UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO

Gabriel Capeletti

THE ROTFATHER: DESENVOLVIMENTO DE UM ADVENTURE GAME COMERCIAL

Florianópolis

2017

Gabriel Capeletti

THE ROTFATHER: DESENVOLVIMENTO DE UM ADVENTURE GAME COMERCIAL

Tese submetida ao Programa de Graduação em Ciência da Computação para a obtenção do Grau de Bacharel.
Orientador: Prof. Dr. Raul Sidnei Wazlawic

Florianópolis

2017

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Capeletti, Gabriel
The Rotfather : Desenvolvimento de um adventure game comercial / Gabriel Capeletti ; orientador, Raul Wazlawic, 2017.
53 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Ciências da Computação, Florianópolis, 2017.

Inclui referências.

1. Ciências da Computação. 2. Game. 3. Game Development. I. Wazlawic, Raul. II. Universidade Federal de Santa Catarina. Graduação em Ciências da Computação. III. Título.

Gabriel Capeletti

THE ROTFATHER: DESENVOLVIMENTO DE UM ADVENTURE GAME COMERCIAL

Esta Tese foi julgada aprovada para a obtenção do Título de "Bacharel", e aprovada em sua forma final pelo Programa de Graduação em Ciência da Computação.

Florianópolis, 26 de junho 2017.

Prof. Dr. Renato Cislaghi
Coordenador de Projetos

Banca Examinadora:

Prof. Dr. Raul Sidnei Wazlawic
Orientador

Prof. Dr. Mônica Stein

Prof. Dr. José Eduardo De Lucca

AGRADECIMENTOS

Agradeço aos meu pais por terem me dado a liberdade de seguir meus sonhos e à minha namorada por sempre acreditar em mim, mesmo nas horas mais difíceis.

RESUMO

A indústria de jogos digitais têm crescido de uma maneira muito rápida nas últimas década, ao mesmo tempo que esta promove um mercado criativo e de entretenimento ela também incentiva o avanço tecnológico. Tais avanços acabam por serem utilizados em vários campos da ciência. Neste trabalho serão aplicados conhecimentos de programação, cálculo, engenharia de software, computação gráfica e inteligência artificial necessários na criação de um jogo digital.
Ao fim deste trabalho terá sido construído um jogo do estilo *adventure*, com capacidade de competir no mercado de jogos digitais atual. Apesar de o Brasil ser um grande consumidor de jogos digitais ele ainda está muito atrás no quesito de desenvolvimento, isto se deve à um conjunto de dois fatores, o pouco incentivo governamental junto com a grande quantidade de impostos cobrados de pequenas empresas formam uma grande barreira para novos desenvolvedores, a cultura brasileira ainda não encara a indústria de jogos digitais como algo sério. Este trabalho irá incentivar aspirantes brasileiros e guiá-los no desenvolvimento de um jogo, por consequência também contribuirá para que o meio acadêmico se aproxime da indústria dos jogos digitais.

Palavras-chave: *game, game programming, game development.*

LISTA DE FIGURAS

Figura 1 *Screenshot* do jogo Braid. 18
Figura 2 *Screenshot* do jogo Limbo. 19
Figura 3 Exemplo de efeito *split screen* 21
Figura 4 Ciclo de vida da Unity. 30
Figura 5 GameObject vazio. 31
Figura 6 Componente *Rigidbody2D*. 31
Figura 7 Opções de colisores e comportamentos. 32
Figura 8 Componente *Sprite Renderer*. 33
Figura 9 Diagrama do fluxo de renderização. 33
Figura 10 Arquitetura base. 35
Figura 11 Diagrama de classe do controlador. 38
Figura 12 Interface da ferramenta de animação 41
Figura 13 Exemplo de uso do sistema dentro da *Unity*. 44
Figura 14 Visão de uma câmera com perspectiva. 45
Figura 15 Visão de uma câmera ortogonal. 45
Figura 16 Monitor com resolução 1920x1080 pixels. 46
Figura 17 Monitor com resolução 800x600 pixels. 47
Figura 18 Exemplo de uso de múltiplas *viewports in-game*. 48
Figura 19 Interface de configuração de duas *Viewports*. 48
Figura 20 Exemplo de grafo criado com a ferramenta *path plot*. 49
Figura 21 Interface criada com a ferramenta *path plot*. 50

LISTA DE ABREVIATURAS E SIGLAS

IGDA *International Game Developers Association*
HLSL *High-Level Shading Language*
NPC *Non-Player Character*
DTO *Data Transfer Object*
QA *Quality Assurance*
PC *Personal Computer*
SBGames Simpósito Brasileiro de Games
TI Tecnologia da Informação
API *Application program interface*
Indies Independente, no contexto de uma empresa de jogos se refere à um jogo que não teve investimento de outra empresa.

