Projeto e desenvolvimento de jogos eletrônicos multiplataforma: um estudo de caso utilizando Cocos2d-x

Marlon M. Andrade, Igor O. Knop

Centro de Ensino Superior de Juiz de Fora (CES/JF)
36.016-000 – Juiz de Fora – MG – Brazil

{marlonmandrade,igorknop}@gmail.com

Abstract. *The advance in mobile technologies and the fragmentation of the market creates a demand for games that is best leveraged when the same codebase is used to generate the same game on multiple platforms. This article describes the main concepts of games and the development of multi-platform video games and addresses the use of the framework Cocos2d-x and the language C++ for creating games as a case study.*

Resumo. *O avanço nas tecnologias móveis e a grande fragmentação do mercado gera uma demanda de jogos que é melhor aproveitada quando uma mesma base de código é utilizada para gerar o mesmo jogo em diversas plataformas. Este artigo descreve os principais conceitos relacionados a jogos e o desenvolvimento de jogos eletrônicos multiplataforma e aborda a utilização do framework Cocos2d-x e a linguagem C++ para a criação de jogos como estudo de caso.*

1. Introdução

O conceito de jogo transcende o meio eletrônico e na literatura encontramos diversas definições ao longo dos tempos. Para Salen e Zimmerman (2012) um jogo é um sistema no qual os jogadores se envolvem em um conflito artificial, definido por regras, que implica um resultado quantificável. Não fazendo distinção entre os jogos digitais e não digitais, as qualidades que definem um jogo em uma mídia também o definem em outra.

A indústria de jogos está em franca expansão, segundo a empresa de pesquisa Newzoo, no ano de 2014, as 25 maiores companhias geraram US$ 54,1 bilhões em receita. Além disto, Needleman (2015) cita que a projeção de crescimento do mercado de jogos para dispositivos móveis nos Estados Unidos deverá crescer mais do que todas outras mídias. Desta forma, a oportunidade presente nesse mercado é cada dia maior.

A grande diversidade e o potencial número de usuários para cada plataforma faz com que muitas empresas tenham a necessidade de criar uma mesma aplicação em diversas plataformas. Segundo relatório da *Gartner* (2015), foram vendidos mais de 1.2 bilhões de *smartphones* para usuários finais em 2014, e deste montante 98.9% estão divididos entre os três principais sistemas operacionais: *Android*, *iOS* e *Windows*. A fim de evitar o custo de implementar a mesma aplicação em três plataformas diferentes surge a necessidade da criação de aplicações multiplataforma, e desta forma, diversas ferramentas são criadas para atingir essa finalidade.

Ao analisar a listagem de jogos com maior rentabilidade nas lojas online nas principais plataformas muitos repetem. Jogos como *Clash of Clans*, *Candy Crush* e *Summoners War* aparecem em todas as listagens (AppAnnie 2015).

As ferramentas ou engines multiplataforma para jogos mais populares são: *Unity*, *Unreal Engine*, *Corona SDK*, *Marmalade SDK* e *Cocos2d-x*. Embora todas estas possuam versões gratuitas, algumas funcionalidades extras são disponibilizadas através da aquisição de uma licença, ou, como o caso da *Unreal Engine*, o desenvolvedor deve pagar um percentual dos lucros obtidos através do jogo, destas, somente o *Cocos2d-x* é totalmente gratuita e possui o código aberto. O *Cocos2d-x* é um conjunto de ferramentas de código aberto para auxiliar no desenvolvimento de jogos multiplataforma, que utiliza a linguagem *C++* e é usada por milhares de desenvolvedores ao redor do mundo.

Neste trabalho será apresentado conceitos e características de jogos e de desenvolvimento de jogos eletrônicos. Será feita criação de um estudo de caso utilizando a linguagem *C++* e o framework *Cocos2d-x* para a criação de um jogo multiplataforma, a escolha se deu principalmente por ser a única ferramenta com código aberta avaliada. Os principais conceitos e dificuldades relacionados a desenvolvimento de jogos multiplataforma serão apresentados e também uma análise do processo de desenvolvimento do mesmo e da maturidade das ferramentas utilizadas.

2. Jogos e design de jogos

Antes de se começar a criar um jogo é necessário definir as principais características do mesmo. Além disto, embora jogos muitas das vezes sejam associados a entretenimento e diversão é necessário separar a relação entre esses conceitos.

Segundo Adams (2009), bons jogos são divertidos e jogos ruins não, porém, diversão é uma resposta emocional ao ato de jogar um jogo, não é intrínseco ao jogo em si. Não é porque um jogo não é divertido que significa que não é um jogo.

Jane McGonigal (apud Rogers 2012) enumera os principais atributos de um jogo: Um objetivo: jogos claramente definem um objeto para os jogadores atingirem. É importante que os objetivos sejam desafiadores, porém alcançáveis. Idealmente, os jogadores estão sempre jogando no limite de suas habilidades. Objetivos dão aos jogadores um senso de propósito enquanto está jogando o jogo. Regras: jogos possuem regras que todos os jogadores devem concordar em seguir. As regras muitas das vezes fazem com que as conquistas do objetivo mais difíceis, o que por sua vez encoraja os jogadores a serem criativos. Retroalimentação: um jogo deve ter como dizer aos jogadores como eles estão progredindo. De fato, um sistema de retroalimentação interessante e criativo é a chave para se criar um jogo prazeroso. Participação voluntária: não é um jogo a não ser que você realmente queira jogar. Este aspecto dos jogos implica que os jogadores aceitem o objetivo, as regras e o sistema de retroalimentação.

2.1. Jogos eletrônicos e jogos não-eletrônicos

Tanto jogos eletrônicos quantos jogos não-eletrônicos são enquadrados dentro de todos os conceitos definidos para os jogos e possuem os mesmos objetivos, porém é

importante citar algumas diferenças, pois um dos objetivos parciais deste trabalho é a criação de um jogo eletrônico.

Uma primeira diferença é na exposição das regras aos jogadores. Em um jogo não-eletrônico, todos jogadores precisam conhecer previamente as regras a fim de conseguir jogar. Já em um jogo eletrônico a própria implementação do jogo serve como uma barreira para as ações possíveis. Para Adams (2009) a imersão em um jogo eletrônico é muito maior, um jogador pode simplesmente tentar uma ação sem ter que conhecer as regras e tentar interagir com o jogo. Os jogadores não vêem o jogo como um ambiente artificial temporário, mas sim como um universo alternativo no qual o jogador faz parte.

Uma segunda diferença de destaque é a forma de apresentação de um mundo ficcional de fantasia de um jogo. Em um jogo não-eletrônico, os aspectos que definem o mundo e o contexto é a cargo da imaginação do jogador, embora cartas, tabuleiros impressos e outros possam ajudar. Já os jogos eletrônicos podem ir muito além, com a utilização da tela e de sons o mundo pode ser sentido pelos jogadores diretamente, principalmente após a grande evolução dos gráficos com cenários foto realistas. Entretanto, a necessidade de se fornecer recursos visuais e auditivos torna o trabalho do projetista de jogos mais complexa, pois os jogadores estão menos suscetíveis a aceitar falhas na elaboração de ambientação.

2.2. Desenvolvimento de jogos

Segundo Adams (2009) o processo do *design* de jogos consiste em: Imaginar um jogo; Definir a forma como funcionará; Descrever os elementos que fazem o jogo (conceituais, funcionais, artísticos e outros); Transmitir as informações sobre o jogo para o time que irá construí-lo.

