão de adicionar o pedido, o sistema cria um *ArrayList* do tipo Bebida_Model, e adiciona os atributos, tipo, nome e preço nesse *ArrayList*. Essa

lógica também é utilizada nas demais telas. O Quadro 16 mostra esse passo importante do sistema.

```
Bebida_Model bebida= null;

for(int idx=0;idx<vetPedidos.size();idx++)
{
        bebida= new Bebida_Model();
        bebida.setTipo(vetPedidos.get(idx).getTipo());
        bebida.setNome(vetPedidos.get(idx).getSabor());
        bebida.setValor(vetPedidos.get(idx).getPreco());
        bebida = null;

}
```

Quadro 16- Ligando parte gráfica a modelo de negócios.

Após é feita ordenação da lista pela quantidade do tipo de bebida, então é desenhado a quantidade, nome, preço e adicionado uma lista do tipo *List_Adapter* e após é feita organização dessa lista.

A modelagem de dados utilizando motor gráfico é um dos processos que mais demandou esforço e tempo, no desenvolvimento da aplicação, além gasta muito processamento, pois é preciso carregar cada nome, e número que será apresentado na tela. Já nas aplicações tradicionais existe componentes como *ListAdapter, ListView* e *TextView* que facilitaria no desenvolvimento.

### 4.6 TESTES

Os testes representam um gasto de tempo no desenvolvimento das aplicações com motores gráfico no Android. Isso acontece porque só é possível visualizar o resultado de cada tela desenhada após a compilação, assim não é possível ter uma previa da interface até que ela seja compilada.

Vale salientar também, que o emulador disponível na IDE não conseguiu executar a aplicação com bom desempenho, com isso foi necessário o uso de um dispositivo móvel com a plataforma Android.

Ao testa o aplicativo em um dispositivo móvel, pode ser percebido uma grande diferença de desempenho do comparado a uma aplicação tradicional. Isso acontece porque

sempre que é criado uma sprite na tela, é preciso carregar a imagem na memória do dispositivo.

## 4.7 AVALIAÇÃO COM USUÁRIO

Esta seção representa a parte final do projeto, sendo divido em duas partes, Apresentação dos sistemas utilizados na pesquisa e os resultados da pesquisa realizada.

### 4.7.1 Apresentação dos Sistemas Utilizados na Pesquisa

Esta seção será para a apresentação de como manipular os dois sistemas que foram comparados na pesquisa, o aplicativo AnimePizza, que foi desenvolvido para a fundamentação desse trabalho; e do PizzApp, aplicativo desenvolvido por Newton Guimarães Filho disponível no *Google Play*. Sendo o primeiro desenvolvido utilizando motor gráfico, e o ultimo feito de com componentes tradicionais.

#### *4.7.1.1 AnimePizza*

Ao entrar no aplicativo, a Figura 16 é a tela inicial, onde está o direcionamento para as demais telas da aplicação.


Figura 16– Menu do aplicativo AnimePizza

É por meio dessa tela apresentada na Figura 16, que pode acessar os pedidos já realizados (caso já se tenha feito o pedido); realizar o pedido de extras; realizar pedido de bebidas, realizar pedido de pizzas.

Ao clicar na opção pizza, irá apresentar a tela onde permite que o usuário monte a sua pizza. Definindo sabores, tamanho, tipo de bordas e outros, conforme apresentado nas Figura 17 e 18.