SUMÁRIO

1 INTRODUÇÃO 17
1.1 CONTEXTUALIZAÇÃO 17
1.1.1 Definição de jogo eletrônico 17
1.1.1.1 Braid 18
1.1.1.2 Limbo 19
1.1.1.3 Super meat boy 19
1.1.2 Definição da equipe 19
1.2 OBJETIVOS 21
1.2.1 Objetivo Geral 21
1.2.2 Objetivos Específicos 21
2 REVISÃO BIBLIOGRÁFICA 23
2.1 RESUMOS 23
2.1.1 Game Development: Harder Than You Think 23
2.1.2 The whats and the whys of games and software engineering 24
2.1.3 What Went Right and What Went Wrong: An Analysis of 155 Postmortems from Game Development 24
3 DESENVOLVIMENTO 27
3.1 ESCOLHA DE FERRAMENTAS 27
3.1.1 Bibliotecas gráficas 27
3.1.2 Motores de física 28
3.1.3 *Game engines* 28
3.2 ESTRUTURA DA UNITY 29
3.2.1 Programação Orientada a Componentes 29
3.2.2 *Game Objects* 30
3.2.3 Sistema de física 31
3.2.4 Sistema de renderização 33
3.2.4.1 Materiais 34
3.2.4.2 *Shaders* 34
3.3 ARQUITETURA 35
3.3.1 Máquina de estados 35
3.3.1.1 Estados 36
3.3.1.2 Sistema gerenciador de estados 37
3.3.2 Controladores 37
3.4 CONVENÇÕES DE NOMENCLATURA 39
3.4.1 Objetos 39

3.4.2 Controladores 39
3.4.3 Estados 40
3.4.4 Mapas 40
3.4.5 Colisores 40
3.5 CONSTRUÇÃO DE FERRAMENTAS PRÓPRIAS 41
3.5.1 Sistema de eventos 41
3.5.2 Ferramenta para o controle de múltiplas *viewports* 45
3.5.2.1 *Main Camera* 46
3.5.2.2 *Secondary Camera* 47
3.5.3 Ferramenta para a criação e manipulação de grafos 49
4 CONCLUSÃO 51
4.1 TRABALHOS FUTUROS 51
5 REFERÊNCIAS 53
APÊNDICE A – Artigo 57

1 INTRODUÇÃO

1.1 CONTEXTUALIZAÇÃO

Dada a grande demanda de jogos digitais no Brasil e a pouca produtividade do mesmo, é possível encontrar um grande mercado pouco explorado, para que esse mercado seja explorado é preciso aproximar o desenvolvimento de jogos aos brasileiros.

Uma parte considerável dos lucros da indústria de *softwares* é destinado ao jogos eletrônicos, consequentemente parte dos profissionais formados nas áreas de TI acabam por trabalhar com jogos (WASHBURN 1° autor et al.,2016). Com este mercado em crescimento as universidades brasileiras devem se adaptar para poder preparar novos profissionais. Este trabalho irá demonstrar o desenvolvimento de um jogo por um ponto de vista acadêmico.

1.1.1 Definição de jogo eletrônico

Para iniciar este trabalho será feito primeiro uma análise sobre a definição de um jogo digital.

De acordo com Marston e Hall (2016) um jogo pode ser considerado um programa que permite que um ou mais jogadores interajam com o mesmo com a finalidade de entretenimento.

De maneira diferente Vu e Gaskill (2016) define de uma maneira mais abrangente que um jogo pode ser definido como um programa onde um usuário controla imagens em uma televisão ou monitor.

Mesmo os conceitos mais específicos deixam aberto uma gama abrangente de interpretações, para uma melhor visualização da definição de um jogo e para embasar a produção deste, será feita a análise de jogos disponíveis no mercado.

Dentre os vários gêneros de jogos no mercado, existe uma categoria específica onde o escopo de produção se encaixa em um trabalho de conclusão de curso, os *indie games*.

A definição de um jogo independente não é clara, para analisar melhor o conceito de *indie game* foram escolhidos alguns jogos de sucesso dessa categoria. Para a sua escolha foram considerados os seguintes pontos importantes que permitissem uma analogia da produção com a construção deste trabalho: terem sido produzidos por uma equipe pequena, possuir um investimento inicial próximo de zero e terem sido

bem sucedidos como jogos através de premiações e vendas.

1.1.1.1 Braid

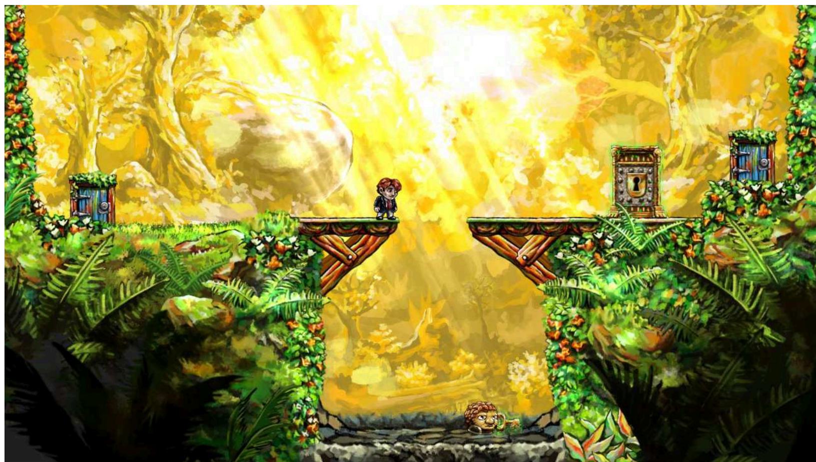

Figura 1 – *Screenshot* do jogo Braid.

Braid foi produzido por duas pessoas, teve um lucro total acima de 6 milhões de dólares e recebeu inúmeras premiações. Ele é um jogo 2D do estilo plataforma, onde o foco principal do jogo é a solução de *puzzles* utilizando a habilidade de voltar no tempo. A experiência do jogo se firma sobre a história que se desenrola durante o jogo.

1.1.1.2 Limbo

Figura 2 – *Screenshot* do jogo Limbo.