Para o desenvolvimento do jogo, métodos iterativos de desenvolvimento ágeis são muito utilizados. O desenvolvimento ágil depende do *feedback* e refinamento das iterações do jogo, e para Bates (2004), este método é efetivo pois a maioria dos projetos não se iniciam com uma clara definição dos requisitos.

A criação de um jogos é uma atividade multi disciplinar com diversos papéis. Este trabalho possui o foco exclusivo em apresentar o papel executado pelo desenvolvedor de jogos, não cabendo a este apresentar o trabalho realizado pelos outros papéis.

3. Plataformas móveis

A evolução dos dispositivos móveis e aumento da sua capacidade de processamento proporcionou um enorme avanço nos jogos eletrônicos portáteis. Com os *smartphones* e *tablets* cada pessoa passa a ter consigo, a todo momento, um computador capaz de apresentar gráficos avançados, computar milhões de operações por segundo e fornecer conectividade com outros dispositivos e a Internet.

Junto do lançamento do *iPhone* de segunda geração em Julho de 2008 a *Apple* fez o lançamento da *App Store*, onde terceiros poderiam desenvolver e publicar os aplicativos em sua loja. Com isto, a *Apple* criou um canal de distribuição que permitia

que os desenvolvedores monetizassem seus aplicativos, ou seja, poderiam cobrar pelos seus aplicativos sem ter que implementar uma plataforma de recebimento e cobrança.

Embora as lojas de publicação de aplicativos não sejam exclusivas para jogos, estes, representam grande parte do número de instalações e de receita. Conforme a listagem dos melhores aplicativos de 2014, publicado pela *Apple* em seu portal *iTunes Store*, da totalidade de aplicativos selecionados como os melhores do ano, 87% deles são jogos. Leo Mirani (2015), em um artigo publicado no portal *Quartz*, realizou uma análise dos dados publicados pela *Apple* e demonstrou que a categoria jogos é a principal fonte de receita gerada por aplicativos. Segundo relatório publicado pela *AppAnnie* (2015), empresa responsável na análise de dados de mercado das lojas online, no *Google Play*, loja online de aplicativos do *Google* para *Android*, todos os 10 primeiros aplicativos com maior rentabilidade nos Estados Unidos são jogos. Ainda sobre jogos para plataformas móveis, a revista *Fortune*, em um artigo de janeiro de 2015, cita que no ano de 2015 os jogos para plataformas móveis superarão o lucro dos jogos de console.

Além disto, o acesso aos jogos estão cada vez maior, destacam Zechner e Green (2011). A partir do momento que as pessoas utilizam *smartphones*, *tablets* e outros dispositivos, estes dispositivos automaticamente possuem a capacidade de executar jogos eletrônicos. Anteriormente, a pessoa deveria tomar uma decisão consciente de comprar um video game ou um computador específico para jogos.

O ato de criar aplicações para dispositivos móveis possui algumas diferenças em relação a aplicações para computadores pessoais ou para a web. Primeiro, o poder de processamento dos *smartphones*, embora sofra uma melhoria constante, ainda é menor que dos computadores. Além disto, os usuários dos aplicativos fazem o uso da aplicação de forma muito diferente das outras plataformas, em geral, os aplicativos são mais simples e diretos, e tem um propósito ou objetivo único. O usuário utiliza muito mais aplicações diferentes do que somente uma única aplicação com diversas funcionalidades.

Já a criação de jogos, possui as mesmas restrições, o poder de processamento gráfico dos dispositivos é menor, e em geral o tempo de permanência do usuário em um jogo nos dispositivos móveis é muito menor do que nas outras plataformas. Porém, ao mesmo tempo, esses fatores são também positivos para um desenvolvedor individual. Jogos mais simples e casuais possuem uma maior visibilidade.

O desenvolvimento de jogos é considerado uma tarefa difícil. Conforme Zechner e Green (2011) a grande quantidade de informações a serem digeridas antes mesmo de começar a escrever o jogo é o principal fator de dificuldade. Por isso, o uso de ferramentas que auxiliem com tarefas básicas que todo jogo deve lidar, tais como: entrada de dados, áudio, gráficos, entre outros é recomendado.

3.1. Desenvolvimento multiplataforma

O desenvolvimento multiplataforma de aplicações para dispositivos móveis ainda é visto com preconceitos. Cada plataforma possui suas formas de interação e consequentemente não há um grande reaproveitamento no código utilizando dentre as

plataformas. Porém, a interface dos jogos para *smartphones* não utiliza de componentes nativos da plataforma, e sim uma representação totalmente customizada de um mundo virtual projetado para o jogo. Desta forma, a utilização de somente uma base de código para todas as plataformas se torna algo mais simples. Para a utilização de somente uma base de código para todas as plataformas, deve-se existir compiladores para a linguagem destino da plataforma. E a única linguagem que é suportada oficialmente pelas principais plataformas móveis do mercado é o *C* e *C++*.

A principal vantagem ao se criar uma mesma base de código para múltiplas plataformas é poder distribuir o mesmo aplicativo em diferentes lojas e desta forma, conseguir atingir um público alvo maior com um menor esforço de desenvolvimento. Entretanto, diversos recursos que são disponibilizados por uma linguagem mais alto nível e também através de funções específicas da plataforma nativa não podem ser utilizadas.

Como ferramenta multiplataforma para desenvolvimento podemos citar a *Unity*, ferramenta de jogos mais popular que permite criar jogos 2d ou 3d, com uma vasta documentação e tutoriais. Além disto, outra ferramenta muito popular e poderosa é a *Unreal Engine*, que também permite a criação de jogos 2d ou 3d, é uma das ferramentas mais completas disponíveis de forma gratuita. Com foco maior em jogos 2d podemos citar a *Corona SDK*, *Marmalade SDK* e o *Cocos2d-x*. A principal diferença entre estas últimas é que o *Cocos2d-x* é a única totalmente gratuita e com código aberto. A *Corona SDK* possui versão gratuita, mas também possui versão paga provendo mais funcionalidades. A Tabela 1 demonstra as principais diferenças entre cada ferramenta. Desta forma, para este trabalho será escolhida a *Cocos2d-x* por ter um foco em jogos 2d e com código aberto.

Tabela 1. Comparação entre principais ferramentas multiplataforma

	Cocos2d-x	**Unity**	**Unreal Engine**	**Corona SDK**	**Marmalade SDK**
Gráficos	2d, mas suporta 3d	2d e 3d	2d e 3d	2d	2d, mas suporta 3d
Linguagens	*C++*, *JavaScript*, *Lua*	*C#*, *JavaScript*, *Boo*	*C++*	*Lua*	*C++*
Popularidade (resultados no google)	1.280.000	202.000.000	8.190.000	453.000	126.000
Multiplataforma	sim	sim	sim	sim	sim
Custo	grátis	grátis, mas versão *pro* (completa) paga	grátis, mas com 5% de *royalty* pelo lucro do seu jogo	grátis, mas sdk extra paga	grátis
Código aberto	sim	não	não	não	não

O *Cocos2d-x* é uma ferramenta que permite, utilizando *C++*, *JavaScript* ou *Lua*, programar o jogo em uma dessas linguagens e compilar diretamente para cada plataforma nativa. Caso o jogo seja escrito em *JavaScript* ou *Lua*, antes de compilar diretamente para a plataforma alvo, o *Cocos2d-x* traduz esse código em *C++*. Um projeto criado com a ferramenta de linha de comando do *Cocos2d-x* já disponibiliza esqueletos de projetos configurados para cada plataforma, bastando compilar o código para a plataforma destino desejada.

