UNIVERSIDADE FEDERAL DO ABC

JULIANO BORGES DA SILVA GOMES

Desenvolvimento de um motor de jogos multiplataforma

SANTO ANDRÉ

2015

UNIVERSIDADE FEDERAL DO ABC

JULIANO BORGES DA SILVA GOMES

Desenvolvimento de um motor de jogos multiplataforma

Monografia apresentada à Universidade Federal do ABC como parte dos requisitos para aprovação na disciplina de Trabalho de Conclusão de Curso do Curso de Bacharelado em Ciências da Computação.

SANTO ANDRÉ

2015

JULIANO BORGES DA SILVA GOMES

Desenvolvimento de um motor de jogos multiplataforma

Monografia apresentada à Universidade Federal do ABC como parte dos requisitos para aprovação na disciplina de Trabalho de Conclusão de Curso do Curso de Bacharelado em Ciências da Computação.

Santo André – SP, 05 de agosto de 2015

COMISSÃO EXAMINADORA

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Professor:

Instituição:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Professor:

Instituição:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Professor:

Instituição:

Resumo

No mercado de desenvolvimento de jogos, as plataformas que auxiliam e aceleram o desenvolvimento estão cada vez mais populares. A maior parte delas tenta, principalmente, facilitar a portabilidade para diversas plataformas. Entretanto, esse é um trabalho complexo e bastante técnico, podendo ser resolvido através de diversas abordagens possíveis. Esta monografia narra as dificuldades encontradas e as soluções utilizadas, mostrando que, apesar do que foi feito, ainda é muito difícil criar um jogo, e que ferramentas do mercado facilitam cada vez mais essa atividade.

PALAVRAS-CHAVE: JOGOS, MOTORES, MULTIPLATAFORMA, C++, SDL, JOGOS 2D

ABSTRACT

In the game development market, the tools that helps and speeds up the development are turning out even more popular. Most of them tries mainly to easy the cross-platform deployment. This monograph narrates the problems and solutions found in the course of development of a similar tool, but simplified. The job turned out to be complex and too much technical, capable to solve through many means. Showing that spite of what have been made. It still too hard to make a game, and that, tools in the market helps each day even more this activity.

KEYWORDS: GAMES, CROSS-PLATFORM, ENGINES, C++, SDL, 2D GAMES

Lista de imagens

Figura 1 Editor visual da unity 14

Figura 2 Editor visual da unreal engine 15

Figura 3 ARQUITETURA DO FRAMEWORK 24

Figura 4 Diagrama de Classes dos principais componentes da Game Engine 25

Figura 5 Representação gráfica da estrutura de uma cena 27

Figura 6 Resumo da arquitetura principal 31

Figura 7 exemplo de spritesheet 39

Figura 8 CÓDIGO PARA A CRIAÇÃO DAS CENAS 45

Figura 9 Menu inicial 46

Figura 10 O jogo 47

Figura 11 Fim de jogo 48

Sumário

[Resumo iii](#_Toc435309597)

[ABSTRACT iv](#_Toc435309598)

[Lista de imagens v](#_Toc435309599)

[Sumário vi](#_Toc435309600)

[1 Introdução 8](#_Toc435309601)

[1.1 O que é um motor de jogos (*gaming engine*) 8](#_Toc435309602)

[1.2 Justificativa 10](#_Toc435309603)

[1.3 Objetivo 11](#_Toc435309604)

[1.4 Organização do Documento 11](#_Toc435309605)

[2 Materiais e Método 13](#_Toc435309606)

[2.1 Unity 5 13](#_Toc435309607)

[2.2 Unreal engine 14](#_Toc435309608)

[2.3 Cocos 2d-x 16](#_Toc435309609)

[2.4 Multiplataforma 17](#_Toc435309610)

[2.5 *Frameworks* para jogos 18](#_Toc435309611)

[2.5.1 Biblioteca LibSDL para C++ 20](#_Toc435309612)

[2.6 Linguagem 20](#_Toc435309613)

[2.7 Criação de projetos 21](#_Toc435309614)

[3 A Game Engine 23](#_Toc435309615)

[3.1 Arquitetura 23](#_Toc435309616)

[3.1.1 Atributos da aplicação 26](#_Toc435309617)

[3.1.2 Cenas 27](#_Toc435309618)

[*3.1.3* *GameObjects* 28](#_Toc435309619)

[*3.1.4* *Behaviours* 29](#_Toc435309620)

[*3.1.5* *Observers* 30](#_Toc435309621)

[3.2 Criando um novo projeto 31](#_Toc435309622)

[3.3 Recursos 32](#_Toc435309623)

[3.3.1 Aplicação 32](#_Toc435309624)

[3.3.2 Cenas 33](#_Toc435309625)

[3.3.3 Vetores 34](#_Toc435309626)

[3.3.4 Objetos de Jogo 35](#_Toc435309627)

[3.3.5 Câmera 36](#_Toc435309628)

[3.3.6 Texto 37](#_Toc435309629)

[3.3.7 Sprites 37](#_Toc435309630)

[3.3.8 Comportamento 39](#_Toc435309631)

[3.3.9 Observadores 40](#_Toc435309632)

[3.3.10 Controles 41](#_Toc435309633)

[3.3.11 Áudio 42](#_Toc435309634)

[3.3.12 Tempo 44](#_Toc435309635)

[4 Estudo de caso 45](#_Toc435309636)

[4.1 Tela de inicio 46](#_Toc435309637)

[4.2 O jogo 47](#_Toc435309638)

[4.3 Fim de jogo 48](#_Toc435309639)

[4.4 Demonstração 49](#_Toc435309640)

[5 Conclusão 50](#_Toc435309641)

[Referências 51](#_Toc435309642)

# Introdução

Motor de jogos, conhecido popularmente com o seu termo em inglês, *engine*, é um conjunto de ferramentas ou *frameworks* que facilitam a criação de um jogo, como por exemplo um conjunto de ferramentas que variam desde o gerador de imagens, um tocador de som, entre outros.

Os motores de jogos, atualmente, são amplamente utilizados e não apenas em jogos, mas também para desenvolvimento de produtos na área de educação, arquitetura e engenharia. Seus custos têm se tornando bastante acessíveis e as ferramentas mais populares são gratuitas, apesar de possuírem limitações em sua versão gratuita, essas limitações variam de acordo com a ferramenta e podem ter recursos, *royalties* entre outros. (Boyer, 2007).

Quando se trabalha com desenvolvimento de aplicações, é necessário pensar, hoje em dia, na existência de diversas plataformas, pois temos um mercado cada vez maior de computadores móveis e pessoais, os quais possuem plataformas de desenvolvimento distintas e muitas vezes nos deparamos com a necessidade de reescrever todo o código em várias linguagens e ferramentas. As ferramentas específicas tentam reverter esse problema, tornando cada vez mais simples o desenvolvimento de um produto em diversas plataformas.

Esse modelo de produção, multiplataforma, é amplamente adotado em ferramentas de desenvolvimento de jogos, além do uso de novas tecnologias como *html 5* entre outras bibliotecas que se atualizam constantemente para se manter no mercado. Há, portanto, uma necessidade cada vez mais urgente de criar novos sistemas que possam ser portados para todas as plataformas, ou seja, uma ferramenta multiplataforma, do inglês *cross-platform*.

## O que é um motor de jogos (*gaming engine*)

Motor de jogos é a junção de programas que determinam o cenário, as interações físicas, gráficas e sonoras. É comumente encontrado de duas formas diferentes: quando o motor é feito apenas para o jogo e é adaptado integralmente para aquele único jogo, podendo apenas parte dele ser aproveitado para outros jogos; ou quando existe um conjunto ou apenas um programa, bem formado e estruturado, para ser usado no jogo. Neste último caso, grande parte das interações é abstraída, portanto, vários jogos poderão ser produzidos a partir desse motor, sendo ele o mais genérico possível, além de dar liberdade ao criador do jogo para que se trabalhe objetivamente com o jogo sem se preocupar com o seu motor. (Ward, 2008).

Uma das ferramentas mais tradicionais do mercado a *Unreal Engine* (Unreal Engine 4, 2015), uma ferramenta que se tornou popular entre vários tipos de empresas, além de possuir em seu portfólio uma grande quantidade de jogos muito bem aceitos pela crítica. É uma ferramenta sólida e que sempre se inova no mercado. Possui, por exemplo, em seu portfólio, jogos como o clássico *Unreal* (Thomsen, 2010)e o popular *Gears of War* (Butts, 2007), que por muito tempo tornaram-se *cases* do *showcase* do *Xbox360* (Thorsen, 2004) pela *Microsoft*.

No site próprio site da Unreal Engine, é possível encontrar uma definição simples do que é uma *engine*:

“*Unreal Engine 4 é um conjunto completo para o desenvolvimento de jogos felto por desenvolvedores de jogos. De jogos móveis 2D até jogos “arrasa quarteirão” e em realidade virtual, Unreal Engine 4 te dá tudo para desenvolver, entregar, crescer e se destacar da multidão*.” [[1]](#footnote-1)

Atualmente, o *Unreal Engine* é gratuito e, em caso de vendas acima de 30 mil doláres,5% dos *royalties*.

Ainda em um mercado extremamente competitivo, temos a *Unity3D* (UNITY3D, 2015), ferramenta difundida entre os desenvolvedores independentes e gratuita, com determinadas limitações que variam de acordo com suas versões. É largamente utilizada em mercados pequenos e emergentes do setor de desenvolvimento. O Brasil, que tem uma quantia significante de usuários, fez com que a empresa sediasse eventos no país.

*Unity 3D* vêm crescendo paulatinamente e hoje domina o mercado de pequenos jogos. Seus jogos podem ser portados para as a maior parte das plataformas utilizadas e com mínimo esforço. O modo de usar é simples e exige um conhecimento bem pequeno da área técnica, podendo um programador iniciante ser capaz de desenvolver um jogo inteiro.

Existem dezenas de outras ferramentas que tentam sobreviver a este mercado e algumas sendo criadas individualmente pelas empresas que desenvolvem jogos. Vale mencionar, neste caso, a ferramenta criada pela *Ubisoft*, conhecida como *Ubi-art[[2]](#footnote-2)*, onde nos últimos anos tem revolucionado o campo de desenvolvimento de jogos em duas dimensões.

Algumas pequenas empresas também disponibilizam ferramentas e muitas vezes de forma gratuita e de código aberto, como é o caso da engine *GODOT* (GODOT, 2014).

## Justificativa

O desenvolvimento de jogos no Brasil é bastante limitado, sendo o conhecimento também muito superficial. Poucas empresas possuem estrutura para desenvolver jogos de qualidade e como potencial, temos um mercado novo e em expansão acelerada. Nossa condição atual é próxima de todos os desenvolvedores independentes do mundo e há cada vez mais um incentivo e investimento para a qualificação de profissionais nesta área.