Limbo teve como seu ponto principal o estilo artístico único e inovador, este de acordo com Washburn (2016) é uma das principais características de jogos que tiveram sucesso. Limbo teve uma equipe com grande rotação, tendo poucos membros fixos. Ele também é um jogo 2D e o seu *core loop* se resume à solução de puzzles onde o jogador arrasta e posiciona objetos para avançar e sutis avanços na história.

1.1.1.3 Super meat boy

Novamente este foi um jogo feito por apenas duas pessoas, diferente dos jogos anteriores *Super meat boy* não possui um estilo artístico único. O diferencial está na jogabilidade, este é um jogo 2D que desafia o jogador a pular e desviar de vários obstáculos que matam o personagem ao menor contato.

1.1.2 Definição da equipe

Seguindo as características mais comuns entre os jogos *indies* de sucesso, será feito um jogo 2D, com foco em história e a solução de

puzzles, ambos serão explorados com elementos do jogo e com diálogos de múltiplas escolhas.

Este trabalho é realizado em conjunto com um grupo de pesquisa multidisciplinar da UFSC o G2E (grupo de educação e entretenimento), que ajudarão a suprir as necessidades da produção de um jogo, nesta equipe se encontram dois ilustradores, um para cenários e outro para personagens, um game designers que é responsável por projetar o jogo, dois músicos e dois programadores, com esta equipe é possível cobrir todos os pontos necessários para a criação de um jogo semelhante aos jogos selecionados como referência.

O G2E têm como objetivo criar e desenvolver projectos na área de entretenimento sob a ótica da gestão do design, desenvolvendo produtos orientados ao mercado para o exercício e aprendizado teórico-prático de seus integrantes. Atualmente vem se especializando em projetos de franquias transmídia, criando e desenvolvendo games, animação, quadrinhos, livros, audiobooks, trilha sonora, brinquedos, jogos analógicos, experiências em realidade virtual, entre outros.

1.2 OBJETIVOS

1.2.1 Objetivo Geral

Este trabalho tem como objetivo geral demonstrar o processo de desenvolvimento de um jogo com características semelhantes aos do mercado atual. Também como apresentar a aplicação dos conhecimentos obtidos no curso de graduação de ciência da computação no desenvolvimento do mesmo.

1.2.2 Objetivos Específicos

- Desenvolver um sistema de múltiplas *viewports*, com uma interface que permita maiores elementos cinematográficos.

Figura 3 – Exemplo de efeito *split screen*

- Desenvolver sistema de eventos que permite o level designer montar a interação entre objetos no jogo apenas usando uma interface gráfica.
- Criar uma arquitetura e implementar um gerenciador geral do jogo, capaz de manter registro do estado do jogo, itens e eventos.
- Desenvolver um sistema de criação de grafos e controle de grafos

que possua uma interface gráfica, este sistema irá ser responsável por definir os caminhos que o jogador e a inteligência artificial poderão percorrer.

- Desenvolver as inteligências artificiais do inimigos e dos companheiros.
- Desenvolver um sistema de diálogos, com múltiplas escolhas. Que irá carregar a partir do disco os diálogo, opções e consequências de cada escolha.
- Criar *shaders* para objetos e efeitos de câmera.

2 REVISÃO BIBLIOGRÁFICA

Dada a natureza pouco ortodoxa deste trabalho, foram realizadas buscas com o objetivo de fortalecer a justificativa e auxiliar o desenvolvimento do mesmo.

Foi utilizado o *Google Scholar* como ferramenta de busca principal. A *string* de busca : (*"software engineering"AND "game development"AND "tools"*) foi executada retornando um total de 4390 resultados. Para uma melhor seleção dos resultados, foi feita a ordenação por relevância e os primeiros artigos que melhor satisfizeram os objetivos citados foram escolhidos.

2.1 RESUMOS

2.1.1 Game Development: Harder Than You Think

Com o avanço tecnológico o foco de um desenvolvedor de jogos mudou, antes a maior preocupação era com a otimização, agora os complexos designs tornam a própria concepção do código um desafio.

Este artigo analisa por dois pontos de vista as maiores dificuldades no desenvolvimento de jogos na atualidade.

O primeiro ponto de vista analisado é em relação ao tamanho e a complexidade de um projeto. As áreas de conhecimento necessárias para a produção de um jogo tiveram um grande acréscimo em quantidade e complexidade. Infelizmente este crescimento não foi acompanhado pelas ferramentas de desenvolvimento, um exemplo disto é a ausência de uma IDE em C++ focada no desenvolvimento de jogos. Além disto é apontado um problema no *workflow*, onde em projetos maiores e mais complexos o tempo para gerar um *build* e testar é muito maior comparado a outros tipos de *softwares*. Este problema de *workflow* também é multiplicado quando se deseja distribuir o jogo em múltiplas plataformas.

O segundo ponto de vista é em relação ao grande domínio exigido em múltiplas áreas de conhecimento, são citadas áreas de matemática, algoritmos e conhecimentos técnicos para assimilar toda a estrutura necessária para rodar um jogo.

Este artigo mostra uma clara deficiência que está se criando na indústria de jogos, deficiência que pode ser reparada com uma aproximação do meio acadêmico à esta indústria.

2.1.2 The whats and the whys of games and software engineering

O objetivo deste trabalho é apresentar a intersecção entre o desenvolvimento de jogos e a engenharia de software, apresentando possibilidades de crescimento mútuo em ambas as áreas.