4. Desenvolvimento de jogos com o Cocos2d-x

Conforme a introdução do projeto, o *Cocos2d-x* é uma ferramenta multiplataforma para construir jogos, livros interativos e outras aplicações gráficas. É baseado no *cocos2d-iphone* e utiliza *C++*. Ele funciona no *iOS*, *Android*, *Windows Phone*, *Mac OS X*, *Windows* e *Linux*. O desenvolvimento do projeto pode ser realizado em qualquer ambiente de programação capaz de compilar código C++.

4.1. Metáfora de filme: Diretor e Cena

A principal metáfora utilizada pelo *Cocos2d-x* é a metáfora de filme. Existe um objeto central que faz o papel de diretor (`cocos2d::Director`[1]) e a representação de um conjunto de elementos na tela se dá através de uma cena (`cocos2d::Scene`), conforme ilustrado pela Figura 1 e a sua cena de introdução.

O diretor é o objeto responsável por realizar a troca de cenas e, consequentemente, dar ritmo ao jogo. Ainda a Figura 1 ilustra possíveis cenas de um jogo desde a sua abertura.

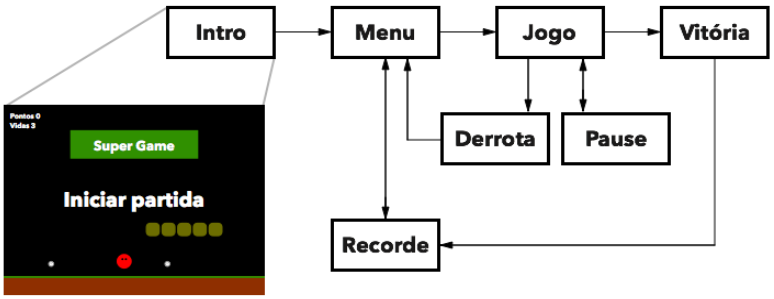

Figura 1. Exemplo da Cena de Introdução e transições entre cenas de um jogo desde a sua abertura - Fonte: do autor

4.2. Modelagem dos elementos

Além do diretor e das cenas, outros objetos principais que são utilizados no desenvolvimento um jogo são: nós (`cocos2d::Node`), sprites (`cocos2d::Sprite`), camadas (`cocos2d::Layer`) e ações (`cocos2d::Action`).

[1] `Director` é o nome da classe e `cocos2d` é o *namespace* que ela pertence. *C++* utiliza *namespaces* para evitar conflitos de nomes e também como mecanismo de expressar um grupamento lógico (Straustrup 2013).

O *Cocos2d-x* utiliza uma estrutura de árvore para representar todos os elementos na tela. Cada elemento é chamado nessa estrutura de nó (`cocos2d::Node`). Conforme ilustrado pela Figura 2, o nó que se encontra no topo da estrutura da árvore é também conhecido como nó raiz, ou somente raiz, e aqueles que se encontram na parte mais inferior da árvore, ou seja, não possuem nenhum outro nó filho são chamados de nós folha ou somente folha.

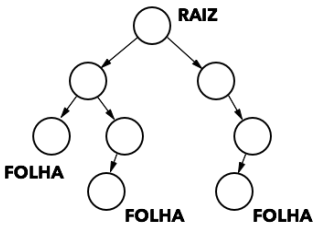

Figura 2. Árvore de nós - Fonte: do autor

Essa estrutura de árvore também é conhecida como grafo de cena (*Scene Graph*), e a ordem de desenho da tela é diretamente influenciada pela ordem de inserção dos nós na árvore e pelo algoritmo de caminhamento da árvore, que é implementado como um caminhamento em ordem.

No caminhamento em ordem, o lado à esquerda da árvore é percorrido primeiro, em seguida o nó raiz e por fim o lado direito. Como o desenho da tela é feita por passagens os últimos nós percorridos (ou seja, o lado direito da árvore) é exibida primeiro na cena.

Todos objetos que são dispostos na cena também são nós (herdam de `cocos2d::Node`). Uma camada (`cocos2d::Layer`) é um tipo específico de nó, que possui o mesmo tamanho da cena e tem como responsabilidade agrupar outros nós, normalmente com mesmas funcionalidades, e que são dispostas umas sobre as outras. A Figura 3 exemplifica a distribuição de camadas e a estrutura de árvore de uma cena.

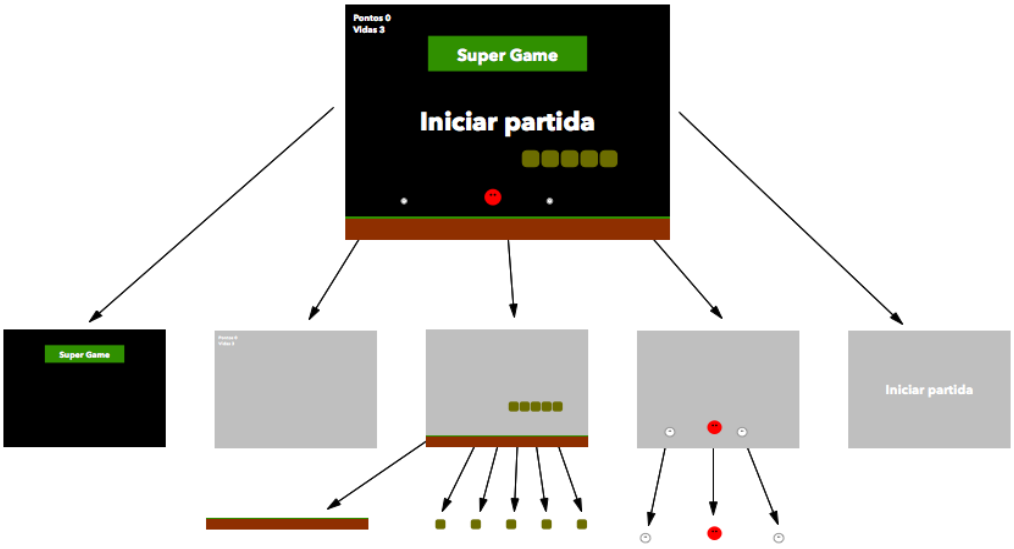

Figura 3. Árvore de nós em uma cena - Fonte: do autor

Juntamente com esses objetos descritos existem as ações (instâncias que herdam da classe `cocos2d::Action`). Uma ação é um objeto capaz de gerar uma ou várias transformações em um nó específico. Todos nós podem executar ações e como exemplos de ações temos: mover (`cocos2d::MoveTo` e `cocos2d::MoveBy`); rotacionar (`cocos2d::RotateTo` e `cocos2d::RotateBy`); modificar a escala, ou seja, aumentar ou diminuir o tamanho de um nó (`cocos2d::ScaleTo` e `Cocos2d::ScaleBy`); animar a visibilidade de um nó (`cocos2d::FadeIn` e `cocos2d::FadeOut`); modificar a cor de um nó (`cocos2d::TintTo` e `cocos2d::TintBy`) e outras.

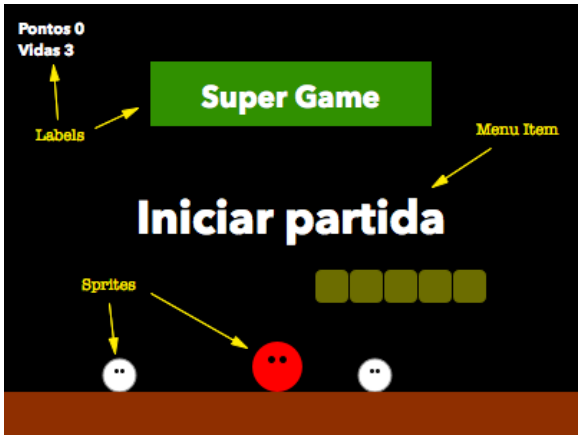

Figura 4. Tipos de nós em uma cena - Fonte: do autor

Todas as ações realizam essas transformações em um determinado intervalo de tempo, desta forma, simplificando muito o esforço do programador ao transformar as propriedades de um nó.