Porém, sempre há algo novo em que uma ferramenta possa acrescer e possivelmente fomentar no mercado, não só apenas consumidor no país, mas também um mercado desenvolvedor de jogos e sistemas robustos, que atualmente é um grande consumidor dessas ferramentas.

## Objetivo

Este trabalho tem como objetivo codificar uma ferramenta de código aberto que possa explorar o desenvolvimento de jogos e acrescentar simplicidade ao desenvolvimento de jogos em duas dimensões (2D), evitando assim a criação de vários códigos distintos para cada plataforma e acelerando o desenvolvimento de jogos. Uma ferramenta leve, prática e objetiva.

Para tal serão estudadas as principais *Gaming Engines* do mercado, sua estrutura e recursos, bem como técnicas para organização de código. Estas técnicas visam aumentar a produtividade do programador que irá desenvolver o jogo e também permitir altas taxas de manutenção de código.

Para acesso, será disponibilizado no repositório publico de Código GitHub, por ser uma iniciativa de código aberto, seu conteúdo será disponibilizado em um repositório gratuito e acessível por quem quiser utilizá-la e modificá-la, obviamente respeitando os direitos autorais e mencionando os autores do projeto.

## Organização do Documento

O documento está dividido em 5 capítulos, sendo a primeira, introdução. O segundo capitulo, como desenvolvimento, aborda a bibliografia utilizada para a criação do projeto, fazendo referência às tecnologias e todo arcabouço teórico necessário para o desenvolvimento do projeto.

No terceiro capitulo é abordada a ferramenta propriamente dita, sua arquitetura, exemplos de código e como utilizá-la para a produção de um jogo simples em um menor tempo.

O quarto capitulo tem como objetivo apresentar o estudo de caso: o jogo-exemplo que vem como demonstração, exibindo como a ferramenta foi utilizada para fazer um jogo simples, com um fluxo de telas, condições de vitória e derrota.

O último capitulo conclui o que foi aprendido e criado pela plataforma, além de abordar como seriam os próximos passos e projetos para a expansão e melhoria da ferramenta.

# Materiais e Método

Para a criação do projeto, iniciou-se uma pesquisa profunda nas ferramentas já existentes, pois antes de criá-la, há necessidade de explorar o que já foi criado. Essas ferramentas, como já citadas, são ferramentas com modelos de negócios únicos e produzidas por empresas bem consolidadas e grande conhecimento técnico. Segue abaixo os exemplos pesquisados.

## Unity 5

Conhecida anteriormente como *Unity3D*, é a ferramenta favorita entre as pequenas empresas e desenvolvedores independentes. A maior motivação entre os seus usuários é sua comunidade unificada e prestativa, além de uma documentação vasta e clara para o desenvolvimento, o que proporciona um início simples e agradável para qualquer desenvolvedor, seja ele iniciante ou com grande experiência.

A ferramenta inicialmente possuía suporte apenas para jogos em 3D, e com o avançar do tempo, notou-se a necessidade de expandi-la. Hoje ela possui todo o sistema 2D e uma estrutura bem sólida para suprir qualquer necessidade, tanto de uma empresa pequena, quanto uma empresa maior que queira criar um jogo mais rápido e de forma barata.

Sua estrutura para a programação é baseada em *scripts*, criados e adicionados em objetos de jogo que são construídos visualmente no editor principal. Esses *scripts* podem ser editados por qualquer editor de texto, apesar de atualmente a *Unity 5* vir com um editor de texto próprio que é uma versão extraída do antigo *monodevelop* (UNITY Documentation, 2014).

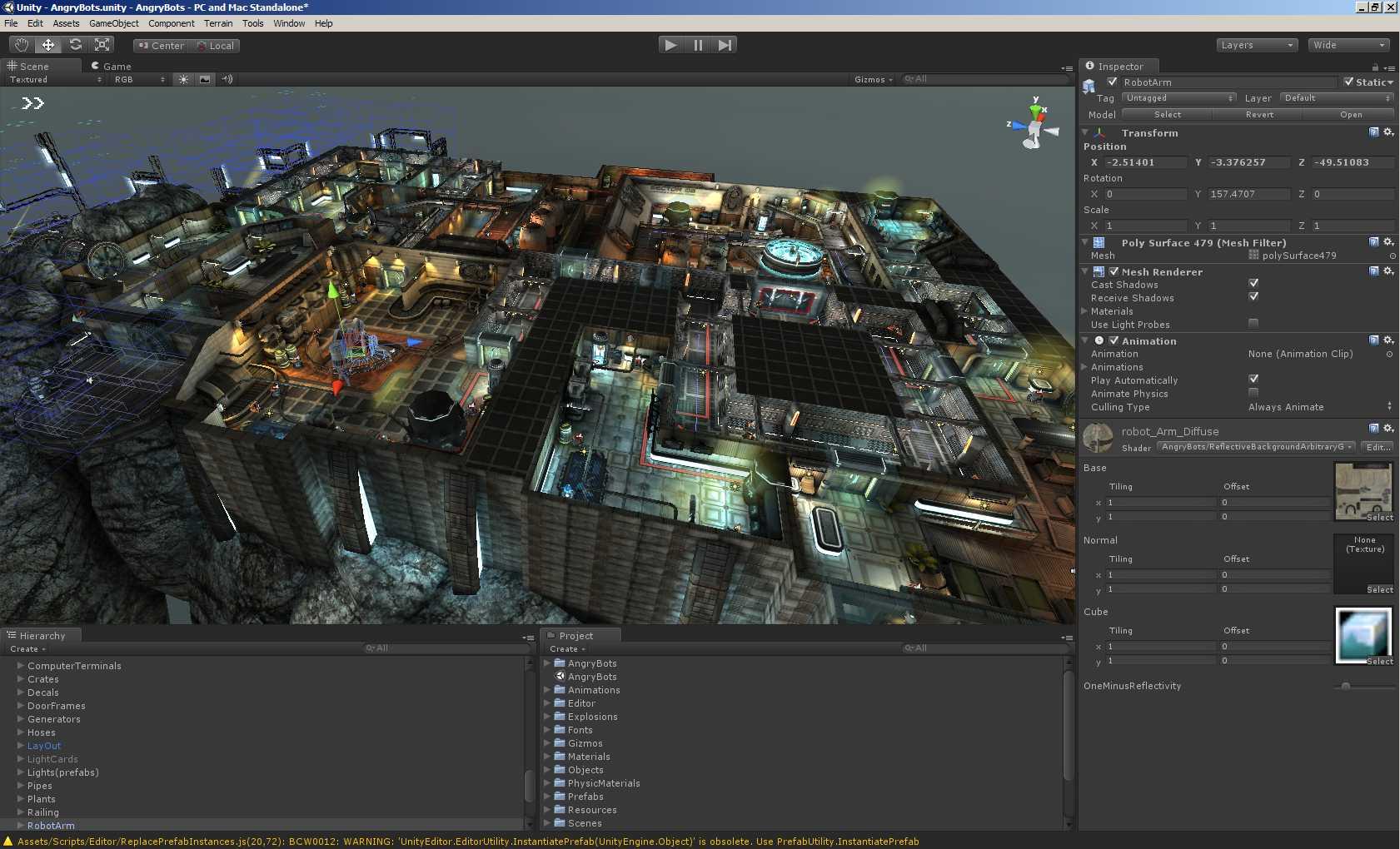


Figura 1 Editor visual da unity

Fonte: <http://bestwinsoft.com/images/Unity\_3D\_1.jpg>

Variáveis públicas podem ser atribuídas de forma visual dentro da ferramenta, portanto a necessidade de vínculo e buscas e organizações de código torna-se bem superficial. Grandes produtos podem ser produzidos sem um conhecimento extenso na área de programação e utilizando códigos prontos. Esses códigos, que são a base da ferramenta, possuem uma gama extensa de *assets* (extensões, *plugins*) produzidos pela comunidade e podem ser adquiridos de forma simples dentro da própria ferramenta.

Mesmo sendo pouco necessária a programação, programar pode ter grande efeito na ferramenta, já que menus e outras formas de facilitar a criação de um novo jogo podem ser criadas de forma prática dentro da ferramenta, no entanto, necessita-se de um conhecimento maior de programação.

## Unreal engine

Aclamada pela crítica, a Unreal Engine já foi utilizada amplamente para jogos de altíssima qualidade. O *show case* da ferramenta é impressionantemente vasto e com jogos que tiveram uma grande quantidade de venda. Em pouco tempo, ela deixou de ser uma ferramenta para grandes empresas e tornou-se uma ferramenta gratuita e de grande poder. (Makuch, 2015).

Seu código é aberto e é possível programar utilizando C++, uma linguagem mais complexa e de grande robustez. Possui integrados, um editor visual e uma loja *online* bem recente. Algo bem semelhante a ferramenta *Unity* e está claro, portanto, que ambas tornaram-se concorrentes de peso.

Figura 2 Editor visual da unreal engine

Fonte: <http://i.ytimg.com/vi/QX1\_wFWLMZU/maxresdefault.jpg>

*Unreal* está disponível apenas para a plataforma *Windows* e depende da ferramenta da *Microsoft Visual Studio* (Somasegar, 2015) um excelente editor de texto que é amplamente usado pela comunidade *Windows*. Suas estruturas são bem dividas e claras com uma documentação também sólida e clara.

Programar em *Unreal* necessita de um conhecimento técnico avançado e a curva de aprendizado pode se tornar drástica, por ser longa e difícil, para programadores inexperientes. Os resultados obtidos são impressionantes, tanto visuais quanto em velocidade e rendimento do produto.

## Cocos 2d-x

Ferramenta mais próxima do que está sendo desenvolvido o projeto e fonte de grande inspiração. *Cocos 2d-x* foi derivada de uma ferramenta *open-source*, criada para desenvolver jogos para iOS (COCOS 2D, 2015), por ser adquirida de forma gratuita e sem nenhum tipo de vínculo ao usuário, tornou-se muito utilizada. Recentemente adquiriu editores visuais e mais ferramentas que são anexas ao seu *workflow*. Esses editores ainda estão em desenvolvimento, o foco da ferramenta continua sendo o seu framework.

Porém amplamente dependente do seu gerador de códigos produzidas em *python* e com exemplos simples e de fácil aprendizado para programadores sem conhecimentos prévios em desenvolvimento de jogos. As *builds* foram criadas utilizando o sistema para builds make e possui acessos bem simples para a geração de projetos através dos seus *scripts*.

A comunidade é menor que de outras ferramentas, mas existem bons jogos produzidos, apenas para celulares. Seu objetivo não é enfrentar grandes empresas, mas sim uma ferramenta simples para aplicativos e jogos, que possuem a necessidade de ser multi-plataforma.

Possui uma interface em C++, de fácil uso e acesso e um bind, sistema que traduz a linguagem C++ para a linguagem Lua[[3]](#footnote-3), para a geração de scripts rápidos e que se encaixam facilmente para programadores sem experiência ampla.