O primeiro problema apresentado é no processo de desenvolvimento, na indústria de jogos esta área sofre grande falha nas configurações das *pipelines* de desenvolvimento e nas relações interpessoais da equipe. É apontado que a indústria de *software* já passou e ainda passa por problemas semelhantes, existe muito a se aprender com a mesma, um exemplo disso é como o SCRUM vem sendo absorvido pela indústria de jogos.

A falta de ferramentas que ajudam no desenvolvimento de jogos, quando solucionada, poderia também ajudar na própria indústria de softwares, com ferramentas que permitiriam uma análise mais profunda dos ambientes desenvolvidos.

A aplicação de *design patterns*, muitas vezes a indústria de jogos assimila e adapta um design já existente para aplicá-lo em jogos, o processo contrário também poderia ser explorado, já que muitas vezes os focos dos são requisitos das aplicações.

Sistemas emergentes são aqueles sistemas onde não é possível prever a saída do mesmo, dificultando a aplicação de testes automatizados. Seguindo esta definição a grande maioria dos jogos são sistema emergentes e já desenvolvem técnicas para contornar este problema.

O inverso pode ser aplicado para teste de interface, onde a indústria de jogos possui muito no que aprender com a engenharia de software.

2.1.3 What Went Right and What Went Wrong: An Analysis of 155 Postmortems from Game Development

Postmortem é um documento escrito após a produção de um jogo, com objetivo de mostrar os principais acertos e erros e auxiliar futuros projetos.

Este artigo analisa 155 *postmortems* e apresenta os pontos positivos e negativos mais ocorrentes no desenvolvimento destes jogos. Devido à grande quantidade de características analisadas, foram filtradas as mais relevantes para este trabalho.

Apesar de ser um senso comum entre os desenvolvedores, é apontado a importância de uma arte única e atraente, este princípio já é

aplicado no *The Rotfather* em vista que já foi premiado no SBGames por sua arte.

A pouca ocorrência de testes acaba por resultar em projetos falhos, este é um ponto a ser considerado neste trabalho, já que não possui nenhum planejamento de QA.

O desenvolvimento de ferramentas é outro fator importante. Como será apresentado no capítulo 3, foram desenvolvidas várias ferramentas para facilitar a construção do *The Rotfather*.

Conseguir seguir um cronograma ou poder adicionar tempo em caso de atraso permitiu que as equipes pudessem ter mais controle sobre a produção. É importante notar que o produto final para os casos bem sucedidos não sofreu alterações mesmo com atrasos.

Obter *feedback* de jogadores durante o desenvolvimento resultou em grande mudanças nos jogos analisados, trazendo melhorias de maneira geral.

Ao fim do artigo é apresentado um série de ações para reduzir o risco de falhar, tais ações serão consideradas no desenvolvimento deste trabalho.

3 DESENVOLVIMENTO

3.1 ESCOLHA DE FERRAMENTAS

Mcsaffry e Graham (2012) nomeiam como camada de aplicação todo tipo de programa que é responsável por estabelecer a comunicação entre o jogo e o computador que está sendo usado, podendo variar de um *pc windows* à um *playstation 4* ou um celular. Esta camada possui grande responsabilidade para o funcionamento do jogo e engloba várias funções.

Como visto anteriormente para que um jogo funcione é necessário que de alguma forma o jogador consiga inserir informações para o jogo, como o movimento de um mouse ou os botões de um teclado. A camada de aplicação fica responsável por lidar com as diferenças entre as plataformas rodadas e a unificação entre os comandos, para que o jogo interprete todos da mesma maneira.

Um jogo normalmente precisa simular todos os objetos que estão sendo apresentados ao jogador, este processo por sua vez consome muita memória, ele também carrega em tempo de execução novos *assets* e descarregá-os quando não necessários. Tamanha demanda por memória acaba por elevar a importância das linguagens e APIs escolhidas para o jogo.

A simulação dos objetos considera sempre o tempo entre ciclos para calcular suas ações, de modo que computadores com *hardware* diferentes e *clocks* diferentes possam apresentar sempre as mesmas coisas dado os mesmo *inputs*, mantendo assim uma coerência entre os objetos.

Estes pontos apresentados são alguns dentre vários que a camada de aplicação fica responsável, diante disso é essencial que para o desenvolvimento de um jogo, se saiba escolher e aplicar as melhores ferramentas para as necessidades do jogo desenvolvido. Este capítulo irá apresentar as ferramentas mais populares no mercado e avaliará suas aplicações no jogo.

3.1.1 Bibliotecas gráficas

As bibliotecas gráficas são responsáveis por apresentar toda a visão do jogo ao jogador. Este processo pode ou não envolver as placas gráficas e muitas vezes é utilizado na apresentação de jogos 3D. Elas permitem que a aplicação se comunique com o SO ou *hardware*, dando

acesso e liberdade para a aplicação apresentar imagens ao usuário.

Existem várias bibliotecas gráficas no mercado, porém duas delas se destacam por popularidade e performance. A Direct3D (integrada na DirectX) é uma API exclusiva da Microsift que dominou por muitos anos o mercado formando um monopólio dos jogos com os SOs da Microsoft.

A OpenGL apesar de existir desde 1992 está apenas agora recuperando espaço no mercado sobre a DirectX mas apresenta um futuro promissor já que as novas tecnologias que surge estão utilizando-a.

Ambas APIs permitem a programação de jogos, mas são relativamente complexas de se utilizar diretamente. Como será visto mais a frente existem *Game Engines* que facilitam essa comunicação.