As ações que terminam com *By*, executa uma transformação no nó baseada em seu estado atual, seja sua posição (*Move*), cor (*Tint*), escala (*Scale*), rotação (*Rotate*), etc.

Já as ações que terminam com *To*, o estado final é o estado desejado pelo programador, simplificando, desta forma, os cálculos referentes ao estado atual. Basta informar o estado final desejado e o *Cocos2d-x* realiza as transformações a fim de atingir esse estado ao fim do intervalo de tempo especificado. A Figura 5 exemplifica as ações de *MoveBy* e *MoveTo*.

Além disto, o sistema de ações é bastante completo. Existe a possibilidade de executar diversas ações ao mesmo tempo ou executar as ações em sequência.

Por padrão, duas ações que forem configuradas para serem executadas em um mesmo nó serão executadas ao mesmo tempo. Porém, se o programador desejar ser explicito nesse comportamento, pode-se criar um objeto do tipo `cocos2d::Spawn` e configurar quais ações serão executadas ao mesmo tempo.

Para executar ações em sequência uma após o término da outra, deve se criar um objeto do tipo `cocos2d::Sequence`. Desta forma, é passada um conjunto de ações que serão executadas em sequência umas das outras.

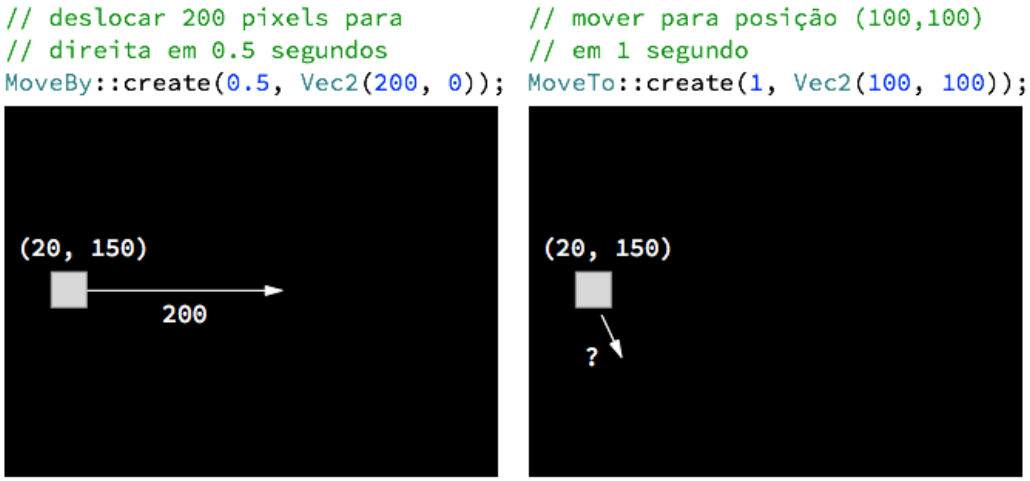

Figura 5. Ações de MoveBy e MoveTo aplicadas a um nó - Fonte: do autor

Além de existir as acões de execução paralelas e sequenciais existe também a possibilidade de facilmente criar-se ações reversas e ações de repetição. Para a primeira, basta chamar o método `reverse()` de uma ação e para o segundo basta criar um objeto do tipo `cocos2d::Repeat` (para repetir um determinado número de vezes) ou um objeto do tipo `cocos2d::RepeatForever` (para repetir indeterminadamente) e passar como parâmetro a ação a ser repetida.

4.2. Sistema de coordenadas

O *Cocos2d-x* utiliza um sistema de coordenadas cartesianas para posicionar qualquer nó na tela. O ponto de origem do sistema é o ponto à esquerda inferior da tela. O eixo *X* (horizontal) é ascendente para a direita e o eixo *Y* (vertical) é ascendente para cima. A Figura 6 mostra o sistema de coordenadas.

As coordenadas são utilizadas sempre como referência relativa a partir do nó pai. A posição de um nó nunca possui sua posição referente a tela, mas sim, referente ao nó pai que ele pertence. Também na Figura 6 podemos observar o posicionamento dos nós em relação ao elemento pai, um nó que se encontra na posição `(70, 30)` de um nó e esse último está na posição `(30, 30)` significa que na tela o nó será desenhado na posição `(100, 60)`, porém o programador nunca precisará realizar cálculos ou informar esse valor referente à tela. A esta propriedade é dado o nome de espaço do nó (*Node Space*). Uma vantagem desse comportamento é que ações realizadas em um nó pai automaticamente modificam as propriedades dos nós filhos, bastando assim executar somente uma ação para animar um conjunto de nós.

Embora esse posicionamento relativo seja um simplificador muito grande na programação, em diversos momentos é preciso realizar conversão de coordenadas entre diferentes nós na estrutura da árvore da cena. O *Cocos2d-x* provê diversas funções para auxiliar nestas conversões, tanto de um ponto do nó para o ponto na cena, quanto de um ponto na cena para um ponto específico em um determinado nó.

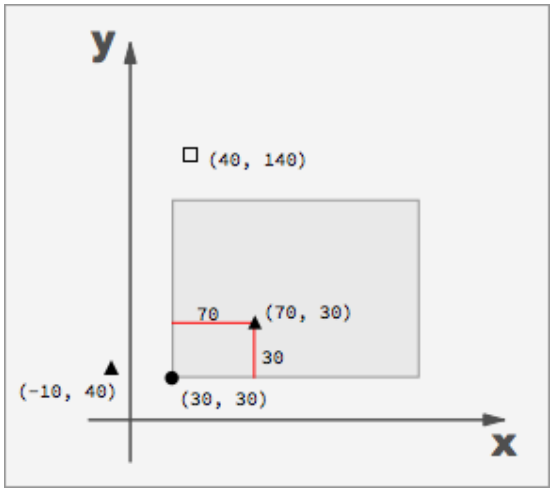

Figura 6. Sistema de coordenadas - Fonte: do autor

Além disto, para o posicionamento de nós, uma outra propriedade é utilizada para determinar qual é o ponto de referência do nó em seu posicionamento. Esta propriedade é chamada de ponto de âncora (*Anchor Point*). Esse ponto de âncora é um vetor unitário com duas coordenadas `(x, y)` e o significado de seus valores é: `x = 0`, totalmente à esquerda, `x = 1`, totalmente à direita, `y = 0`, totalmente inferior, `y = 1`, totalmente superior. Portanto, um vetor com valor `(0, 0)` se trata do ponto à esquerda inferior e `(1, 1)` é o ponto direita superior do nó. Mudanças no ponto de âncora, independente de modificar a posição do nó gera uma representação gráfica distinta umas das outras conforme podemos observar na Figura 7.