Apesar de ser uma comunidade pequena, seus desenvolvedores dão suporte de forma rápida e possuem várias ferramentas para a comunicação, como fóruns, *twitter* e até mesmo um canal no IRC.

## Multiplataforma

Com a evolução dos computadores, surgiu a necessidade cada vez maior de diversos sistemas operacionais e com cada vez mais novos tipos de aparelhos e máquinas, capazes de processar informação de forma única. A divergência nas plataformas é algo que existe desde os primórdios da computação e hoje existem dezenas de formas e ferramentas para sanar esse problema.

Uma das formas mais populares de programação multiplataforma é a linguagem Java, que gera diretamente um arquivo executável, que poderá ser processado pela máquina virtual *JAVA (JVM)*, porém é necessário que o usuário instale a máquina virtual na máquina.

Atualmente existem diversas interfaces que possibilitam a criação de dezenas de produtos que são disponibilizados em celulares, computadores e até consoles de jogos digitais, é o popular "*Write once, deploy everywhere"*, escreva uma vez e disponibilize em todos os lugares [tradução livre].

Essas ferramentas para interface multiplataforma são bem difundidas, já que a maior parte das interações com usuários são diferenciadas entre sistemas operacionais. A forma como o *Windows* exibe a informação é bem distinta da forma do *MacOS* e ainda bem mais distinta que os aparelhos móveis. Ferramentas como *QT[[4]](#footnote-4)* e *GTK+[[5]](#footnote-5)* são bem conhecidas e conseguem atingir uma gama de plataformas extensa.

Com o surgimento da era dos aplicativos nos aparelhos móveis, paulatinamente, a produção de softwares que pudessem ser disponibilizados em todas as plataformas tornou-se necessária. Dispositivos móveis não executam máquinas virtuais e, apesar do sistema *Android -* um dos sistemas para celulares mais utilizados e conhecidos no mercado - possuir *java* em seu âmago, não era possível disponibilizar com o mesmo código algo que executasse em diversas plataformas.

Nesse ambiente, surge o *Apache Cordova[[6]](#footnote-6)* , software que resolve o problema de forma engenhosa, criando uma *api* que utiliza *html* e *javascript* para acessar as funções do celular, gerando um *micro-browser* que acessa sua aplicação como se ela fosse *web* e visualmente agradável. No entanto, isso gera desvantagens em processamento, caso seja necessário um processamento maior de informações, torna-se inviável a criação de jogos.

*Apache Cordova* é uma das ferramentas, para este fim, mais adotadas, principalmente por utilizar-se de uma linguagem bem comum a desenvolvedores, e, seguindo essa frente, surge o *Xamarin[[7]](#footnote-7)*, ferramenta que utiliza *C#* para produzir softwares também disponíveis em várias plataformas. *Xamarin* é uma *api* de interface que funciona para as plataformas móveis mais conhecidas e para *desktop*. Ela traduz todo o código programado para o código nativo da plataforma em que é compilado. É gratuito, porém possui versões empresariais pagas.

Até a versão 4 da *Unity3D*, era utilizado o *Xamarin* para gerar os seus jogos, portanto, é uma ferramenta viável para a criação de ferramentas, apesar de que no momento do planejamento e início do projeto, o *Xamarin* custava 300 dólares para cada plataforma que o usuário necessitasse produzir.

## *Frameworks* para jogos

Quando se define um motor de jogos, sua principal função é abstrair a criação de jogos. Em motores de jogos, tem-se um framework no qual se trabalha e cada motor possui suas próprias características e abstrações que o torna único. Todos abstraem de forma a simplificar visualmente, os jogos no final são apenas expansões criadas pelo desenvolvedor de *frameworks* robustos.

Existem *frameworks* que propõem apenas abstrair funções de baixo nível, essas ferramentas não possuem o interesse de se tornar uma *engine*, mas são utilizadas para o desenvolvimento de jogos. Nesse grupo se incluem *libgdx* e a *libsdl*, elas possuem funções básicas que abstraem a criação de imagens e sons, além de otimizações e funções que são direcionadas à criação de jogos, porém elas ainda necessitam de conhecimento técnico avançado e não conseguem abstrair totalmente a criação de um jogo.

Engines sólidas, como as já citadas, *Unreal Engine* e *Unity,* possuem seu próprio *framework* que tem como objetivo a abstração máxima e, portanto, um retorno imediato do que foi programado além de ter seu próprio ciclo de vida, que consegue abstrair até mesmo as plataformas nas quais seus jogos serão construídos.

*Unity*, como já explicado anteriormente, possui uma série de características para desenvolvimento ímpares e que não necessita de um conhecimento técnico avançado. Utiliza-se de linguagens de alto nível, *C#* e *javascript* e possui funções que estão sempre ligadas com seu editor visual de jogos, obtendo, por fim, retorno imediato da ferramenta e seu próprio *workflow*.

*Unreal Engine*, por sua vez, necessita um conhecimento técnico mais apurado e uma configuração ainda mais específica, evidenciando o porquê da linguagem escolhida ser *C++*. No entanto, os resultados obtidos são otimizados e bem diretos, além de fácil trabalho, justificando sua ampla utilização por grandes e pequenas empresas.

Desse modo é possível entender que o cerne de todo motor para jogos é o seu *framework*, sendo a característica base necessária para qualquer motor. Além disso, facilita o desenvolvimento do jogo, pois possui características para retorno imediato.

### Biblioteca LibSDL para C++

Para criação do *framework* para desenvolvimento de jogos foi utilizado a biblioteca *libSDL[[8]](#footnote-8)*, que propõe eliminar o trabalho na criação de mídia, porém ela é desenvolvida na linguagem C, uma linguagem não orientada a objetos, o que torna um pouco complicado para o desenvolvimento de jogos entre iniciantes.

O objetivo central da *libSDL* é criar uma abstração no acesso direto à mídia, como a criação de imagens, sons e *inputs* de telas, mouse, teclados e até mesmo *joysticks*. Para tal, ela implementa funções simples e as adapta para diversas plataformas, tanto para telefones celulares quanto para computadores pessoais em diversos sistemas operacionais.

Ao utilizar a *libSDL,* foram obtidos resultados de forma rápida e com pouco esforço, deixando mais espaço para a abstração dos resultados, no entanto, uma grande quantia de tempo foi gasta para gerar projetos em cada plataforma - já que em alguns casos a *libSDL* não era bem documentada e não há um padrão correto e de fácil uso -, apesar disso, ela facilitou o trabalho mais pesado, braçal, evitando a criação de algo que já é difundido e padrão, como a criação de imagens e textos - onde não há ganho significativo de memória ou processamento.

Além da *libSDL*, outras bibliotecas foram utilizadas na criação do *framework*, ferramentas essas que já são comumente integradas, dando suporte ao que a *libSDL* não se propõe fazer, como criação de textos (utilizando o formato TTF para fontes), criação de imagens fora do padrão bmp (dando suporte até mesmo a imagens com fundo transparente) e vários canais de som.

## Linguagem

Durante a pesquisa para o desenvolvimento do *software* vários fatores foram levados em consideração, uma vez que o objetivo do *framework* é ser escalável a ponto de se tornar um motor. Portanto, todas as linguagens consideradas foram linguagens orientadas a objeto. Para o projeto, após uma breve pesquisa, foi decidido utilizar C++, por motivos citados no decorrer da pesquisa.

Inicialmente foram levadas em consideração linguagens populares e de alto nível. Pensando em popularidade, foi levada em consideração a linguagem *Java*, ideal para desenvolvimento em multiplataforma, normalmente a linguagem é descartada pela comunidade de desenvolvimento de jogos, apesar de existir um *framework Java* bem difundido, o *libgdx*, já citado anteriormente e que cada vez mais possui adeptos.

As outras linguagens também cogitadas foram *C#* e *python*, porém nenhuma delas atingiu a portabilidade e maleabilidade que *C++* fornece. Na construção do projeto, nenhum *framework* foi cogitado. Porém, com o tempo gasto para desenhar apenas um *Sprite*(uma simples imagem no formato *png*) na tela, foi cogitado o uso de *frameworks*.

## Criação de projetos

Foi necessária a criação de diversos projetos, uma para cada plataforma. Utilizando um compilador para cada caso e configurações específicas, os projetos mais bem estruturados foram aqueles para a portabilidade nas plataformas *Mac*, *Windows* e *Android*.

Para tal, no *Mac,* necessitou-se criar o projeto utilizando o *XCode*, porém, dada a falta de aparelho para uso e testes, não foi possível criar uma versão final e funcional do projeto. Ao ser aberto o *XCode*, notamos uma separação de pastas do projeto principal, denominado *AGameEngineCore*, onde temos todas as funções do *framework* aqui apresentado. As dependências, já previamente construídas, estão localizadas na pasta *lib*, o projeto está acessando as bibliotecas por caminhos relativos, portanto a estrutura de pastas é primordial.

O projeto para *Windows* se tornou um pouco mais complexo, pois as bibliotecas disponíveis estão em formatos de bibliotecas dinâmicas (*dlls*). Para funcionar, as *dlls* são necessárias na pasta do projeto e devem permanecer juntas ao executável gerado. O projeto é gerado pelo *Visual Studio* - que pode ser obtido gratuitamente no site da *Microsoft* – e faz referência aos cabeçalhos também localizados na pasta *Libs*, que está localizado na mesma pasta do projeto.

O projeto gerado para o *Android* acabou se tornando o projeto mais trabalhoso. Possuindo as configurações ideais do *eclipse Android* é possível fazer uma importação sem grandes problemas, configurando a ferramenta para utilizá-la de forma adequada. Os arquivos já estão devidamente configurados, compilar para a plataforma necessita um conhecimento prévio da ferramenta e está pouco intuitivo.

A intenção do projeto foi evitar as bibliotecas pré-compiladas, já que se continuasse seguindo esse caminho a biblioteca poderia se tornar inviável em tamanho, possuindo um pacote para cada caso e como todas as bibliotecas são de código aberto, seu código pode ser facilmente distribuído junto com a ferramenta.

Para utilizar em outras plataformas, é necessária a configuração dos projetos, referente a cada situação e configurar tendo cuidado de importar as bibliotecas necessárias, localizadas na pasta Libs na raiz do projeto.

# A Game Engine

A partir do conhecimento obtido com o estudo das diferentes ferramentas para desenvolvimento de jogos atuais, decidiu-se criar um protótipo de *Game Engine* baseado em C++ fazendo uso da biblioteca libSDL, justamente por ser uma biblioteca portátil e compatível com a maior parte dos sistemas operacionais (incluindo desktops e mobile).