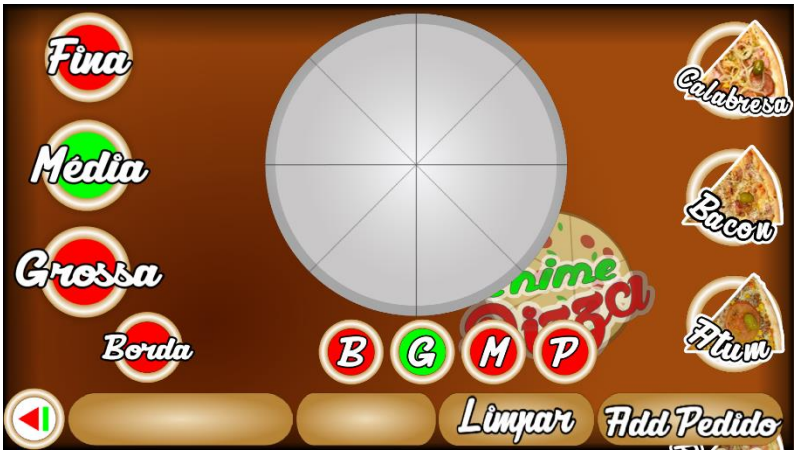


Figura 17– Montagem da pizza.


Figura 18 - Pizza montada.

Nessa tela, assim como as demais que envolvem pedidos ao selecionar um sabor, ou modificar a composição da pizza é apresentado o valor de acordo com o que foi selecionado. Quando termina a montagem da pizza, a aplicação exibirá pra o usuário as informações que foi por ele escolhida pra que o mesmo confirme a sua escolha.

Ao clicar na opção bebidas, o funcionamento é semelhante ao apresentado na opção pizza. Como pode se observar na Figura 19, é apresentado ao usuário as opções de bebidas, permitindo que o mesmo arraste a que quiser até a bandeja.


Figura 19- Selecionar bebidas.

Quando o usuário vai adicionar os pedidos a lista é exposto para ele as informações referente ao pedido, como por exemplo quantidades, sabores, valores, dentre outros. Essas informações aparece conforme a Figura 20.


Figura 20 - Confirmar pedidos.

Cabe ao usuário confirmar ou cancelar envio desses produtos a lista. Essa mesma situação acontece quando está realizando o pedido de pizza.

Se por acaso o usuário colocar na bandeja algum produto que ele não deseja consumir ele pode limpar esse produto se ainda não estiver adicionado à lista, e se já estiver adicionado na lista pode excluir posteriormente da lista de pedidos, por meio do menu pedidos.

Ao clicar na opção pedidos na tela inicial, a tela exposta na Figura 21 irá aparecer uma lista vazia, caso não se tenha realizado nenhum pedido ainda ou os pedidos que já foram adicionados à lista, conforme apresentado na Figura 22.


Figura 21- Nenhum pedido cadastrado


Figura 22-Lista de Pedidos

Quando se tem certeza de que quer realmente confirmar os pedidos que estão listados basta clicar no botão enviar, e então a aplicação confirmará se realmente deseja fechar o pedido, como mostra a Figura 23, situação em que está pedindo uma pizza. E na Figura 24, situação em que está sendo pedido bebidas.


Figura 23– Pizza adicionada à lista de pedidos.


Figura 24- Bebidas adicionados à lista de pedidos.

Existe também a possibilidade de cancelar os pedidos que estão na lista. Se por acaso o usuário desejar cancelar toda a lista de pedidos, existem duas situação: os pedidos que foram adicionados à lista mas não foram enviados ainda, e os pedidos que possuem o status de já enviados. Esses somente podem ser cancelados se não estiver sido confirmado o pedido, já aqueles podem ser cancelados a qualquer momento. Nesse momento o sistema irá confirmar com o usuário se ele realmente deseja cancelar a lista, conforme apresentado na Figura 25. Também é possível excluir ou editar apenas um item específico da lista. Para isso clica na lista sobre o pedido e seguir as instruções e conforme apresentado na Figura 26.


Figura 25 – Confirmar cancelamento de todos os pedidos.

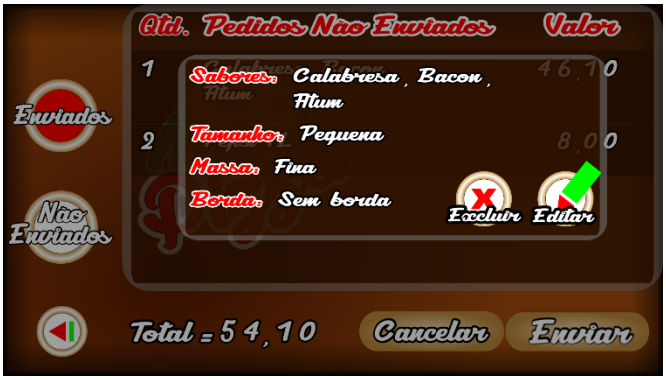


Figura 26 - Cancelar ou Editar item da lista de pedidos.