3.1.2 Motores de física

Os motores de física ou *physics engines* permitem simulações físicas sobre um grupo de objetos. Os jogos que utilizam alguma forma de cálculo físico acabam por utilizar um desses motores. Mesmo em jogos menores que não exista tanto cálculo a ser feito, é ainda recomendado o uso desses motores, pois o cálculo por eles aplicado já sofreu várias otimização e estão sempre em constante melhoria.

Existem motores de física para várias situações diferentes, cada qual com seus pontos fortes e fracos. Diferente das duas APIs gráficas apresentadas, eles são de fácil uso e para cada plataforma existe uma variedade diferente.

3.1.3 *Game engines*

Foram apresentados dois tipos de bibliotecas que auxiliam o desenvolvimento de jogos, mas existem outros tipos, como as *engines* de áudio, gerência de memória, comunicação em rede, uso de periféricos entre outros.

Ao se desenvolver um jogo, serão escolhidas algumas dessas ferramentas que serão ainda integradas. Visando poupar os desenvolvedores de passar por esse processo, surgiram as *Game engines*. Elas agrupam várias funcionalidades de outras bibliotecas em uma mesma *engine*.

É possível encontrar *Game engines* para vários tipos de plataformas e em várias linguagens. De acordo com uma pesquisa realizada pelo site thenextweb existem duas *engines* que dominam o mercado:

- Unreal Engine

  A Unreal é uma engine desenvolvida pela Epic Games, teve sua estréia com o jogo Unreal. O foco dela é em jogos 3D com alto processamento gráfico, por estas razões sempre foi a favorita pelas grande empresas de jogos. Atualmente ela utiliza C++ como linguagem de programação, facilitando seu uso, já que a grande maioria dos desenvolvedores utilizam C++ por permitir uma melhor otimização do código.

- Unity Engine

  A Unity foi lançada em 2005 e era exclusiva para OS X. Por ser uma engine mais nova e menos poderosa em relação a seus competidores(Unreal, CryEngine), possuía uma curva de aprendizado menor, facilitando a entrada de novos desenvolvedores no mercado. Como já visto anteriormente, a maioria dos jogos indies acabam por serem feitos em 2D, já que 3D consome muito tempo e trabalho, se tornando inviável para pequenos grupos sem financiamento. Tendo este como seu maior cliente a Unity foi se moldando aos poucos, sua versão 4 foi a primeira a vir com vários suportes para jogos 2D e atualmente ela é uma engine recomendada tanto para jogos 2D quanto 3D.

Como este trabalho será um jogo 2D, a Unity se torna a melhor opção. Além de já possuir várias ferramentas para o desenvolvimento 2D, também da suporte para a criação de *shaders*.

3.2 ESTRUTURA DA UNITY

Primariamente a Unity utiliza C# como sua principal linguagem de programação, sendo ela uma linguagem que aceita o paradigma orientado a objetos permite que os jogos dentro dela sejam programados utilizando este paradigma, porém, ela possui uma API que aprimora a estrutura básica disponível pelo C#, permitindo o uso do paradigma orientado a componentes.

3.2.1 Programação Orientada a Componentes

Programação orientada a componentes permite que programas sejam construídos a partir de componentes de *software* pré-definidos, que são reutilizáveis e independem um do outro. Estes componentes

devem seguir alguns padrões predefinidos incluindo interface, conexões, versões e entregas.(WANG e QIAN, 2005).

Dentro da Unity estes componentes devem herdar de um *MonoBehaviour* que é responsável por implementar todas as comunicações entre os componentes e inserir o componente dentro do ciclo de vida da Unity. A estrutura da Unity se resume de forma superficial à uma simulação de uma lista de *GameObjects*, de forma que a cada *frame* ou *loop* cada um destes objetos realizarão uma sequência de operações.

A figura 4 mostra a sequência que os métodos de um *MonoBehaviour* são chamados, para cada um desses métodos a Unity irá percorrer a sua lista de *GameObjects* e realizar a chamada do respectivo método. Apesar da grande quantidade de métodos normalmente são utilizados principalmente dois deles, que já são implementados ao se criar uma classe de componentes dentro da Unity, o método *Start* e *Update.* O *Start* como pode ser visto na figura 4 é chamado apenas uma vez, quando é criado, enquanto o *Update* é chamado uma vez a cada ciclo.

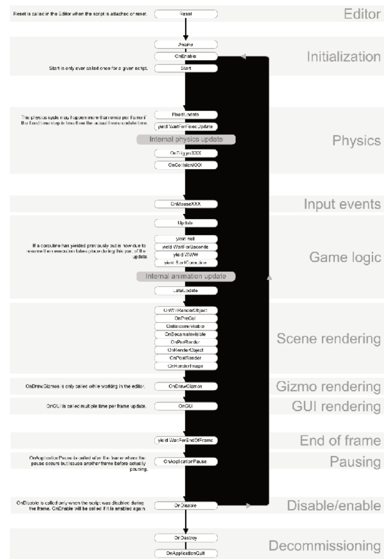

Figura 4 Ciclo de vida da Unity

3.2.2 *Game Objects*

Como apontado anteriormente, a Unity simula uma lista de *Game Objects* que representam todos os objetos dentro do jogo. Por definição qualquer *Game Objects* deve possui o componente *transform*, responsável por representar o objeto no espaço tridimensional e também aplicar as transformações de translação, rotação e escala sobre sua matriz.