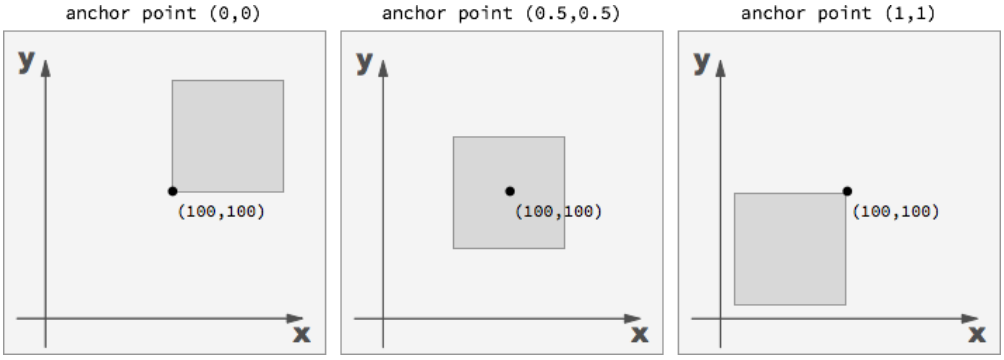

Figura 7. Modificação do ponto de âncora - Fonte: do autor

Todos os nós, exceto aqueles que representam uma imagem ou textura a ser desenhada na tela possuem por padrão o ponto de âncora com o valor `(0, 0)`. Já os outros nós que representam algo a ser desenhado na cena, os *sprites* por exemplo, possem o valor do ponto de âncora como `(0.5, 0.5)` por padrão, ou seja, o centro do nó.

4.3. Detalhes de programação com o *Cocos2d-x*

Jogos criados com o *Cocos2d-x* utilizam a linguagem *C++* como linguagem de programação, e nesta, toda a gerência de memória dos objetos criados precisa ser

controlada pelo desenvolvedor, não existe um coletor de lixo (*Garbage Collector*) como a maioria das linguagens de alto nível modernas. Portanto, todo endereço de memória deve ser cuidadosamente alocado e desalocado a fim de evitar perda de memória (*Memory Leaks*).

A fim de simplificar o gerenciamento de memória dos objetos criados, o *Cocos2d-x* utiliza o conceito de contagem de referências (*Reference Counting*). A Figura 8 exemplifica a quantidade de referências que existem para um dado objeto. Toda operação onde um objeto utiliza a referência de outro objeto e no contexto de utilização o objeto não puder ser destruído deve ser realizada uma chamada ao método `retain()` (aumentando assim a quantidade de referências para este objeto), e, logo após a finalização desse contexto é feita uma chamada ao método `release()` (diminuindo assim a quantidade de referências para o objeto).

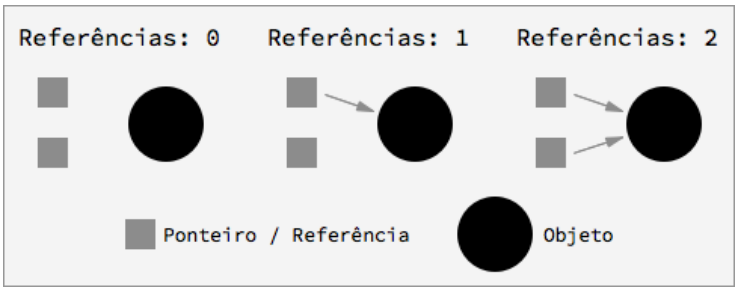

Figura 8. Contagem de referências - Fonte: do autor

Utilizando a contagem de referências, quando um objeto chegar a ter a contagem com o valor `0` (zero), o seu método destrutor é chamado e consequentemente a memória por ele alocada é liberada.

A fim de utilizar o conceito de contagem de referências todos os objetos devem herdar do objeto `cocos2d::Ref`. Além disto, o *Cocos2d-x* provê coleções que encapsulam as coleções básicas do *C++* e utiliza nelas a contagem de referências. Desta forma, sempre que um objeto for adicionado a uma coleção, a contagem de referências para este objeto irá incrementar e sempre que um objeto for removido a contagem será decrementada. Para utilizar desse comportamento automaticamente, ao invés de utilizar as coleções providas pela biblioteca padrão do *C++* é necessário utilizar as coleções providas pelo *Cocos2d-x*: o `cocos2d::Vector` (que representa uma lista de objetos) e o `cocos2d::Map` (que representa um dicionário de objetos).

4.4. Resolução de tela e resolução de design

Embora cada dispositivo possua uma diferente resolução de tela, uma diferente densidade de *pixels* e também um diferente proporção de aspecto de tela a utilização do *Cocos2d-x* para a criação de jogos simplifica esse gerenciamento.

É recomendado que uma resolução seja escolhida para o *design* do jogo, em geral `480` por `320` *pixels*. Porém, se todas as imagens do jogo forem fornecidas nessas resoluções, dispositivos com uma alta resolução e alta densidade de *pixels* apresentará o conteúdo de uma forma deformada. Para solucionar esse problema é utilizado o mesmo padrão existente para a criação de aplicações nativas para *iOS* ou *Android*, uma mesma

imagem é fornecida em diversas resoluções.

Portanto, a base do *design* do jogo é essa resolução escolhida, mas, para os dispositivos maiores, as imagens finais geradas serão maiores, porém, seguindo a mesma proporção. Um exemplo para isto seria: criar imagens com o tamanho 1x, 2x e 3x. Nas quais: 1x é a resolução como base `480x320` e focada nos dispositivos com uma menor resolução, 2x é o dobro da resolução base, portanto, `960x640`, e o foco são os dispositivos com uma resolução média, e 3x é o tripo da resolução base, portanto `1440x960`, e o foco são os dispositivos com uma resolução alta.

Desta forma, ao se criar uma imagem que no *design* tem um tamanho `100x100` deve-se criar 3 versões da mesma. Uma `100x100`, outra `200x200` e outra `300x300`. O *Cocos2d-x* utilizará aquela que melhor se enquadrar na resolução do dispositivo que o usuário estiver utilizando.

5. Estudo de caso

Masmorra de Dados é um jogo de tabuleiro, criado em 2014 pelos *designers*: Daniel Alves, Patrick Matheus e Eurico Cunha Neto. O jogo foi financiado através de um projeto de financiamento coletivo no site Catarse, e o valor arrecadado pelo financiamento superou R$ 242 mil, sendo reportado pelo próprio Catarse (2014) como um recorde em valor arrecadado para projetos de jogos não eletrônicos no Brasil.

O Masmorra de Dados é um jogo de rolagem de dados, que simula a exploração de um calabouço, tais como jogos de personificação (do inglês, *Role Playing Game* ou RPG), nos quais, os jogadores exploram diversos níveis contendo armadilhas, tesouros e principalmente monstros. O jogo se evolui em turnos, nos quais, cada jogador realiza suas ações um após o outro e por fim há o turno do inimigo, onde os monstros se movimentam em direção aos jogadores e novos monstros nascem. Além disto, o jogo permite que os dados sejam rolados novamente até 3 vezes a fim de conseguir a face desejada pelo jogador. As ações que podem ser realizadas no turno são: exploração de novas salas, efetuar combate, desarmar armadilhas e abrir baús.

O presente trabalho irá implementar um jogo eletrônico baseado no Masmorra de Dados. Sua escolha se deu para permitir que o foco do trabalho seja exclusivo no desenvolvimento da versão eletrônica, evitando desta forma, a criação de tarefas relacionadas a outras disciplinas, como a definição dos objetivos, regras e mecanismos do jogo, ou seja, o *game design* do jogo.

5.1. Regras e objetivos principais

O objetivo principal é explorar o maior número de níveis (andares) possíveis e também, eliminar a maior quantidade de monstros, conseguindo assim mais pontos de experiência.

Cada ação que o jogador pode realizar é retratada através de uma face de um dado de seis faces. As ações possíveis são: explorar (bota), defender (escudo), atacar corpo a corpo (espada), atacar a distância (arco e flecha), curar (cruz) e modificar qualquer outro dado (magia). De acordo com as faces sorteadas pelos dados no turno do

jogador ele deve tomar decisões com objetivo de otimizar a exploração da masmorra.