Vale ressaltar que o processo de exclusão é o mesmo para todos os tipos de produtos.

#### *4.7.1.2 PizzApp*

Nessa aplicação, o usuário tem a opção de realizar pedidos de pizza e bebidas. Na Figura 27 pode-se observar essa situação.

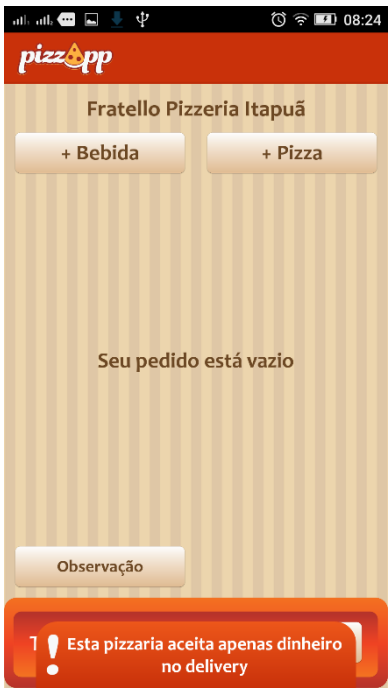


Figura 27-Lista de pedidos(vazia).


Figura 28 – Adicionar Bebidas

Na Figura 27 pode-se observar que não foi realizado nenhum pedido. Para acrescentar pedidos a lista é necessário clicar no botão pizza ou bebidas. Pressionando o botão bebidas, aparecerá a tela exposta na Figura 28 que permite que o usuário adicionar suas bebidas a lista e tem a opção de definir a quantidade de cada produto.

O processo de seleção de pizza é semelhante ao de bebida, como pode-se observar na Figura 29 é apresentada uma lista de pizza com suas devidas composições. Quando a pizza é adicionada ela é acrescentada à lista de pedido, e caso o usuário deseja fazer alguma observação sobre a pizza é só clicar no botão observações que fica localizado logo abaixo a lista de pedidos. Nesse campo observação o usuário pode digitar sua observação, tal como dizer que quer pizza sem cebola, o campo observação é apresentado na Figura 30.


Figura 29-Adicionar Pizza


Figura 30 - Adicionar Pizza

A partir do momento que algum produto é colocado na lista, o valor da soma dos produtos é apresentado na parte inferior da lista. Juntamente com o valor, também aprece o botão entrega que leva o usuário a tela de pagamento, conforme exposto na Figura 31.


Figura 31– Pagamento/Entrega

Antes de fechar o pedido a aplicação mostra as informações referente a valores de pedido, embalagem, e frete; e informa como pode ser realizado o pagamento, conforme exposto nas Figura 32. Após fechar o pedido o usuário poderá visualizar as informações do seu pedido em, como é apresentado na Figura 33.


Figura 32 - Valores e Forma de Pagamento.


Figura 33– Informações sobre o pedido.

Após o pedido ter sido realizado o usuário poderá cancela-lo. Ao fazer isso, aplicação apresentará a mensagem apresentada na Figura 34. Cabe ao usuário definir se quer ou não cancelar o pedido.

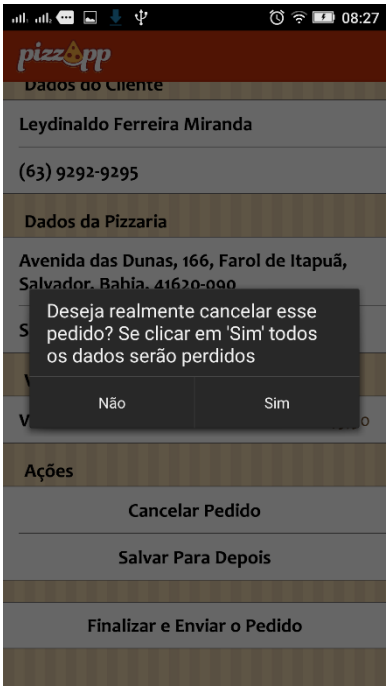


Figura 34– Cancelar o pedido.