A figura 5 mostra a representação de um *Game Objects* pela interface da Unity. A partir dele é possível então implementar os componentes necessários para a estrutura do jogo. A Unity apresenta uma grande variedade de componentes já implementados para agilizar o desenvolvimento. Dentre esses componentes existem dois sistemas muito utilizados durante o desenvolvimento de um jogo: os sistemas de física e renderização. Ambos são separados em duas categorias, os sistemas bidimensionais e os sistemas tridimensionais, neste trabalho foi utilizado apenas os sistema bidimensionais já que o jogo implementado é 2D.

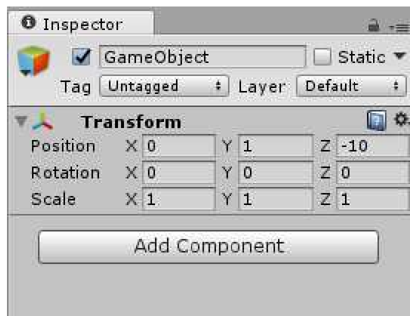

Figura 5 – GameObject vazio.

3.2.3 Sistema de física

A Unity possui um completo sistema de física para jogos, capaz de simular aceleração, colisão e aplicação de forças de maneira realistas. Para que um *Game Objects* esteja incluso dentro da simulação física, ele deve possuir o componente de *Rigidbody2D* como apresentado na figura 6.

O objeto então recebe os efeitos de todos os tipo de força dentro da cena, como gravidade, forças do vento ou até explosões, porém, ele ainda não irá colidir com outros objetos, para isso existe uma hierarquia de componentes colisores.

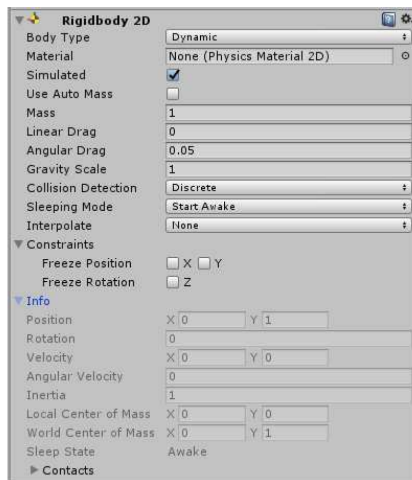

Figura 6 – Componente *Rigidbody2D*.

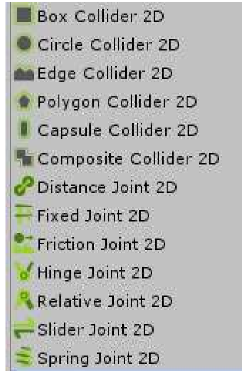

Figura 7 – Opções de colisores e comportamentos.

A figura 7 mostra as opções de colisores que existem para um objeto, um mesmo objeto pode ter um ou mais colisores, cada qual com sua forma. Também é possível adicionar aos colisores efeitos e comportamentos aos objetos, sendo possível simular um carro e suas rodas com facilidade com apenas a API física.

3.2.4 Sistema de renderização

Para a renderização em 2D a Unity utiliza apenas o componente *Sprite Renderer*, este componente utiliza um plano para se representar dentro do espaço 3D e as configurações fornecidas são aplicadas sobre um *Shader* que é aplicado no plano.

Como apresentado na figura 8, o *Sprite Renderer* permite a configuração de alguns atributos relevantes para a renderização, a imagem que é apresentada, a cor que irá ser aplicada através de um produto escalar entre a cor da textura e a cor atual, a ordem que será desenhado na tela e o material aplicado no plano.

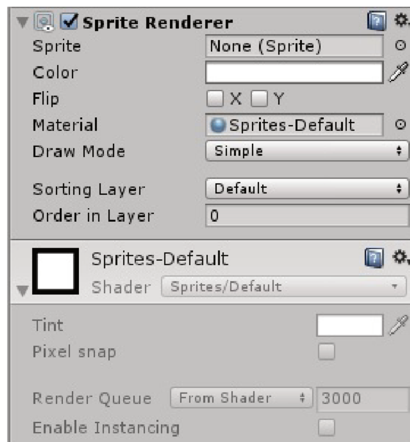

Figura 8 Componente *Sprite Renderer*.

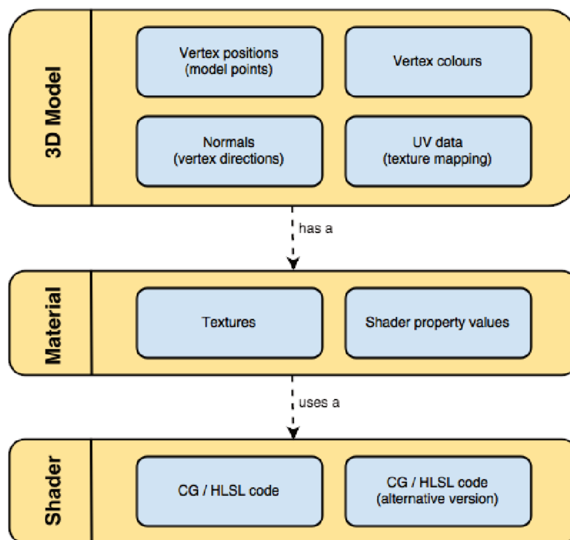

Figura 9 Diagrama do fluxo de renderização.

3.2.4.1 Materiais

Os materiais da Unity nada mais são que uma representação dos *Shaders* configurados previamente, na figura 8 é possível ver que o material configurado é o *Sprite-Default*, logo abaixo do componente é possível ver uma representação do material, por ele ser um material pré-definido não é possível alterá-lo, porém, é possível utilizar diversos materiais para a representação de *Sprites*, desde que eles possuam a implementação dos atributos previamente citados.