A exploração se dá de forma infinita, com cada andar maior que o anterior e o jogo termina quando o jogador é eliminado durante o combate. Só então sua pontuação final é calculada e armazenada para as próximas partidas.

5.2. Detalhes de implementação

O jogo inicia com a cena de introdução (`IntroScene`) que é responsável por carregar os principais recursos do jogo. Após o carregamento dos recursos é iniciada a cena de jogo (`GameplayScene`).

No momento de inicialização da cena de jogo é realizada a criação dos objetos que representam o modelo de uma partida. A Figura 9 descreve a relação entre os principais objetos. A classe `Game` é o objeto principal da hierarquia. Ela contém um objeto do tipo `Dungeon`, que por sua vez contém um conjunto de `DungeonRoom` que podem ser de diferentes tipos, tais como: `InitialRoom`, `RuneRoom`, `MonsterRoom`, `TreasureRoom` ou `DownstairsRoom`.

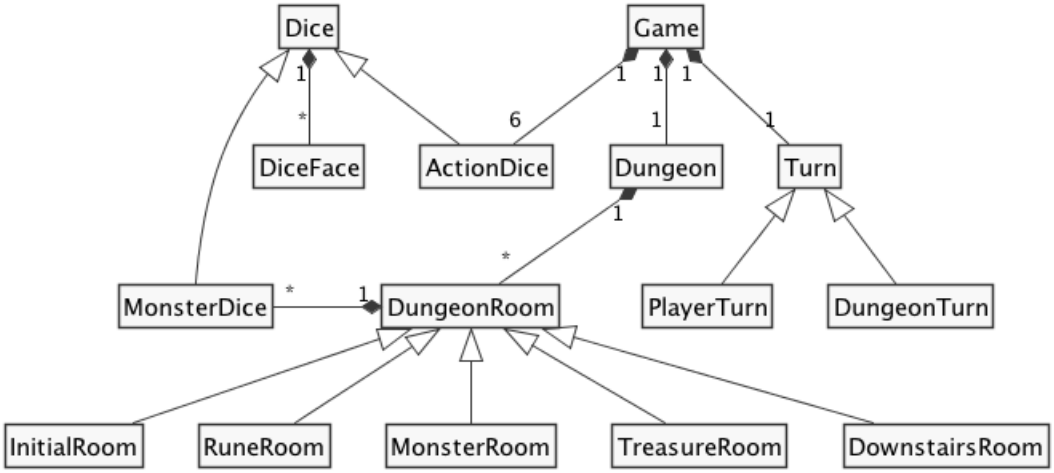

Figura 9. Classes que representam o modelo do jogo - Fonte: do autor

O objecto `Game` é implementado como um `Singleton` e este representa o estado de uma partida. O `Game` contém uma `Dungeon`, e esta representa o estado de um andar da masmorra. A principal responsabilidade da `Dungeon` é armazenar as salas (`DungeonRoom`) e também prover métodos responsáveis pela geração e posicionamento de novas salas e também de calcular a distância de cada sala para o jogador. Cada `DungeonRoom` pode conter um conjunto de monstros (`MonsterDice`).

Além da estrutura de objetos que representam o estado do jogo existem os objetos que representam a interface. A Figura 10 é uma captura de imagem do jogo logo após a inicialização da cena de jogo.

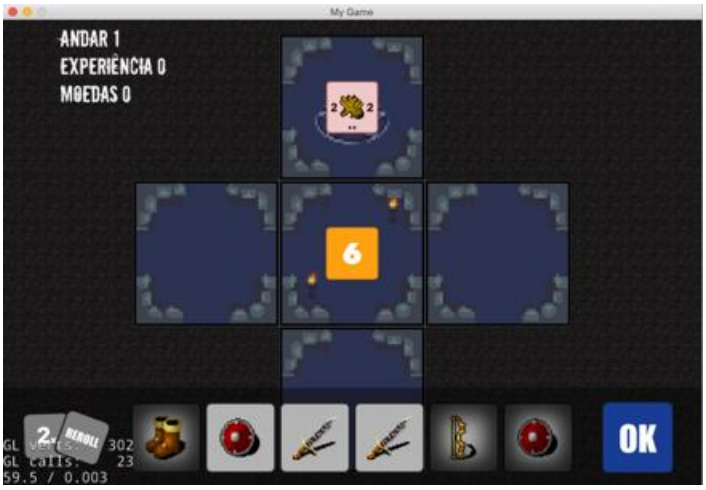

Figura 10. Captura de imagem da cena logo após inicialização - Fonte: do autor

Uma cena é composta de diversos nós. O objeto que representa a cena é a raiz no grafo de cena e todos outros objetos são adicionados a ela como filhos, a Figura 11 ilustra o grafo de cena de jogo logo após sua inicialização.

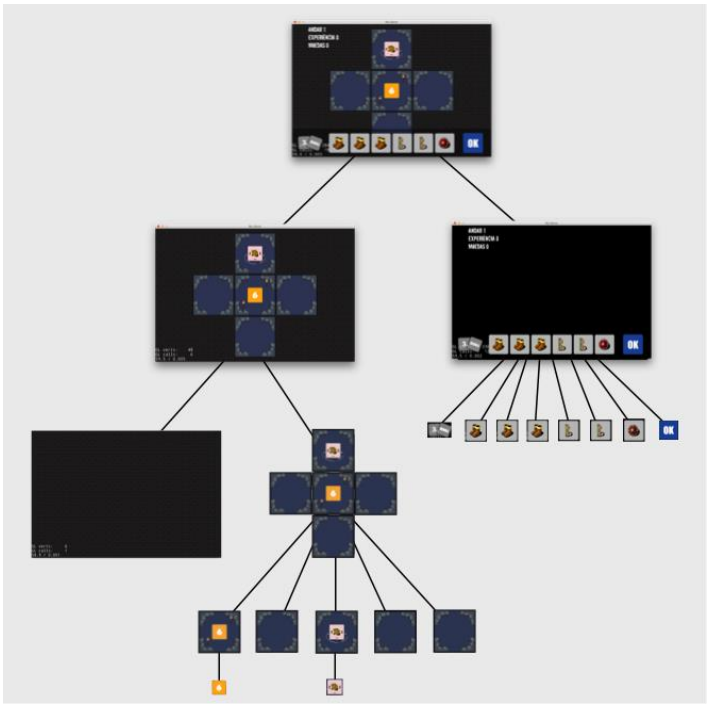

Figura 11. Grafo da cena de jogo - Fonte: do autor

Foram criados classes e objetos que encapsulam esses nós, portanto, além de ter um conjunto de classes representando o modelo de dados, temos também um conjunto de classes representando a interface, que pode ser ilustrada pela Figura 12.