4.7.2 Resultados

Com o objetivo de avaliar o grau de satisfação do usuário quando interage com interfaces personalizadas foi realizada uma pesquisa com 10 pessoas entre os dias, 23 a 30 de novembro de 2016, com clientes de restaurantes e lanchonetes e, universitários.

A pesquisa consiste em fazer um comparativo entre uma interface personalizada, utilizando componentes não convencionais do Android e uma interface tradicional com componentes convencionais da plataforma.

Para realizar a pesquisa foi utilizado um *Smartphone Samsung Galaxy* J5, com os dois aplicativos instalados, no qual o usuário teve que realizar os seguintes pedidos.

I. Uma Pizza, tamanho grande (8 fatias), com massa fina, borda de catupiry e sabores de bacon e muçarela.
II. Duas bebidas sendo 1 refrigerante Coca-Cola um litro e 1 Refrigerante Guaraná em lata.

Após realizar o pedido o usuário deve visualizar o que foi pedido sem enviar ou realizar o pagamento do pedido. Para finalizar a tarefas o usuário deve cancelar o pedido.

Na pesquisa, conforme estabelecido no questionário iniciou por conhecer os usuários através da- *Identificação geral* avaliando sexo e a idade. Outro ponto importante foi, analisar o usuário quanto a sua experiência em utilizar aplicativos para realizar compra, pedidos ou operações bancarias.

Após, foram avaliados o pontos mais importante para a pesquisa, que é a facilidade, atratividade e interatividade quando comparado a um aplicativo com interface tradicional e os níveis de atratividade e satisfação do aplicativo Anime Pizza.

A pesquisa revelou que ao comparando a facilidade em realizar os pedidos nos dois aplicativos, o sistema tradicional apresenta uma média vantagem como mostra o Gráfico 1.

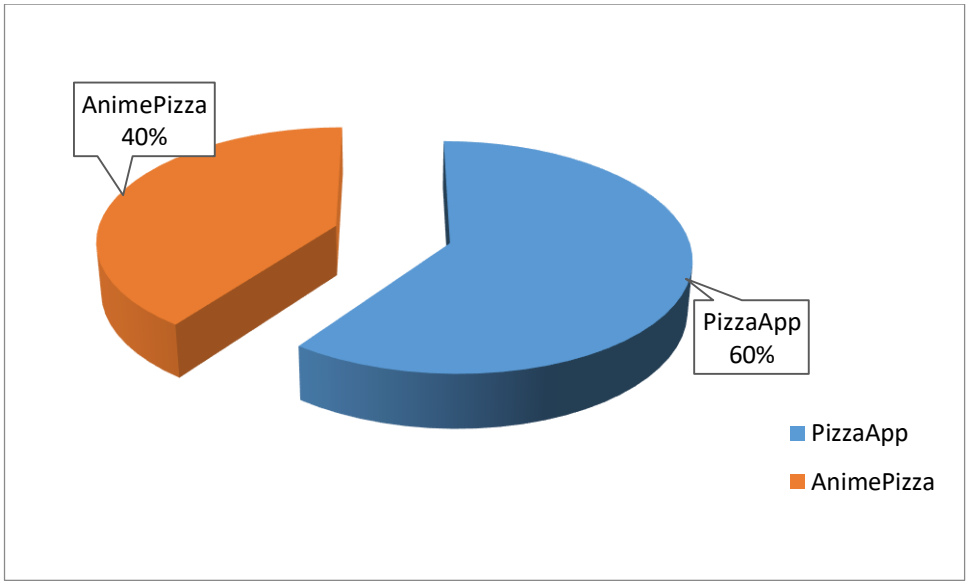


Gráfico 1–Facilidade, quando comparado os dois aplicativos.