3.2.4.2 *Shaders*

A implementação dos *Shaders* utiliza a linguagem *ShaderLab* que foi criada especificamente para a Unity, ela encapsula códigos de HLSL/CG permitindo configurar o funcionamento do shader e também a implementação de substitutos para o caso do hardware não ter capacidade de rodar os códigos passados.

Existem dois tipos predominantes de *Shaders*, *vertex fragment* e *surface*. O *vertex fragment* permite total liberdade ao programador, podendo alterar atributos do vértice à vontade. O *surface shader* por outro lado já aplica uma resultando na cor dos vértices em relação à normal e a iluminação.

Como o *The Rotfather* possui foco na narração é importante que a ambientação seja imersiva, de modo que para utilizar as vantagens da iluminação é melhor utilizar um *surface shader*, para isso é possível utilizar um *pre-build shader* oferecido pela Unity, o *Sprite-Diffuse*. A implementação deste *shader* é bem simples, ele apenas pega a cor do vértice de acordo com uma mapa UV de textura e aplica uma cor resultante através de um produto escalar. A Unity então calcula a cor dos vértices baseados na normal e na iluminação da cena.

```
1 void Surf(Input IN, inout SurfaceOutput o)
2 {
3   fixed4 c = SampleSpriteTexture(IN.uv_MainTex) * IN.color;
4   o.Albedo = c.rgb * c.a;
5   o.Alpha = c.a;
6 }
```

Listing 3.1 – Algoritmo do *shader sprite-diffuse*.

3.3 ARQUITETURA

Para a programação dos objetos na Unity é necessário a criação de componentes que serão adicionados à eles. A comunicação entre componentes ocorre através da referencia do objeto e o método de busca **GetComponent<T>()**, que pertence aos objetos do tipo *MonoBehaviour*.

Apesar de útil este método é custoso e portanto costuma-se utilizá-lo na inicialização do objeto. Como mostrado anteriormente no ciclo de vida da Unity existem dois métodos para a inicialização dos objetos, **Awake()** e **Start()**, ambos são chamados apenas uma vez logo após a inicialização, sendo que o método **Awake()** é chamado primeiro.

Dada esta ordem de procedência foi escolhido o método **Awake()** para a inicialização e busca de referências, e o método **Start()** para o uso destas referências e configurações dos objetos. Deste modo é possível evitar a ocorrência de erros por referências nulas.

Visando facilitar a legibilidade do código e a modularização do mesmo, foi criada uma arquitetura base dos componentes, que buscam as referências dos componentes mais utilizados.

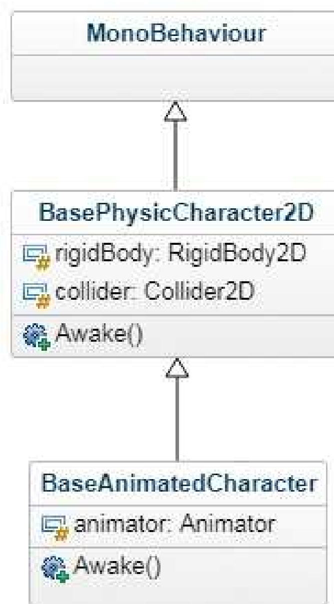

Figura 10 – Arquitetura base.

3.3.1 Máquina de estados

Nos jogos digitais o uso de máquinas de estado é muito comum, cada ação do jogador gera inúmeras transições de estados para cada objeto, por exemplo se uma parede é destruída ele teve uma alteração de estado, seus colisores foram alterados e ela apresentou uma animação, se

um objeto do chão é carregado pelo jogador ele não pode ser carregado novamente e deve estar na mão do jogador. O personagem principal é quem geralmente acaba por ter o maior número de estados, já que cada *input* acaba por realizar uma alteração no estado do jogo e cada estado possui uma regra diferente e se comporta de maneira diferente em relação aos *inputs*.

Como a programação na Unity utiliza o paradigma de orientação à componentes, grande parte dos *Design Patterns* da orientação à objetos não podem ser aplicados diretamente, eles devem ser adaptados à estrutura da Unity e ao paradigma. No caso da máquina de estados foi utilizada uma versão já adaptada para Unity e em cima dela foram feitas alterações para otimizar e facilitar o uso da ferramenta.

Esta estrutura faz o uso de duas **Enums**, uma para representar os estados e outra para representar as transições entre os estados, a implementação dos estados ocorre em uma classe referente àquele estado e que herda de uma classe base genérica para todos os estados, a gerência dos estados ocorre por uma outra classe também genérica que realiza o processo de transição e atualização dos estados.

3.3.1.1 Estados

Cada estado de um objeto é independente do outro, porém, todos eles herdam da classe base **FSMState** que é genérica, de modo que o estado sempre possa usar a referencia de seu controlador base.

Os estados estão fora da estrutura de componentes da Unity mas possuem sempre a referencia ao componente que os gerencia. Um estado possui como atributos além da referencia ao componente, um mapa entre as transições e os estados, ambos em formas de **Enum** e por último uma referencia à **Enum** de estados para identificar qual estado esta classe representa.