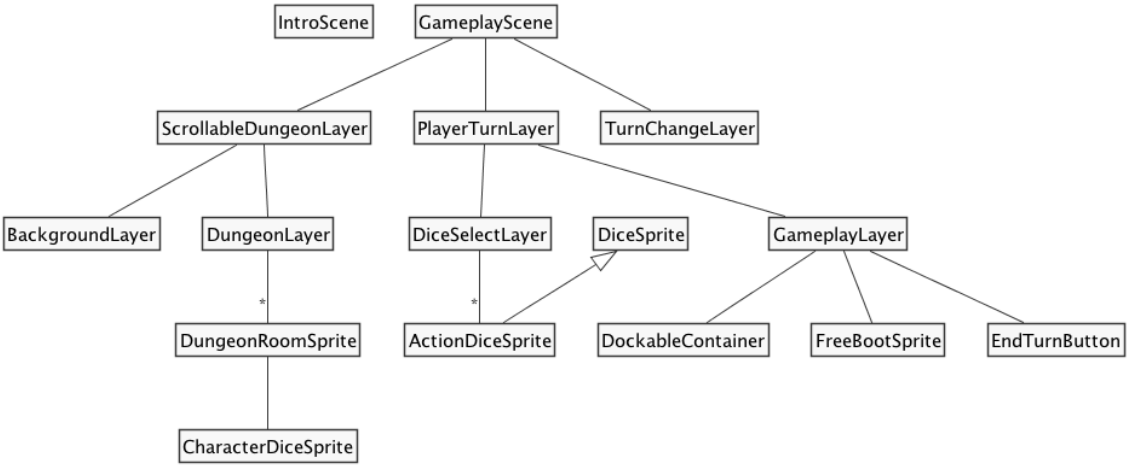

Figura 12. Classes que representam a interface do jogo - Fonte: do autor

A cena de jogo contém três camadas, a mais inferior é a `ScrollableDungeonLayer`, que representa as salas e uma textura de fundo. Como a cena pode ser maior do que o tamanho visível na tela, esta classe que é responsável por criar o conceito de câmera e permitir que o usuário interaja com a tela e consiga fazer rolagem na tela para poder visualizar as salas por completo. A camada intermediária é a `PlayerTurnLayer`, que representa todos os controles ou botões que o usuário interage na tela. E a superior é a `TurnChangeLayer`, que fica invisível na maior parte do tempo, porém é utilizada em toda troca de turno para informar que é a vez do jogador, a Figura 13 exemplifica essa camada.

Figura 13. Informação de mudança de turno - Fonte: do autor

O turno do jogador é dividido em duas partes, a primeira consiste na rolagem de seis dados de ação, e a segunda na utilização dos poderes que foram sorteados por cada dado. Para as ações não serem totalmente aleatórias os dados podem ser rolados novamente até três vezes. A Figura 14 ilustra os passos para a rolagem dos dados.

Figura 14. Rolagem de dados- Fonte: do autor

As salas são dispostas em um formato de grade. Cada nova sala explorada pelo jogador dispara a geração de novas salas adjacentes ainda não geradas. A Figura 15 exemplifica o processo de animação das salas adjacentes sendo posicionadas.

Figura 15. Posicionamento de novas salas - Fonte: do autor

Modificações no estado do jogo devem refletir em mudanças na interface, porém, essas mudanças devem ser animadas a fim de evitar uma experiência do usuário ruim. Para estas animações são utilizadas as funções de ações do *Cocos2d-x*. Embora estas funções sejam completas, uma dificuldade encontrada durante o desenvolvimento foi controlar o tempo de execução das animações, pois, enquanto uma animação está acontecendo, o estado do jogo pode ter sofrido modificações e consequentemente invalidando as animações anteriores e necessitando um nova representação de interface.

Outro ponto de implementação que apresentamos é a estrutura utilizada para armazenar as salas. As salas possuem uma representação na interface em grade e a utilização de uma estrutura de dados bidimensional, como uma matriz, seria a solução mais simples. Porém, a utilização de vetor ou matriz necessitaria pré alocar a memória necessária para todas as salas que fossem ser posicionadas, e como não é possível prever para quais direções o jogador explorará a masmorra, o tamanho da estrutura de dados iria ser muito grande desnecessariamente. A fim de otimizar o uso de memória a

estrutura foi achatada para somente uma dimensão e foi implementada utilizando um dicionário (`cocos2d::Map`), no qual a chave de acesso é inteiro que agrupa o valor do eixo X com o valor do eixo Y conforme ilustrado pela Figura 16.

```
int TAMANHO_DA_MASMORRA = 100;
int x = 49;
int y = 50;

// representação padrão com 2 dimensões
rooms[x][y] = nova sala

// representação utilizada achatada
indice = x * TAMANHO_DA_MASMORRA + y; // 49 * 100 + 50
rooms[indice] = nova sala             // rooms[4950] = nova sala
```

Figura 16. Achatamento da estrutura de dados - Fonte: do autor

Durante o turno adversário, todos os monstros devem se movimentar para uma sala adjacente mais próxima ao jogador. Para isto, é necessário realizar o cálculo da distância das salas até a sala onde o jogador se encontra. Este cálculo foi implementando após cada movimentação do jogador utilizando um algoritmo recursivo de cálculo de distância. Partindo da sala onde o jogador está, que possui uma distância zero, o algoritmo percorre as salas adjacentes configurando nestas salas a distância como sendo a distância da sala anterior acrescido de um. A execução do algoritmo é ilustrado pela Figura 17. Desta forma, a movimentação dos monstros se tornou mais simples, para cada sala onde contém um monstro basta verificar entre as salas adjacentes qual possui a menor distância para o jogador.

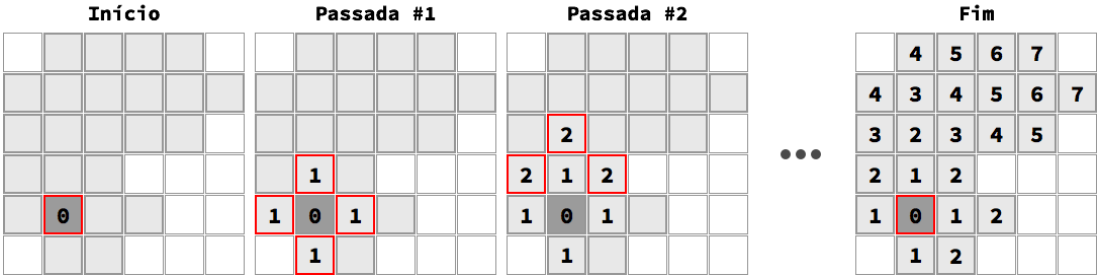

Figura 17. Algoritmo de cálculo de distância - Fonte: do autor

Para evitar que objetos de modelo sejam acoplados a objetos de interface a comunicação entre eles é feita de forma assíncrona utilizando eventos. O *Cocos2d-x* provê esse mecanismo de eventos através do objeto `cocos2d::EventDispatcher`. Todos objetos que se interessam por um tipo de mensagem são registrados como ouvintes (*listeners*) de uma mensagem. Mudanças no estado do modelo de dados ou interações na interface podem disparar esses eventos. Para cada evento gerado, o `EventDispatcher` notifica todos ouvintes interessados nessa mensagem. A Figura 18 exemplifica através de um diagrama de sequência um evento disparado pela `Dungeon` informando que salas foram posicionadas, por sua vez a classe `DungeonLayer` recebe essa mensagem e posiciona as salas, logo após finalizar a animação de posicionamento da última sala ela dispara um outro evento diferente que outros ouvintes podem escutar.