Esse resultado pode ser visto pelo fato do usuário não está acostumando com uma tela diferenciada, onde ele teve que pressionar, e arrastar cada pedido até a bandeja.

Um dos quesitos mais importante para a pesquisa, quando questionado sobre qual layout é mais atrativos, a interface do aplicativo AnimePizza, teve uma grande vantagem como mostra o Gráfico 2.

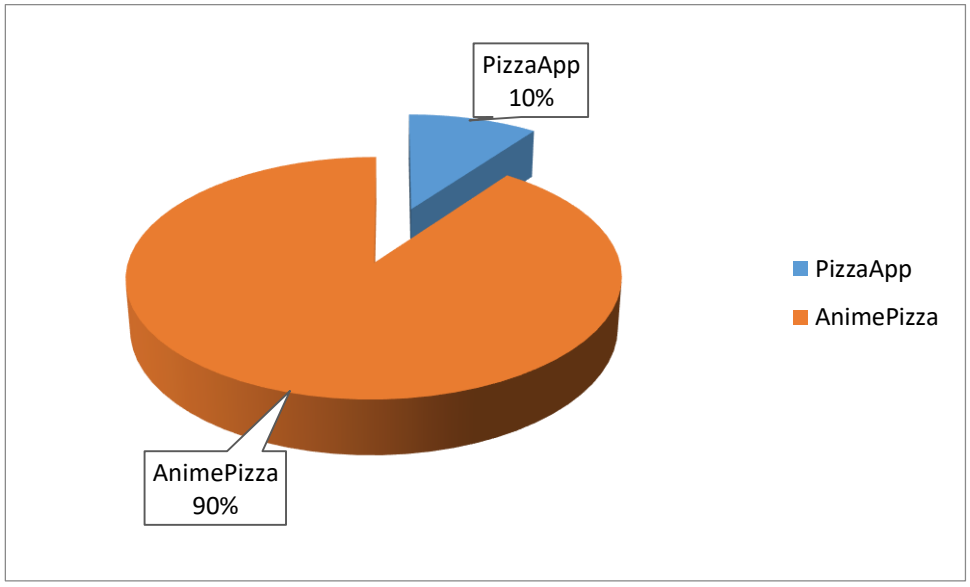


Gráfico 2–Atratividade, quando comparado os dois modelo de interface.

Isso se deve ao fato do aplicativo ter componentes visuais personalizados e comportamento diferenciado.

Com o objetivo de melhor avaliar a interface personalizada, na última parte da pesquisa, foi questionado como o usuário avalia somente a interface do aplicativo AnimePizza. Com isso a pesquisa revelou que mesmo quando não comparado a interface tradicional, o aplicativo AnimePizza se manteve com uma aceitação muito alta, sendo avaliado como Ótimo e Bom, como mostra o Gráfico 3.