A criação da *engine* foi feita utilizando-se de duas ferramentas separadas: V*isual Studio*, da *Microsoft* - sendo desenvolvida no sistema operacional *Windows 8.1*; e também foi utilizado o *XCode*, *IDE* de desenvolvimento da *Apple* - por sua vez no sistema operacional *MacOsx Maverick*. Mostrando que o mesmo código pode ser usado para compilar o jogo em ambas plataformas, os compiladores utilizados também foram os proprietários dessas plataformas. O trabalho está disponível no *github* através do link: <https://github.com/Julianobsg/AGameEngine>

Toda sua estrutura possui como objetivo ser clara e simples para o usuário, portanto, alguns padrões e convenções são adotados. Para tornar mais claro o uso de apenas uma linguagem, além da linguagem de programação definida, o uso de inglês foi utilizado já que grande parte das bibliotecas e referências estão neste idioma.

Foi estipulado o uso de orientação a objeto, por ser um padrão bem aceito e mais claro para o uso da aplicação, ao criarmos jogos o tipo de abstração feito por uma linguagem orientada a objetos é necessário. Além disso, a ferramenta possui o objetivo de manter consistência, formando um produto o mais uniforme possível.

## Arquitetura

A arquitetura foi definida para facilitar a criação do jogo e abstrair do usuário a criação do *loop* principal. Neste tópico será abordado como o projeto foi estruturado e o porquê das decisões tomadas; e, na seção seguinte, será abordado com mais profundidade e com exemplos de código o que será explicado a seguir de forma ampla.

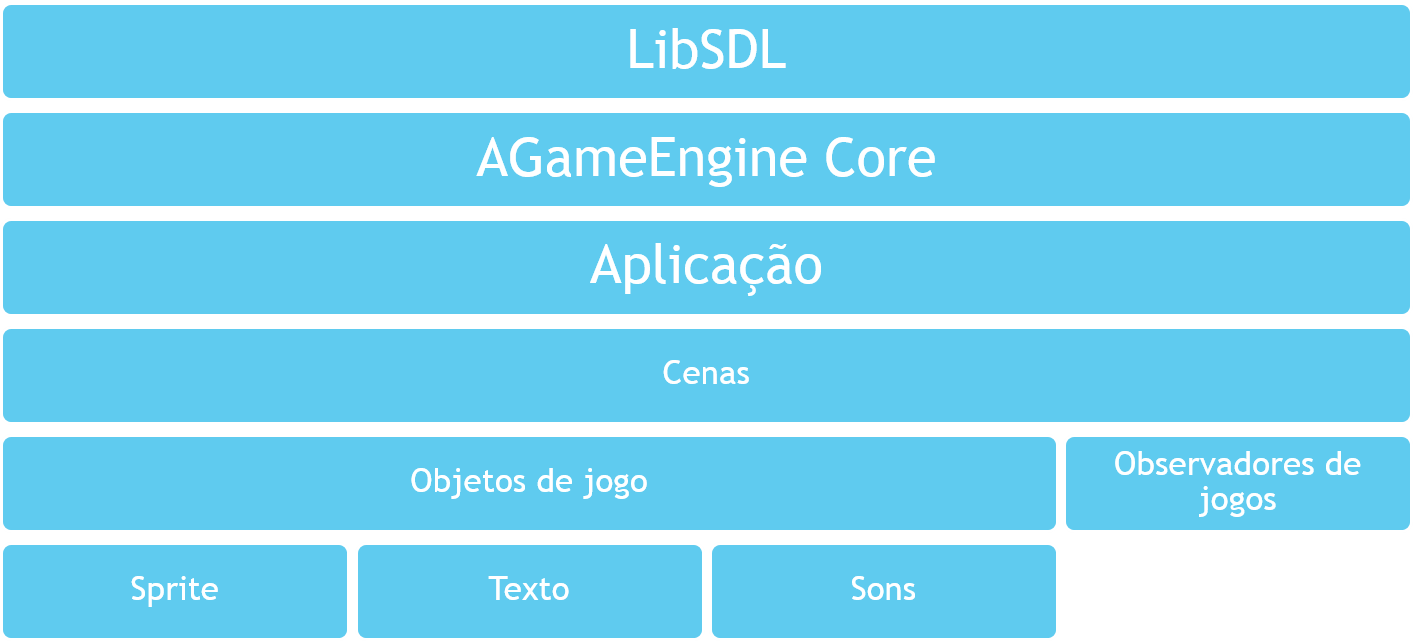


Figura ARQUITETURA DO FRAMEWORK

A base é composta pela biblioteca SDL, todo o núcleo da ferramenta utiliza metodos do framework que ficarão invisiveis para o usuário. Esse encapsulamento ocorre na camada tida como AGameEngine Core, ela é composta pela aplicação que regirá todo o sistema e essa aplicação é composta por diversas cenas, a partir dessa camada o usuário começará a trabalhar para o desenvolvimento do jogo.

Ao dividir os projetos, foi necessário criar um padrão de uso, portanto, todos os projetos possuem uma estrutura de início de arquivo semelhante, onde invocam uma classe que deve ser criada pelo usuário, determinada nos projetos - criadas como *MainGame* -, esta deve necessitar apenas dos métodos *Init* e *Run*, onde no *Init* serão feitas todas as inicializações do projeto e do jogo em si e o *Run* deve executar apenas a classe *Application*.

Esta classe possui toda a estrutura inicial do jogo e é ela quem cuida dos fluxos do jogo e dos métodos *de retorno* necessários para o ciclo de vida do jogo, pela própria característica do algoritmo ser baseado em um loop infinito, conforme o trecho de código abaixo.

int Application::\_Run()

{

Init;

\_LoadScene(0);

while (isRunning)

{

Timer::CalcFPS();

if (newScene > -1)

{

\_LoadScene(newScene);

}

CheckInputs();

//Render the scene

Draw();

Update();

}

UnloadScene();

free(mainCamera);

AudioPool::Destroy();

return 0;

}

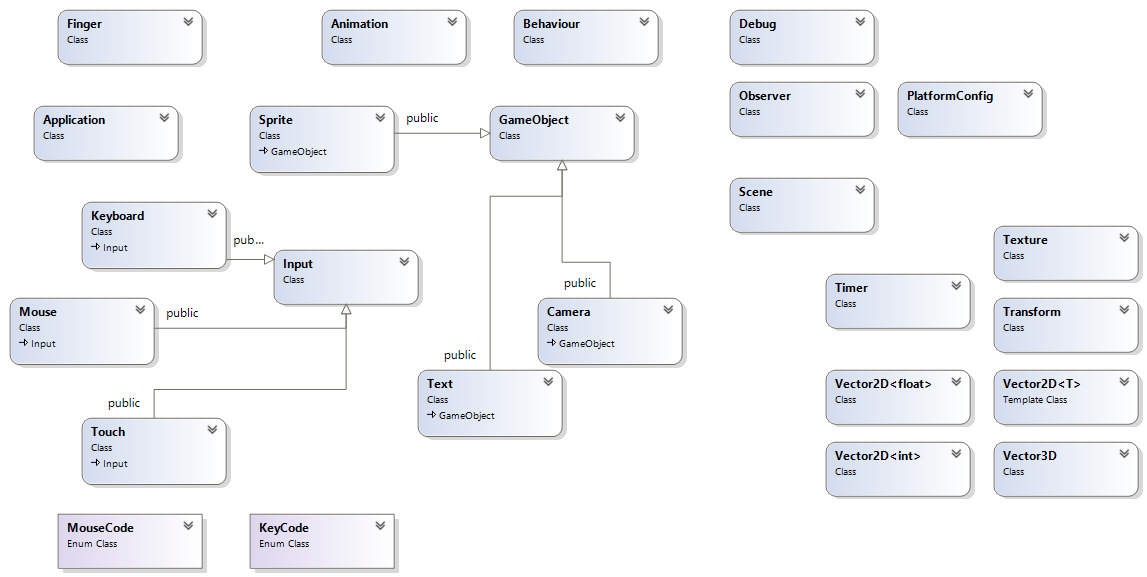


Figura Diagrama de Classes dos principais componentes da Game Engine

Ao inicializar o projeto, há várias configurações expostas na classe *application*; essas configurações são valores expostos e são automaticamente inicializados por valores específicos ao instanciar a classe, porém, esses valores devem ser substituídos ainda no método *Init*.

Esta classe é um *singleton*, estrutura que tem apenas uma instância em toda execução do sistema, que controla a estrutura de todo o jogo e que dessa forma terá apenas uma via de acesso onde a aplicação será manipulada através dela, removendo a complexidade do *looping* de jogo e controle de memória do usuário. Nela serão determinadas quais câmeras estão ativas, objetos que serão carregados e cena ativa; também utilizará todos os métodos utilizados no jogo.

### Atributos da aplicação

Tem-se o nome da aplicação atribuído na variável pública *name*. Essa variável indica o nome que estará no topo da janela ao criar-se um executável.

O tamanho da tela gerada é exposto pelo método *SetScreenSize*. Esse método recebe uma variável do tipo *Vector2D* - variável que foi criada pelo projeto para determinar um ambiente vetorial de duas dimensões, classe com *template* e que recebe como especialização o valor do tipo inteiro, portanto, o tamanho da tela são dois valores do tipo inteiro - por padrão estão definidos em X = 640 e Y = 480 - esses valores são determinados em *pixels* e o *framework* redimensiona os *Sprites* da cena dada a proporção(esse assunto será abordado adiante com mais detalhes). Ao editar o tamanho da tela, os valores serão redimensionados em qualquer momento.

Outro valor exposto é o valor *isRunning*, variável lógica simples que é iniciada com o valor verdadeiro, ao alterar para falso a aplicação sairá do *loop* de jogo e encerrará a aplicação.

A câmera principal também poderá ser alterada, essa câmera é determinada em cena e sempre podendo ser referenciada no *application*. No entanto, uma câmera padrão é criada e poderá ser referenciada a qualquer momento no projeto; essa opção é dada para o uso de várias câmeras no projeto.

### Cenas

As cenas são as estruturas primordiais do jogo, nelas serão criados e adicionados todos os objetos, *sprites* e estruturas que serão utilizados no jogo. A aplicação controla a navegação de cena e sempre executará inicialmente a primeira cena adicionada ao projeto. Essas cenas serão instanciadas e adicionadas na classe única de aplicação, através do método *Init* do *MainGame*.

Uma classe cena poderá ser criada facilmente, mas é recomendado para maior clareza e facilidade de organização do projeto, que a cena se torne uma classe criada pelo usuário que herde a classe padrão *Scene*; assim, o usuário poderá organizar e utilizar melhor os métodos dessa classe, tornando de fácil uso e com métodos devidamente organizados.

Cena

Objeto de jogo

Comportamentos

Observador

Objeto de jogo

Comportamentos

Sprite

Texto

Figura Representação gráfica da estrutura de uma cena

A organização da estrutura da cena é a cargo do usuário, mas é recomendado que no método construtor da cena sejam criados todos os seus atributos. Esses atributos são o nome da cena, que poderá ser acessada durante o jogo pelo usuário através dele, portanto, nomes claros podem facilitar a estrutura do jogo e buscas poderão ser feitas através deles - causando, dessa forma, uma navegação de tela simplificada. A parte fundamental são os objetos de jogo e os observadores, esses dois tipos de estrutura atuam constantemente na cena e é onde toda a lógica do jogo se encontra.