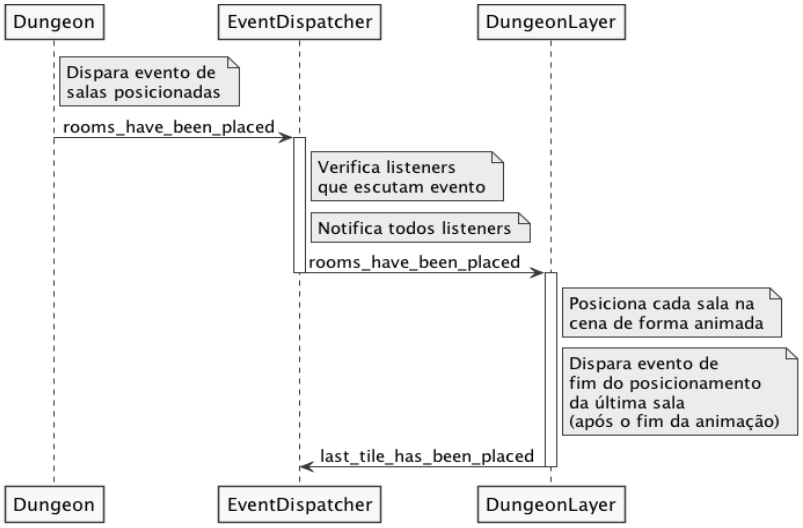

Figura 18. Troca de mensagens utilizando o `EventDispatcher` - Fonte: do autor

5.3. Distribuição do jogo para múltiplas plataformas

O *Cocos2d-x* provê uma ferramenta de linha de comando que é utilizada tanto para criar o projeto, quanto para efetuar a compilação do mesmo para cada plataforma. Durante o processo de criação do projeto é criado também arquivos de projetos para as principais plataformas possíveis de distribuir o jogo, são elas: *iOS*, *Mac OS X*, *Android*, *Linux*, *Windows* (32 *bits*) e *Windows* 8.1 universal e *Windows Phone*.

Embora não seja necessário realizar nenhuma modificação no código escrito para que o jogo funcione, em cada uma das plataformas é necessário que o ambiente de compilação tenha todos os pré-requisitos necessários para compilar uma aplicação específic. A Figura 19 ilustra a mesma aplicação sendo executada em diferentes plataformas.

Figura 19. Execução da aplicação em múltiplas plataformas - Fonte: do autor

6. Considerações Finais

Devido a grande quantidade de plataformas existentes, a utilização de uma ferramenta que permita a criação de jogos multiplataforma se faz necessária para projetos com alta produtividade. Este trabalho verificou a utilização do *Cocos2d-x* como ferramenta para a criação de jogos multiplataforma. Apresentamos os principais conceitos de sua arquitetura e operações necessárias para o projeto de um jogo. Como

estudo de caso, apresentamos um jogo baseado no jogo de tabuleiro Masmorra de Dados totalmente desenvolvida com a ferramenta em questão. Com o mesmo código fonte, não foi necessário realizar modificações para executar em diferentes plataformas, evidenciando portanto, a redução do esforço para produzir um mesmo jogo para várias plataformas.

O processo de geração do jogo para cada plataforma alvo necessita que o ambiente de programação da mesma esteja configurado e, também, é necessário um conhecimento prévio por parte do programador sobre como gerar aplicativos para a mesma. O *Cocos2d-x* não introduz nenhum aspecto dificultador ao processo de compilação para as plataformas.

Além disto, as funções da ferramenta são estáveis, intuitivas e bem documentadas. Entretanto, ainda é pequena a quantidade de conteúdo de terceiros para a criação de jogos utilizando o *Cocos2d-x*. O grande envolvimento de empresas chinesas em seu desenvolvimento e uso, implica que a maior parte do material disponível na Internet é em língua chinesa. O que impõe uma barreira para desenvolvedores que não dominam a língua.

O presente trabalho não explorou todas as funcionalidades presentes em diversos jogos, como exemplo interação multi jogador. Como melhoria direta, a inclusão deste modo significaria um grande avanço no jogo de forma a deixá-lo com mais modos de jogo e promover a interação dos jogadores via rede de dados.

Uma outra melhoria desejável, mas que ficou fora do escopo do trabalho é permitir o compartilhamento do estado do jogo entre diferentes dispositivos do mesmo jogador. Assim seria possível sincronizar os dados através de um servidor e o jogador poder começar uma partida em um *smartphone* e continuá-la, por exemplo em um *tablet, smart tv* ou mesmo um *desktop.*

O presente trabalho abordou o desenvolvimento de jogos multiplataforma com foco para computadores e dispositivos móveis. Um futuro trabalho poderia abordar a geração de jogos multiplataforma com foco nos principais videogames do mercado.

Referências

Adams, E. (2009). “Fundamentals of Game Design”, New Riders, 2nd Edition.

AppAnnie. Ranking de aplicativos gratuitos, pagos e mais rentáveis no Google Play nos Estados Unidos. Disponível em < https://www.appannie.com/apps/ios/top/?device=iphone>. Acesso em 7 jun. de 2015.

_____. Ranking de aplicativos gratuitos, pagos e mais rentáveis para na App Store nos Estados Unidos. Disponível em <https://www.appannie.com/apps/google-play/top/united-states/>. Acesso em 7 jun. de 2015.

Bates, B. “Game Design”, Thomson Course Technology, 2nd Edition.

Catarse. “Retrospective Dois Mil e Catarse: R$ 1 milhão por mês”. Disponível em <http://blog.catarse.me/retrospectiva-dois-mil-e-catarse-r-1-milhao-por-mes/>. Acesso em 11 jun. de 2015.

Corona Labs. “Corona SDK | Corona Labs”. Disponível em <https://coronalabs.com/products/corona-sdk/>. Acesso em 11 jun. de 2015.

Gartner. “Gartner Says Smartphone Sales Surpassed One Billion Units in 2014”. Disponível em <http://www.gartner.com/newsroom/id/2996817>. Acesso em 11 jun. de 2015.

Gaudiosi, J, Fortune. “Mobile game revenues set to overtake console games in 2015”. Disponível em <http://fortune.com/2015/01/15/mobile-console-game-revenues-2015/>. Acesso em 7 jun. de 2015.

GitHub. “cocos2d-x”. Disponível em <https://github.com/cocos2d/cocos2d-x>. Acesso em 03 maio de 2015.

Marmalade SDK. “Marmalade C++ SDK”. Disponível em <https://www.madewithmarmalade.com/products/marmalade-sdk>. Acesso em: 5 jun. de 2015.

Mirani, L. “Apple is overwhelmingly reliant on games for App Store revenue”. Disponível em <http://qz.com/309715/apple-is-overwhelmingly-reliant-on-games-for-app-store-revenue/>. Acesso em: 3 jun. de 2015.

Needleman E. Sarah. “Mobile-Games Revenue Growth Is Outpacing Other Content, for Now”. Disponível em <http://blogs.wsj.com/digits/2015/02/18/mobile-games-revenue-growth-is-outpacing-other-content-for-now/>. Acesso em 11 jun. de 2015.

Rogers, R. (2011) “Learning Android Game Programming”, A Hands-On Guide to Building Your First Android Game, Addison-Wesley.

Salen, K. e Zimmerman, E. (2012) “Regras do jogo: fundamentos do design de jogos: principais conceitos: volume 1”, [tradução Edson Furmankiewicz]. - São Paulo: Blucher.

Straustrup, B. (2013). “The C++ Programming Language”, Addison-Wesley, 4th Edition.

VentureBeat. “Gaming's top 25 public companies generated $54.1B in revenue last year”. Disponível em <http://venturebeat.com/2015/04/20/gamings-top-25-public-companies-generated-54-1-billion-in-revenue-last-year/>. Acesso em 11 jun. de 2015.

Unity 3D. “Unity 3D - Game Engine”. Disponível em <https://unity3d.com>. Acesso em 14 jun. de 2015.

Unreal Engine. “What is Unreal Engine 4”. Disponível em <https://www.unrealengine.com>. Acesso em 10 jun. de 2015.

Zechner, M. e Green, R. (2011) “Beginning Android 4 Games Development”, Apress.