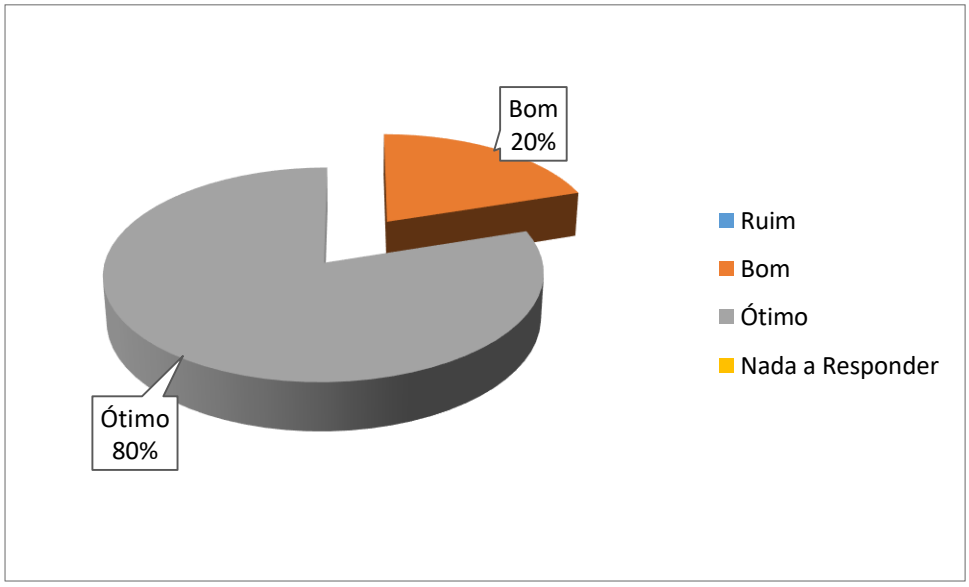


Gráfico 3–Atratividade do Aplicativo.

Na última pergunta foi analisado o grau de satisfação do usuário em utilizar uma interface personalizada. Sendo que, comparada a atratividade, a satisfação do usuário foi menor, como mostra o Gráfico 4. Isso pode ser explicado pelo fato de uma aplicação, com motores gráficos, não apresentar o mesmo desempenho de uma aplicação tradicional, já que, o dispositivo gasta mais memória para executar uma aplicação que utiliza um grande número de imagens.

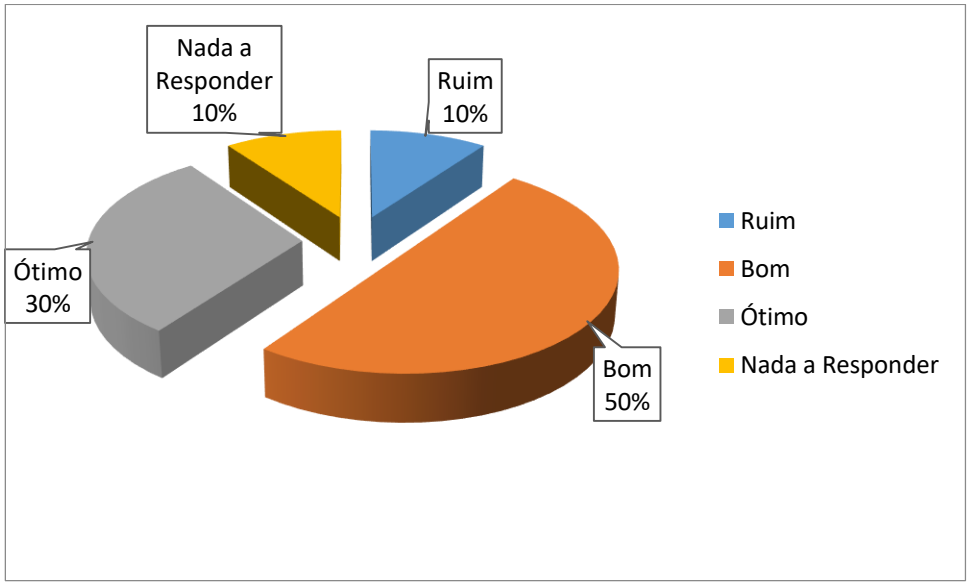


Gráfico 4 – Satisfação do Usuário.

# 5 CONSIDERAÇÕES FINAIS

Este trabalho apresentou o esforço necessário, no desenvolvimento de um sistema com interface personalizada utilizando, um motor gráfico. Diferenciando a organização e a estrutura do modelo de desenvolvimento tradicional e do modelo com motores gráficos. Apresentando etapas no processo de construção do aplicativo AnimePizza, que foi utilizado na pesquisa realizada com os usuários.

O aplicativo foi desenvolvido como estudo de caso, tendo um *layout* personalizado, contendo animações gráficas, onde é possível montar a pizza com diversos sabores e escolher, tamanho, massa e borda, além de bebidas e extras. Sendo que foi desenvolvido um modelo de negócios permitindo que a aplicação mostre ao usuário informações como nome de pedidos, quantidade e preço.