Esses elementos são estruturados na cena através de uma lista, tipo padrão do *C list*, e possuem acessos públicos, já que alguma ordenação ou acesso mais complexo seja necessário. Portanto, as listas de elementos do jogo possuem visibilidade pública.

### *GameObjects*

Objetos de jogo, ou *game objects*, como são estruturados no jogo, é definido como tudo aquilo que será exibido no jogo e de forma independente: um objeto de jogo possui apenas seu comportamento e não deve ter conhecimento de todo o universo, a não ser que alguns dos objetos entrem em conflito (como alguma ferramenta de física, por exemplo), aí deverá haver contato entre dois objetos de jogo e interferir em seu comportamento.

Todos os objetos possuem características em comum, a principal delas é o método *draw*. O que diferencia um *game object* em sua essência é que ele é desenhado na tela, o que o torna conceitualmente como um ator no jogo. Para desenhar na tela, precisamos de dados como rotação, tamanho e posição na cena. Para encapsular essas informações foi criada a classe *Transform*. Um dos atributos públicos do objeto de jogo.

Todo objeto de jogo possui um identificador, o atributo *name*. Esse identificador tem a intenção de tornar mais claro e fácil o acesso aos objetos, além de diferenciar vários objetos do mesmo tipo. Em alguns jogos é necessário, por exemplo, mover vários inimigos idênticos, portanto, o nome os diferenciam.

Ao iniciar um objeto de jogo, pode-se criar sua estrutura em dois lugares distintos, o primeiro lugar é o mais óbvio, o seu próprio construtor, mas em muitos jogos, necessita-se que outros construtores tenham sido inicializados para podermos inicializar um construtor específico. Por isso foi criado o método *Init*, esse método é inicializado após o construtor. *Init* é chamado no momento em que a cena é carregada, auxiliando na inicialização de todos os objetos da cena.

### *Behaviours*

Todos os objetos de jogo possuem uma forma de controle padrão, esse formato é dado via *Behaviour*(ou comportamento). Ao criar seu objeto de jogo, a ele pode ser dado um comportamento específico. Vários objetos de jogo distintos possuem comportamentos semelhantes, esses comportamentos são o que de fato ditam as regras do jogo.

Dar pontos ao jogador é um bom exemplo de comportamento. Quando matamos um inimigo A, ao morrer, ele terá o comportamento que subirá os pontos do jogador. Para evitar replicar código, ele terá a oportunidade de utilizar o mesmo comportamento em diversos objetos de jogo, ao matar um outro objeto, como um chefe, ele irá ganhar pontos também. Parâmetros podem ser passados ao inicializarmos o objeto de jogo e dizer quantos pontos serão acrescidos em cada caso; desenvolvendo assim, a necessidade mínima para replicarmos o código.

Cada comportamento possui um método de acesso, esse método é o *Update*, utilizado a cada frame, portanto, em uma aplicação padrão, ele será invocado 60 vezes por segundo. É recomendado adicionar funções de forma cautelosa já que isso pode causar um *overhead* no sistema e tornar o jogo lento caso muitas atividades forem executadas nesse método.

Todos os comportamentos também possuem um método *Init* para a inicialização pós-construtor possuindo os mesmos objetos apresentados para os objetos de jogo.

### *Observers*

A comunicação entre objetos de jogo é um desafio comum. Muitos objetos de jogo precisam referenciar objetos novos a todo momento e esses objetos comumente localizam-se na cena. Essa cena precisa gerenciar todos os objetos de jogo e muitas vezes há necessidade de criar um novo objeto apenas para gerenciar informações, que no caso não necessita ser um objeto de jogo - já que não precisa ser desenhado ou ter uma posição definida na tela.

Outro caso é a necessidade de encontrar outros objetos de jogo e fazer referência durante a partida. Esse objeto precisa ter armazenado a localização na memória de todos os objetos de jogo e fazer buscas em determinadas ações e alterar os objetos.

Para suprir essas necessidades, foi criado os *observers*, ou observadores. Estes possuem um ponteiro para a lista de objeto da cena que estão localizados e possuem os métodos já definidos anteriormente, o *Init* e o *Update*. Portanto, com esses observadores você consegue criar novos objetos na cena, criar *pool*, sistema que manterá objetos na memória e irá reutilizá-los devidamente, de objetos visíveis ou não e melhorar o acesso a memória, criando assim um sistema otimizado e específico de acesso e controle de toda a cena e de forma separada. Assim, podemos ter os mesmos observadores em diversas cenas e assumindo um papel semelhante do comportamento para evitar códigos replicados.

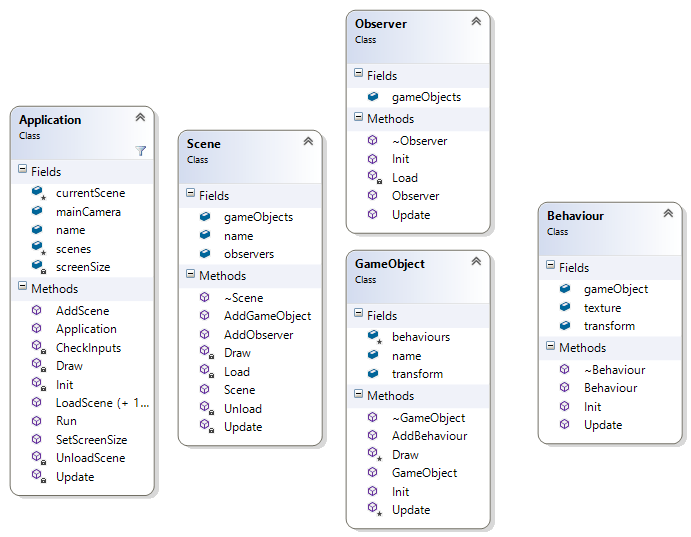


Figura 6 Resumo da arquitetura principal

## Criando um novo projeto

A ferramenta possui um *script* de instalação localizado em sua pasta raiz. Esse *script* apenas adiciona a pasta *bin* do projeto ao *path* do sistema. A pasta *bin* possui os arquivos utilizados para a criação de projetos e nele foi utilizado a ferramenta *gradle,* que necessita da instalação do *java*. Nos arquivos de configuração do *gradle* foi criada uma *task* chamada *create*. Para executar, vá para a pasta em que o projeto será criado e utilize o comando *create*, com um parâmetro com o nome do projeto, através do *script agameengine.sh*. Exemplo:

*agameengine.sh create* NovoProjeto

O projeto irá copiar os arquivos necessários para a criação de um jogo e já trará o exemplo criado, pode-se ver em sua estrutura inicial o seguinte código:

*int main(int argc, char\* argv[]) {*

*MainGame mainGame;*

*mainGame.Init();*

*return mainGame.Run();*

*}*

Onde a classe *MainGame* é criada pelo usuário e deve utilizar a classe da *Engine Application*.

Apenas por questões de facilidade de programação, o *MainGame* foi criado; apenas o objeto *Application* é necessário para executar o projeto. Nesse primeiro método,*Init* é onde ele inicializa todos os dados do jogo e é apenas um exemplo, o importante aqui é chamar o método *run*. Em seguida é chamado o método *run*, esse método apenas executa a função *application→Run*, esse é o método que espera ter tudo previamente inicializado e já executa o jogo previamente estudado.

## Recursos

A ferramenta possui uma lista de recursos básicos, esses recursos podem ser utilizados para a construção de um jogo inteiro e de forma básica. Lembrando que estamos ainda no ambiente em duas dimensões. Esses recursos controlam fluxo de jogo, imagem, animações e som.

### Aplicação

A aplicação é única no jogo: só podemos ter uma instância, portanto, ela é um *singleton*. Essa aplicação gerencia o jogo, a gestão dos itens criados se assemelha ao modelo comumente adotado em desenvolvimento de software, o MVC (*Model View Controller*) (Model, 2014). Fazendo um paralelo, o *model*, pode ser interpretado como os comportamentos que serão criados, adicionados aos objetos, contendo toda a lógica a ser executada. As *views* podem ser comparadas as cenas, onde serão adicionados os objetos, no caso os *controllers*, que possuirão, comportamentos.

Para acessarmos as funções da aplicação utilizamos através de suas variáveis estáticas no jogo, sendo ela executada pela função *Run*.

*Application::Run();*

Executado na classe exemplo *main game*. Outras funções também estão disponíveis. É a criação inicial do jogo, a aplicação determina também qual será a câmera principal, através do método:

*Application::SetMainCamera(mainCamera);*

O nome do jogo indica qual é o nome que é exibido no topo da janela e configurações exibidas.

*Application::SetGameName(string name);*

Seguindo o mesmo padrão de chamada, temos as funções *CloseGame()*, que pode ser executada a qualquer momento do jogo; e ele será interrompido e a memória devidamente limpa. É possível pegar o tamanho da tela em que o jogo será exibido e editar.

*AddScene* criará cenas e o conceito de cenas será mais detalhado a seguir.

### Cenas

Cenas são objetos que estruturam tudo que será mantido no jogo. Uma cena é basicamente um mapa, onde objetos e informações serão mantidos. Cenas podem receber *game objects*, que são objetos gerais de jogo; *sprites,* por exemplo, são objetos de jogo que podem ser carregados nas cenas necessárias. As funções necessárias da cena são de adicionar novos objetos de jogo.

*scene1.AddGameObject(sprite);*

As cenas possuem duas etapas de inicialização, assim como grande parte das classes da ferramenta, o construtor e a classe *Init*. A classe *Init* é invocada a cada momento que a cena é carregada.

Quando a cena é encerrada, ou seja, ao trocarmos de cena, todos os objetos vinculados são destruídos. Portanto, os objetos só podem ser adicionados no método *Init()*.

Adicionar nomes, através da variável *name*, é possível carregar e descarregar cenas através do nome e índice na aplicação (*item application*). Cenas podem ser adicionadas na aplicação antes de da aplicação ser executada.

*Application::AddScene(scene1);*

### Vetores

Ao trabalharmos com jogos em duas dimensões, com frequência necessitamos um vetor de posições com dados X e Y, representando um ponto no espaço ou tamanho. É comum trabalhar com valores inteiros para referenciarmos pontos, já que não é possível termos, tecnicamente falando, um ponto em 0,5 pixels por exemplo. Porém, com a funcionalidade de tamanho escalável de acordo com a resolução da tela, vários pontos no jogo são dados em decimais.