Ao realizar o estudo desse trabalho, pode-se perceber que a plataforma Android possui arquitetura e componentes que facilita no desenvolvimento de aplicações tradicionais. Além disso, possui recursos como OpenGl que podem ser utilizado por motores gráficos na construção de aplicativos com interfaces personalizadas.

Conclui-se também, que desenvolver aplicações comerciais com layout personalizado que não utiliza recursos convencionais do Android, exige do desenvolvedor, conhecimentos que não se restringem somente a lógica de programação, além de tempo e esforço. Ainda que esse processo seja facilitado quando se usa motores gráficos.

Com resultado da pesquisa realizada, foi possível avaliar o usuário, quando interage com interface personalizada, constatando que, a interface do aplicativo AnimePizza, possui menor facilidade comparada à interface tradicional, porém é considerada muito atrativa e com grau de satisfação entre bom e ótimo. Sendo que essa diferença entre a atratividade e a satisfação ocorre devido ao aplicativo, não ter o mesmo desempenho das aplicações tradicionais. Portanto desenvolver aplicações comerciais, com motores gráficos se torna viável, por ter um bom índice de satisfação do usuário.

## 5.1 TRABALHOS FUTUROS

Um dos pontos negativos no desenvolvimento de uma aplicação com motores gráficos para a plataforma Android, é o desempenho da aplicação, assim um dos trabalhos futuros consiste em desenvolver um modulo que faço o carregamento do elementos visuais de forma rápida.

Outra situação negativa, apresentada no desenvolvimento do estudo de caso, foi ligar o modelo de negócio a parte gráfica, sendo que um dos motivos que exigiu grande esforço no desenvolvimento da aplicação. Por tanto, outro trabalho futuro, consiste em desenvolver um modulo para o motor gráfico AndGraphics, que possa desenhar uma imagem com textos, apenas passando, através de método uma *string*.

Se incluído essas funcionalidade no motor, as aplicação desenvolvida com motores gráficos passaram a ter maior aceitação.

# 6 REFERÊNCIAS


ABLESON, W.F. et al. **Android em Ação.** 3ª. ed. São Paulo: NovatecElsiever, 2012.

ANDROID DEVELOPER. **API Guides. Android Developers**. 2016. Disponível em: <http://developer.android.com/guide>. Acesso em: 11 ago. 2016.

BUCCO, Rafael. **Android se torna a plataforma preferida dos desenvolvedores mobile.** Disponível em: <http://www.telesintese.com.br/android-se-torna-plataforma-preferida-dos-desenvolvedores-mobile/>. Acesso em: 5 ago. 2016.

CARDENAS, D.S.T. **Pílula de Conhecimento – Android.** Disponível em: <https://www.docdroid.net/roWXh2g/apostila-pilula-android.pdf.html>. Acesso em: 21 ago. 2016.

GARBIN. **Estudo da evolução das interfaces home-computador,** 2010. Disponível em: <http://www.tcc.sc.usp.br/tce/disponiveis/18/180450/tce-25112011-104445/publico/Garbin_Sander_Maeda.pdf>. Acesso em: 30 ago. 2016.

LECHETA, Ricardo. R. **Google Android: aprenda a criar aplicações para dispositivos móveis com Android SDK**. 3ª. ed. São Paulo: Novatec Editora, 2013.

MAIA, J,G,R. **CRAbGE – uma arquitetura para motores gráficos flexíveis, expansíveis e portáveis**,2005. Dissertação de mestrado, Universidade Federal do Ceará.

MALFATTI, **S.M. ENCIMA - um motor para o desenvolvimento de aplicações de realidade virtual**, 2009. Dissertação de mestrado, Instituto Militar de Engenharia.

MANSSOUR, I.H. Introdução à OpenGL, 2000. Disponível em: <http://www.inf.pucrs.br/~manssour/OpenGL/Tutorial.html>. Acesso em: 23 set. 2016.

REIS, B.Tutorial Android: **Content Provider – Entenda O Content Provider – Parte 1**, 2014. Disponível em: < http://www.decom.ufop.br/imobilis/tutorial-android-content-provider-parte-1-entenda-o-content-provider/>. Acesso em: 20 out. 2016.

RIBEIRO, R.R. **Artigo WebMobile 21 - Construindo layouts complexos em Android**, 2013 Disponível em:<http://www.devmedia.com.br/artigo-webmobile-21-construindo-layouts-complexos-em-android/10761>. Acesso em: 12 set. 2016.

RIBEIRO, R.R. **Artigo WebMobile 18 - Android: um novo paradigma de desenvolvimento móvel**, 2013. Disponível em:http://www.devmedia.com.br/artigo-webmobile-18-android-um-novo-paradigma-de-desenvolvimento-movel/9350. Acesso em: 12 set. 2016.

THIENGO, V. **Ciclo de Vida de Uma Atividade no Android**, 2013. Disponível em:<http://www.thiengo.com.br/ciclo-de-vida-de-uma-atividade-no-android>. Acesso em: 22 ago. 2016.

# 7 APÊNDICES

## 7.1 Apêndice I

Este apêndice apresenta o pesquisa realizada com usuário.


**Fundação Universidade do Tocantins – UNITINS**

**Acadêmico:** Leydinaldo Ferreira Miranda

**Orientador:** Silvano Malffati (Professor Mestre)

**Tema da pesquisa:** Avaliar o grau de satisfação do usuário quando interage com interfaces personalizadas.

A pesquisa consiste em fazer um comparativo entre uma interface personaliza utilizando os componentes não convencionais do Android e uma interface tradicional.

O usuário deve realizar a tarefa abaixo e responder o questionário na pagina seguinte.

Realizar o seguinte pedido nos dois aplicativos.

Uma Pizza, Tamanho Grande (8 Fatias), Massa Fina, com Borda de Catupiry, e Sabores de Bacon e Mussarela.

Duas Bebidas sendo 1 Refrigerante Coca-Cola um Litro e 1 Refrigerante Guaraná em Lata.

Após o usuário deve visualizar o que foi pedido antes de enviar ou realizar o pagamento do pedido.

Para Finalizar a Tarefa o usuário deve cancelar o pedido.

**IDENTIFICAÇÃO GERAL:**

**1.1 Sexo:**

| | | | |
|---|---|---|---|
| | Feminino | | Masculino |

**1.2 Idade:**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 1<br>8 – 25 anos | | 2<br>6 – 35 anos | | 3<br>6 – 45 anos | | 4<br>5 – 55 anos | | Acim<br>a de 55 anos |

1- Você costuma utilizar aplicativo para realizar compras, pedidos ou operações bancarias?

( )Sim ( )Não

2- Você conseguiu realizar a atividade proposta nos dois aplicativos?

( )Sim, nos dois Aplicativos.

( )Não, somente no Aplicativo PizzaApp.

( )Não, somente no aplicativo AnimePizza.

( )Não consegui em nenhum dos dois.

3- Você encontrou dificuldade para realização da tarefa proposta?Se sim, informe abaixo?

_____

_____

_____

4- Qual dos aplicativos você encontrou mais facilidade para realizar a tarefa proposta?

( ) AnimePizza ( ) PizzaApp

5- Analisando o layout dos aplicativos propostos, qual você considera mais atrativo?

( ) AnimePizza ( ) PizzaApp

6- Analisando o layout dos aplicativos propostos, qual você considera mais Interativo?

( ) AnimePizza ( ) PizzaApp

7- Analisando o layout dos aplicativos propostos, qual você considera mais Intuitivo?

( ) AnimePizza ( ) PizzaApp

8- Como você avalia a atratividade da tela do aplicativo AnimePizza?

( )Ruim ( )Médio ( )Bom ( )Ótimo ( ) Nada a Responder

9- Analisando a interface do aplicativo AnimePizza, como você avalia o grau de satisfação?

( )Ruim ( ) Bom ( )Ótimo ( )Nada a Responder

10- Deixa aqui suas sugestões para o aplicativo AnimePizza

_____
_____
_____
_____
_____