A classe *Vector2D* trabalha com esses pontos, porém, ela é definida através de um *template* onde os dois *templates* definidos variam em inteiro ou flutuante (*integer, float*). Para usarmos o vetor, portanto, usamos:

*Vector2D<float> floatVector;*

*Valores tanto inteiro e flutuantes possuem as mesmas funções:*

*Vector2D<int> Zero();*

*Vector2D<int> One();*

Ambas retornam vetores do tipo definido, ou com ambos os valores preenchidos. O método *Zero()* retorna o vetor preenchido com 0, convencionalmente usado para posição e o método *One()* o vetor preenchido com 1, que pode ser usado ao trabalhos com escalas.

*SetVector(x, y);*Retorna um vetor com os valores indicados.

*Abs();* Retorna o vetor com valores positivos.

### Objetos de Jogo

Objeto de jogo é uma das bases da ferramenta, uma classe virtual e que faz referência a todos os objetos que possuem uma posição no espaço, tamanho e rotação, que é obtido através do *Transform*, descrito a seguir. Além de um ciclo de vida em comum, um ponto de início.

Além da posição, objetos de jogo possuem um ciclo de vida em que temos o construtor - que é iniciado antes de todos os objetos serem posicionados e a cena devidamente criada. Após a construção da cena e posicionamento dos objetos é invocado o método *Init*, e nesse momento já possuímos a cena construída e o método é invocado como um segundo construtor, útil para referenciar novos objetos e obter dados únicos da cena.

#### Transform

Todo objeto de jogo possui uma classe de transformação chamado *transform*. Essa classe contém as informações relativas ao objeto de jogo como posição, rotação e escala. Todos esses parâmetros são públicos e passíveis de alteração e podem ser utilizados pelo código do usuário. Esse objeto é necessário para compartilhar informações necessárias e evitar problemas de código cíclico. Escala relativa ao seu tamanho, por padrão, essa escala assume a forma de um vetor unário de duas dimensões. Ao aumentar ou diminuir tem-se uma multiplicação do tamanho que ocupa na tela, e caso esse número se torne negativo, ele inverte a imagem, tanto no eixo X, quanto no eixo Y.

*objeto→transform→scale.x = -1;*

Além da escala, é possível rotacionar o objeto; ele é iniciado com os valores em zero, podendo rotacionar no eixo X e no eixo Y. As transformações de rotação também são relativas a posição 0 e trabalham em graus, então, para rotacionar, é só alterar os valores da opção *rotation*. O último elemento é a posição que é determinada de acordo com o eixo x, y da tela e podendo estourar os limites da tela. Bastante utilizado e nenhuma operação matemática é feita, o usuário pode utilizá-lo para mover o objeto.

### Câmera

Câmeras são objetos de jogo único, elas não são desenhadas na tela, porém necessitam de posição e tamanho. Sua função primordial é desenhar objetos na tela. A ferramenta trabalha automaticamente com uma câmera, e um objeto do tipo câmera é criado durante o fluxo da aplicação, essa câmera pode ser obtida através do método estático:

*Application::GetMainCamera();*

As câmeras determinam qual o tamanho do objeto que será desenhado na cena que varia de acordo com a resolução da tela e determinadas proporções; com isso, o usuário poderá criar um jogo que funcionará com resoluções diferentes e sendo automaticamente tratadas durante a inicialização do *framework*.

O tamanho da tela da câmera não necessariamente é o tamanho inteiro do jogo, apesar de por padrão ser delimitado assim, sendo que pode existir câmeras menores que a tela do jogo. Portanto, existe a função para mudar seu tamanho padrão.

*mainCamera->SetScreenSize(this->screenSize);*

Para trabalhar com posições e tamanhos relativos, todos os objetos visualizados possuem um tamanho específico dado o tamanho em X da câmera - por padrão, esse tamanho é inicializado com o valor 10, porém, ele pode ser alterado através da variável pública *size*.

Para obter os dados reais, e não relativos:

*mainCamera-> WorldToCameraTransform (this->transform);*

*mainCamera-> WorldToCameraScale (this->transform.scale);*

*mainCamera-> WorldToCameraPosition (this->transform.position);*

### Texto

Textos são gerados a partir de fontes e podem ser posicionados em qualquer região da tela. Para inicializá-lo é necessário passar o caminho do arquivo da fonte que será usado: o *framework* não usa nenhuma fonte padrão, portanto, essa etapa é necessária.

*Text\* button = new Text("CaviarDreams.ttf");*

Com a fonte inicializada, é necessário preencher com o texto (o texto padrão é o texto vazio).

*button->SetContent(“Texto exemplo”);*

É possível alterar a cor e para tal, é necessário informar o código RGB da cor e passá-lo através da função *ChangeColor(),* a função irá aceitar os campos variando de 0 a 255.

*button->ChangeColor(0, 255, 255);*

### Sprites

Nessa primeira etapa, foi criado um sistema simples de *sprites* onde o programador apenas necessita informar o caminho da imagem. Um dos problemas apresentados foi a diferença entre os caminhos de acordo com o sistema operacional envolvido.

*Sprite\* sprite = new Sprite;*

*sprite→AddTexture("Media/batafire-spritesheet.png");*

Com isso, uma textura é adicionada a lista de imagens, contida na pasta *Media*. Essa imagem será por padrão posta na posição 0,0 da tela. Além disso, ela terá o tamanho proporcional ao tamanho da imagem em relação a tela, então, se tivermos uma imagem de 50 por 50 pixels, ela irá seguir esse tamanho em relação a resolução da tela. A resolução da tela é, por padrão, inicializada com os valores de 640 por 480. Essa resolução também poderá ser alterada.

#### Animação

Para animar o *sprite* é necessário colocar mais texturas, porém, estas serão colocadas em um objeto da classe *Animation*.

*Animation\* newAnimation = new Animation;*

*newAnimation→AddTexture("Media/batafire-spritesheet.png");*

Essa *animation* torna-se um *frame* da animação. Ao adicionar várias texturas em uma animação, ela seguirá sequencialmente. A classe *animation* possui configurações de *frames* por segundo de exibição entre outras opções. Após adicionar a textura, é necessário vincular o sprite a animação, o que também é bem simples.

*sprite→AddAnimantion(newAnimation);*

Também é possível retornar a última textura da animação.

*sprite→LastTexture();*

#### Clipping

*Clipping* é a função de separar um pedaço da textura, determinando um tamanho a textura referência a mesma imagem ,porém possui a capacidade de exibir pedaços distintos da mesma imagem, essa função é muito útil para separar *spritesheets*. Pode-se, por exemplo, cortar a última textura adicionada no *sprite*. Os parâmetros do *clipping* são: a posição inicial do corte em X,Y e os tamanhos da imagem que será resultante do corte. Há a opção de adicionar *clips* - ele gera um outro apontador para a mesma textura assim não gerando cópias da textura.

*sprite→AddClip(75 , 0, 75, 75);*

*sprite→AddClip(180, 0, 75, 75);*

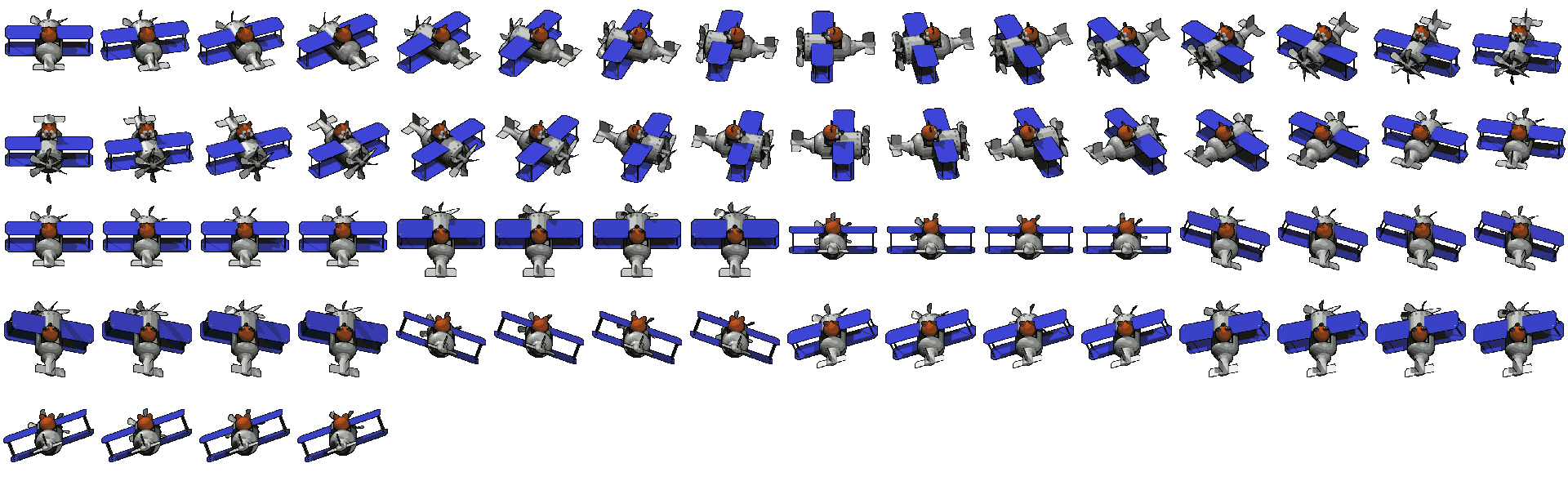
*sprite→AddClip(280, 0, 75, 75);*

Figura exemplo de spritesheet

### Comportamento

Todo objeto de jogo pode possuir comportamentos, esses comportamentos dão uma ação ao objeto. Podemos exemplificar que, para cada um objeto de jogo que represente um jogador, seus comportamentos sejam se movimentar e atacar, por exemplo. Cada comportamento tem ação ao objeto de jogo em que ele está localizado e podendo fazer uma configuração no mesmo, no entanto, esse objeto só será acessível após a inicialização do objeto utilizando o método *Init().*

Para criar o comportamento, seguimos o padrão de herança utilizado nas outras classes; esse comportamento deve herdar a classe *Behaviour*. Essa classe deve implementar os métodos *Init()* e *Update().*

O método *Update* é onde grande parte da lógica do objeto é criada, executado a cada frame do jogo. Ele poderá acessar os *inputs* do jogo e até mesmo outros objetos.

O comportamento pode ser adicionado na cena a qualquer momento, porém ele não será executado logo após inserido, ao ser inserido o método *Init()* será invocado e o método de *Update* só será utilizar no próximo frame.

Se o objeto do jogo em questão for removido antes do comportamento dele ter sido executado, o objeto irá esperar executar o comportamento naquele mesmo frame e depois será destruído logo em seguida.

### Observadores

Observadores se diferem aos objetos de jogo por não possuírem posição ou imagem na tela, assim, eles tornam mais leves e simples de trabalhar. Possuem acesso à cena e são úteis para controlar estatísticas de jogo e ações que dependam de vários objetos do jogo, por exemplo, um gerador de inimigos aleatório.

Para utilizá-lo é necessário criar uma classe e herdar o observador, exemplo:

*class ExampleObserver :*

*public Observer*

*{*

*virtual void Init();*

*virtual void Update();*

*};*

Com isso, o observador possui os mesmos métodos que são executados que um objeto de jogo, com as mesmas regras, porém, objetos de jogo serão executados antes em ambos os métodos. Após criar um observador é necessário adicioná-lo na cena.

*AddObserver(new ExampleObserver);*

Os objetos de jogo possuem a referência da cena e essa referência pode ser obtida durante o método *Init()*, com isso, um observador possui acesso a toda cena executada e pode trabalhar da melhor forma que o usuário desejar.

### Controles

Todas as formas de controle herdam da classe *Input*, a qual não possui nenhum método padrão e apenas definem sintaticamente. No futuro essa classe pode ser usada de forma mais ampla para analisar os sistemas de controle multiplataforma.

No momento, estão sendo captados três tipos distintos de *input* do jogador. Eles são inseridos em *singletons*, que pega toda a informação de um único jogador. Esses *inputs* são o teclado, mouse e o toque de telas sensíveis ao toque.

#### Mouse

O mouse é a classe única mais simples, possuindo um método para acessar a posição atual do mouse e um método para identificar se algum dos botões foi usado; detecta apenas os botões esquerdo, direito e do meio. Botões configuráveis de mouse não são detectados.

O método para adquirir a posição retorna um *Vector2D* do tipo inteiro e dá a posição exata em pixels e de onde foi tirada, para conseguir a posição relativa, esse valor deverá ser passado para a câmera executar a devida correção.

*Mouse::Position();*

Para saber se um determinado botão de mouse foi clicado, utilizar o enumerador *MouseCode*, que lista os valores para cada botão, ao passar na classe *MouseButtonDown*, ela retornará um valor lógico que representará se o botão foi clicado ou não.

*Mouse:: MouseButtonDown(mouseButton);*

#### Teclado

Teclados são mais simples, a única função registrada é a função *KeyDown*, onde ela retorna um valor lógico se a teclada passada foi apertada. A tecla é o enumerador *KeyCode*. Nele é listado todas as teclas de um teclado padrão.

*Keyboard::KeyDown(keyCode);*

#### Toque

O uso de toques é ligeiramente mais complicado já que ele precisa definir o multi-toque nas telas. Para tal, a classe *Touch* possui o método *Touching* que retorna variável lógica que é verdadeira caso haja algum toque.

*Touch::Touching();*

Para sabermos se possui algum dedo na tela, temos a função *GetFinger*, que retorna um objeto do tipo *Finger()*, objeto que possui dados do toque, que são posição e um identificador, que é possível saber o id do toque.

*Finger \*finger = Touch::GetFinger(0);*

*position = finger->position;*

*id = finger->id;*

### Áudio

A ferramenta de áudio é a única ferramenta que está em no *namespaceAGameEngine*, por esse motivo, para usá-la deve-se acessar o *namespace* necessário. Para iniciar, é necessário invocar o construtor passando o caminho do som que será utilizado.

*Audio\* audio = new Audio(“musica2.wav”);*

Pode ser executado o efeito sonoro ou música apenas uma vez, para isso, é simplesmente chamar o objeto de áudio através do método *Play();*

*a udio->Play();*

Principalmente quando tratamos de música, é comum pegarmos um trecho e repetirmos infinitas vezes, para isso usamos a função *PlayLooped()*, como exemplificado a seguir.

*a udio->PlayLooped();*

Também podemos determinar um número finito de repetições, que podem ser passadas através do mesmo método para repetição infinita, porém passando um valor inteiro.

*a udio->PlayLooped(repeatTimes);*

Várias vezes precisamos determinar se algum efeito sonoro ou trecho de música está sendo executado. Para isso, temos que ter a referência exata para o áudio que queremos e utilizar o método *IsPlaying()*, que retorna uma variável lógica.

*a udio->IsPlaying();*

É possível alterar individualmente o volume de cada áudio utilizado, por isso, estão listados a seguir os métodos que acessa a esse valor, onde *Volume()* informa o valor e o método *SetVolume(inteiro),* altera o mesmo. O volume varia entre 0 e 128 representado por números inteiros.

*a udio->Volume();*

*a udio->SetVolume(volume);*

### Tempo

Muitas vezes é útil termos funções que controlam o tempo do jogo, essas funções evitam que o jogo atinja mais que um determinado número de *frames* por segundo, ou avisem ao jogo caso esses *frames* estejam muito baixos. Muitas vezes necessitamos corrigir a velocidade de objetos ou acerá-los através do tempo de jogo, e para isso foi criada a classe estática *Timer*, que calcula os *frames* desenhados por segundo.

*Timer::framesPerSeond;*

Para compensarmos a velocidade de um objeto de jogo em relação a esse tempo, é necessário multiplicarmos sua velocidade ao valor da variação do tempo, que é o inverso dos *frames* por segundo, se determinarmos esse valor como 60, temos uma variável que nada mais é que 1/60.

*Timer::deltaTime;*

# Estudo de caso

Para propósitos de demonstração da ferramenta, foi criado o jogo “*Tomato Attack*”; nele, controlamos uma nave e temos como objetivo atirar nos tomates. É uma simples demonstração, já que o jogo não possui fases, nem uma progressão. Os tomates aparecem infinitamente até que o jogador morra, como a dificuldade não aumenta, ao pegar o jeito é possível jogar infinitamente.

O projeto está localizado na pasta *AGame* do projeto e as artes e *assets* estão localizados na pasta mídia. As artes foram adquiridas de forma gratuita no site:

<http://poufpoufproduction.fr/>

O jogo foi dividido em três cenas, onde cada uma delas cumpre o papel de manter o fluxo do jogo, sendo elas o início, o jogo em si e uma tela de fim de jogo. Todas elas são instanciadas e colocadas na aplicação; após instanciadas, elas são executadas na ordem em que são tratadas, a única necessidade é a primeira cena adicionada a camada de aplicação é de fato a ser a primeira cena executada.

Figura CÓDIGO PARA A CRIAÇÃO DAS CENAS

void MainGame::Init()

{

MainMenu\* menu = new MainMenu;

menu->name = "MainMenu";

GamePlayScene\* scene1 = new GamePlayScene;

scene1->name = "Gameplay";

EndGameScene\* endGame = new EndGameScene;

endGame->name = "EndGame";

Application::SetScreenSize(Vector2D<int>(720, 640));

Application::AddScene(menu);

Application::AddScene(scene1);

Application::AddScene(endGame);

}

## Tela de inicio

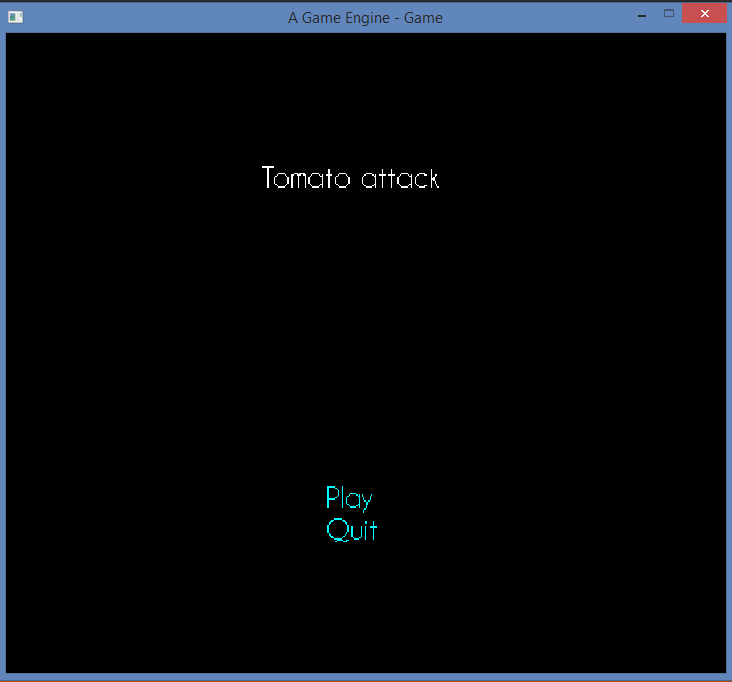
Nela é exibido apenas o título do jogo e dois botões, jogar e sair. Para a criação dos botões para evitar a repetição de código, foi criada uma classe *button*, que usa *call-backs* para executar após clique, todos os botões possuem o texto e o *call-back*. Nessa cena foram criados 1 texto e dois botões que derivam a texto, para a

Figura Menu inicial

navegação os call-backs foram criados na inicialização da cena que possuem a função de ir para a cena seguinte ou utilizar: *Application::CloseGame();*

A classe *UIButton* possui a função de mudar a cor do texto ao passarmos o mouse sobre.

## O jogo

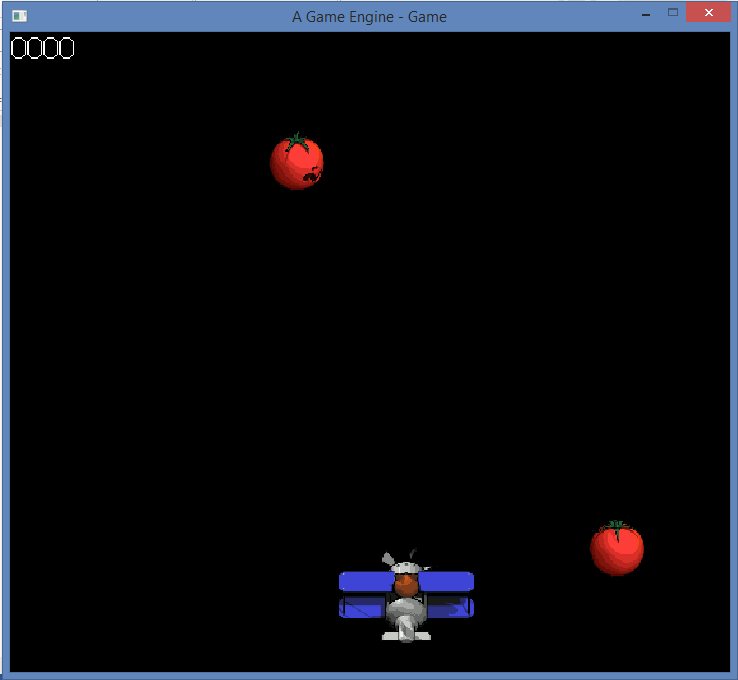
O jogo é composto por dois observadores, além de *Sprites* e o texto que acompanha o score, cada observador tem o poder de controlar os eventos do jogo. Foram implementados um sistema simples de colisão, porém nada otimizado e que é executado a cada milissegundo o que o torna pesado e a fonte de atrasos ao criarmos o jogo, para um sistema mais robusto é recomendo implementar melhores sistema de colisão, ou usar bibliotecas prontas para tal.

Figura O jogo

Além do sistema de colisão, foi criado outro observador, seu nome é abrangente, com o nome de *GameActionObvserver*, ele acumula funções, inicialmente ele é avisado a cada inimigo destruído e sua função é aumentar o placar do score. Além disso, ele checa a existência do jogador, e, se por acaso o jogador estiver morto, esse mesmo observador carrega a cena de fim de jogo, passando o score como parâmetro.

Sua outra função é controlar a criação dos tomates, caso fosse algo mais complexo, como algo que mude a dinâmica do jogo, seria melhor criar um terceiro observador. Porém, não foi necessário a criação do terceiro e não houve muitas linhas de código. Os tomates são instanciados pelo observador em uma posição no eixo X aleatória e sua única função é descer em direção ao fim da tela, se entrar em contato com o jogador, ele é automaticamente destruído.

A criação dos Sprites foi em seu início determinar os comportamentos e as animações referentes, o tomate possui apenas uma animação, girando constantemente e dois comportamentos, descer na tela e destruir o jogador. Porém o jogador é um Sprite mais robusto e possui um conjunto de animação ao estar parado, virando à direita e virando à esquerda, que é acionado de acordo com a função do teclado. O jogador também possui uma função de instanciar tiros, que é ativado ao apertarmos a tecla espaço.

## Fim de jogo

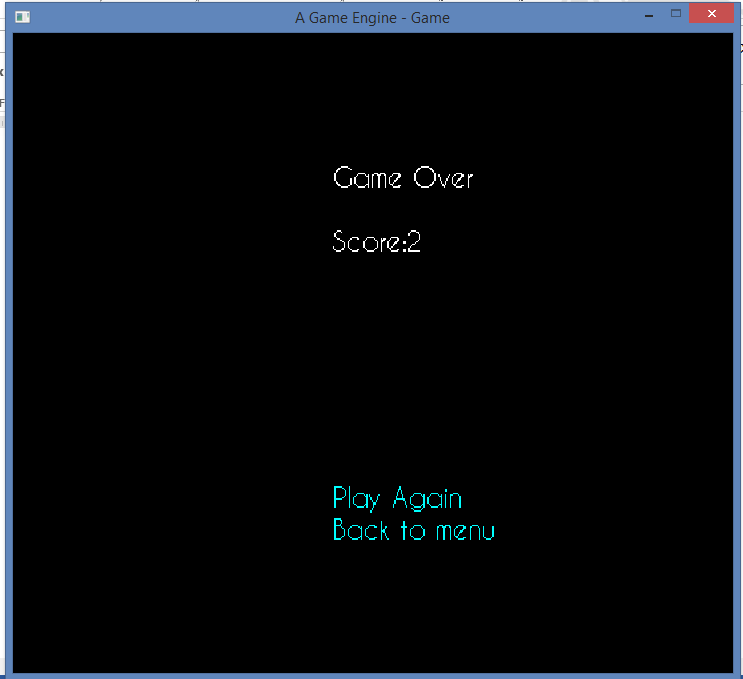
Essa cena possui um parâmetro que é passado através do observador anterior, possui dois botões, que usam a mesma função da cena de menu principal, esses botões carregam a cena inicial, ou a cena final. Cena simples que conclui o ciclo do jogo, mostrando um final e exemplifica como se tornaria uma transição de parâmetros entre cenas.

Figura Fim de jogo

## Demonstração

Um jogo simples que serve de demonstração a todos os recursos, através dele alguns bugs não encontrados anteriormente foram encontrados e alguns recursos novos foram pensados. Graças a essa demonstração temos uma ferramenta que pode ser utilizada na criação de um jogo rudimentar em duas dimensões.

O jogo não possui adaptação total ao recurso de toque, portanto não foi disponível sua versão para versões móveis, porém ele foi executado com sucesso na versão *android*, necessitando apenas de uma adaptação do sistema de controle do jogo. A demonstração ainda pode evoluir muito e a ferramenta ainda mais aprimorada.

# Conclusão

Criar uma ferramenta para desenvolvimento de jogos é um trabalho bastante amplo e complexo; todas as ferramentas que se propõe a tal, vivem em um fluxo constante de melhorias. Assim sendo impossível comparar a ferramenta projetada com qualquer ferramenta comercial, ou até mesmo de código aberto, que é produzida por equipes e há um período maior de desenvolvimento.

O pequeno projeto conseguiu obter com sucesso o que propôs, mas ainda é necessária grande evolução para tornar-se uma ferramenta útil para pequenos desenvolvedores. O custo para desenvolver jogos com ela ainda é grande, sendo que com as ferramentas comerciais podemos produzir algo semelhante com um tempo menor.

Podemos notar a dificuldade de gerenciamento de memória, versionamento, e até mesmo a forma de criar projetos, que, por sua vez, são custosos. Mesmo utilizando uma biblioteca que facilitou grande parte do trabalho, criar um *framework* multiplataforma se mostrou uma tarefa árdua e constante. Podemos utilizar ferramentas para a criação de arquivos, como foi utilizado o *gradle* para gerar o projeto principal e aumentar o nível de configuração, podendo parametrizar nomes e outras particularidades do projeto.

Criação de projetos também deveria ser um recurso compartilhado, onde todas as partes do projeto utilizam a ferramenta do mesmo local. Para isso, seria necessária uma ferramenta para *builds*, uma ferramenta que facilitaria o uso do *framework*.

No entanto, uma demonstração básica foi gerada, mesmo sendo custoso o seu desenvolvimento, houve uma melhora na forma em que se configura e compila um novo projeto. A criação de projetos se torna possível e um exemplo que pode ser editado e transformado em um jogo mais complexo e amplo pode ser feita.

Referências

Apache Cordova. Disponível em: <http://cordova.apache.org />. Acesso em: 13 mar. 2015.

BONDI, André B. **'Characteristics of scalability and their impact on performance', Proceedings of the 2nd international workshop on Software and performance**, Ottawa, Ontário, Canadá, 2000. pp. 195 – 203. Disponível em: <http://www.win.tue.nl/~johanl/educ/2II45/Lit/Scalability-bondi%202000.pdf> Acesso em: 8 abr. 2012.

BOYER, Brandon. **SGS Feature: 'Serious Game Engine Shootout'.** 2007. Disponível em: < http://www.gamasutra.com/view/news/12772/SGS\_Feature\_Serious\_Game\_Engine\_Shootout.php>. Acesso em: 23 nov. 2014.

BUTTS, Steve. **Gears of War hands on.** 2007. Disponível em: < http://www.ign.com/articles/2007/10/05/gears-of-war-hands-on>. Acesso em: 24 nov. 2014.

COCOS 2D. **Migrating to -x**. Disponível em: <<http://cocos2d-objc.org/migrating-to-x/>>. Acesso em: 20 abr. 2015.

Developer tech. Disponível em: <http://www.developer-tech.com/news/2014/nov/06/6-popular-mobile-development-tools/>. Acesso em: 13 mar. 2015.

Godot Engine. Disponível em: <http://www.godotengine.org/wp/>. Acesso em: 17 mar. 2015.

GODOT. **Overview: Project Description**. 2014. Disponível em: <<http://www.godotengine.org/projects/godot-engine>>. Acesso em: 26 nov. 2014.

GREGORY, J. **A Game Engine Architecture**. A K Peters: 2009. 864 p.

GTK+. Disponível em: <http://www.gtk.org/>. Acesso em: 13 mar. 2015.

MAKUCH, Eddie. **Unreal Engine 4 Now Free for Everyone**. Disponível em: <<http://www.gamespot.com/articles/unreal-engine-4-now-free-for-everyone/1100-6425627/>>. Acesso em: 14 mar. 2015.

MODEL, Mitchell L. **Model View Controller History**. 2014. Disponível em: <<http://c2.com/cgi/wiki?ModelViewControllerHistory>>. Acesso em: 20 abr. 2015.

Phonegap. Disponível em: <http://phonegap.com/>. Acesso em: 13 mar. 2015.

Qt. Disponível em: <http://qt-project.org/>. Acesso em: 13 mar. 2015.

RABIN, Steve. **Introdução ao Desenvolvimento de Games : Tradução da 2ª edição norte-americana : Programando Vol. 2.** 2 ed. São Paulo: Cengage, 2011. 480 p.

SOMASEGAR, S. **Visual Studio for Game Development: New Partnerships with Unity, Unreal Engine and Cocos2d.** Disponível em: <<http://blogs.msdn.com/b/somasegar/archive/2015/04/17/visual-studio-for-game-development-new-partnerships-with-unity-unreal-engine-and-cocos2d.aspx>>. Acesso em: 20 abr. 2015.

TERATHON. **C4 Engine.** Disponível em: <http://www.terathon.com/>. Acesso em: 17 mar. 2015

THOMSEN, Mike. **History of the Unreal Engine**. 2010. Disponível em: <<http://www.ign.com/articles/2010/02/23/history-of-the-unreal-engine>>. Acesso em: 24 nov. 2014.

THORSEN, Tor. **First Look: Unreal Engine 3.0**. 2004. Disponível em: <<http://www.gamespot.com/articles/first-look-unreal-engine-30/1100-6098240/>>. Acesso em: 24 nov. 2014.

Ubi-art. Disponível em: < http://ubi-art.uk.ubi.com/>. Acesso em: 18 mar. 2015

UNITY. **Documentation**. 2014. Disponível em: <<http://docs.unity3d.com/Manual/MonoDevelop.html>>. Acesso em: 26 nov. 2014.

Unity3D. Disponível em: < http://unity3d.com// >. Acesso em: 18 mar. 2015

Unreal Engine 4. Disponível em: < http://www.unrealengine.com// >. Acesso em: 18 mar. 2015

WARD, Jeff. **“What is a Game Engine?”**. 2008. Disponível em: < http://www.gamecareerguide.com/features/529/what\_is\_a\_game\_.php>. Acesso em: 15 nov. 2014.

Xamarin. Disponível em: <http://xamarin.com/>. Acesso em: 13 mar. 2015.

1. Em tradução livre: *“Unreal Engine 4 is a complete suite of game development tools made by game developers, for game developers. From 2D mobile games to console blockbusters, Unreal Engine 4 gives you everything you need to start, ship, grow and stand out from the crowd”*. Disponível em: <https://www.unrealengine.com/unreal-engine-4>. Acesso em: 14 mar. 2015. [↑](#footnote-ref-1)
2. http://ubi-art.uk.ubi.com/ [↑](#footnote-ref-2)
3. http://www.lua.org/ [↑](#footnote-ref-3)
4. http://www.qt.io/ [↑](#footnote-ref-4)
5. http://www.gtk.org/ [↑](#footnote-ref-5)
6. https://cordova.apache.org/ [↑](#footnote-ref-6)
7. https://xamarin.com/ [↑](#footnote-ref-7)
8. https://www.libsdl.org/ [↑](#footnote-ref-8)