Os métodos responsáveis por configurar as transições do objeto permitem a adição e remoção de transições e informam qual estado resulta de uma dada transição. Estes métodos são apenas utilizados para a configuração dos estados que ocorre na inicialização dos objetos. Para o controle do objeto existem três métodos, um chamado quando ocorre uma transição para este estado, outro quando ocorre uma transição deste estado e por último um método de *update*, que é chamado a cada frame e permite que ações contínuas sejam realizadas no objeto. Qualquer um deste métodos pode realizar uma chamada à máquina de estados para realizar uma transição, porém, existe um

outro método apenas para conferir se alguma transição deve ocorrer, este método é chamado também a cada *update* e é abstrato na classe base, obrigando a implementação em todos os estados.

3.3.1.2 Sistema gerenciador de estados

Como falado anteriormente os estados são independentes, de modo que um estado não possui a referencia para outro. Eles utilizam apenas **Enums** para realizarem as transições, logo para que a máquina de estados funcione é necessário um sistema responsável por conectar e gerenciar estes estados.

Para guardar as referências e acessá-las rapidamente é utilizado um mapa entre a **Enum** de estados e o estado, além do mapa, existe uma referencia ao estado atual que é atualizado a cada frame.

Esse sistema permite então a inserção e remoção de estado e a transição entre os estados. Durante a transição ele irá através do mapa do estado atual, verificar se é possível realizar a transição desejada, caso possível, irá fazer a chamada para o método de saída do estado atual, atualizar o estado atual e fazer a chamada para o método de entrada.

A construção desta máquina e a conexão dela com a estrutura da *Unity* ocorre por meio de um controlador que irá montar os estados e serializar o dados necessários para a máquina construída, permitindo que o *game designer* possa configurá-la pela interface da Unity.

3.3.2 Controladores

Um controlador é responsável por criar os estados, armazenar os atributos que serão utilizados pelos estados e atualizar a máquina de estados, é através dele também que outros objetos aplicam alterações na máquina de estados pelos atributos, por convenção estes atributos são criados dentro de classes que agrupam e organizam a interface da *Unity* e são chamados de *component.*

Por ser uma peça central e precisar manter a referencia para a maior parte dos atributos pertencentes ao objeto, o controlador herda da classe **BaseAnimatedCharacter**, que já resgata os atributos responsáveis pela animação e simulação física.

O controlador base é abstrato e obriga a implementação do método **MakeFSM**, responsável por configurar os estados e suas transições, mostrado abaixo.

```
public override void MakeFSM() {
    IdleAICompanionExplorationState idle = new
    IdleAICompanionExplorationState(this);

    idle.AddTransition(ExplorationTransitionMap.Walking
    , ExplorationStateMap.Walk);

    idle.AddTransition(ExplorationTransitionMap.
    DialogueIdling , ExplorationStateMap.DialogueIdle);

    WalkAICompanionExplorationState walk = new
    WalkAICompanionExplorationState(this);

    walk.AddTransition(ExplorationTransitionMap.Idling ,
     ExplorationStateMap.Idle);

    walk.AddTransition(ExplorationTransitionMap.
    DialogueWalking , ExplorationStateMap.DialogueWalk);

    this.StateSystem.AddState(idle);
    this.StateSystem.AddState(walk);
}
```

Listing 3.2 – Configurando uma máquina de estados

Com este controlador e a máquina de estados é possível ter a estrutura base para implementação do personagem principal e IAs necessárias ao jogo, para objetos simples que não possuem grandes variações de estados é utilizado apenas as classes bases sem controladores. Na figura 12 é possível ver um diagrama de classes referente à estrutura montada entre o controlador do *Player*, a máquina de estados e as classes base.

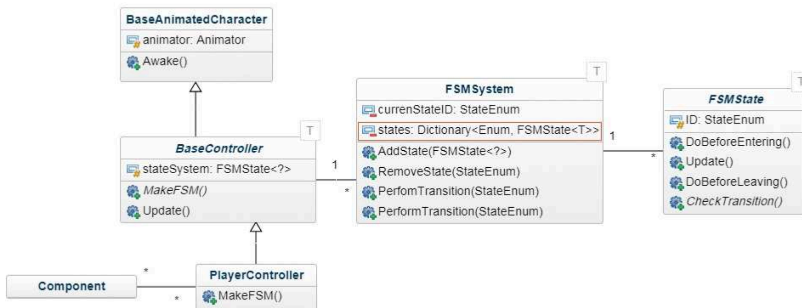

Figura 11 – Diagrama de classe do controlador.

3.4 CONVENÇÕES DE NOMENCLATURA

Por este projeto envolver pessoas e organizações de outros países foi optado por se utilizar apenas a língua inglesa.

Com a estrutura base de funcionamento é possível a visualização da maior parte dos objetos e classes que serão criados, com isso em mente foi definida uma estrutura de nomenclatura para as classes e objetos dentro da *Unity.*

Inicialmente foram criadas definições que abrangessem todas as necessidades do jogo, abaixo é mostrado uma forma simplificada dela, que serve de guia para as convenções.

- *Object : Enemy, Boss, NPC, Player, AI, Item, etc.*
- *State Action: Idle, Walk, Jump, etc.*
- *Game Mode: Exploration, Shooter, Melee.*
- *Map Type: State, Transition, Animation, Layer, etc.*
- *AI Behaviour Type: Standard, Aggressive, Passive, etc.*

3.4.1 Objetos

Objetos englobam qualquer *GameObject* que possa ser representado visualmente no jogo, podendo ser um inimigo, NPC, o jogador, um item, etc.

Cada classe referente ao objeto deve possuir uma característica em seu nome que seja clara o bastante para o diferenciar de todos os outros objetos. Deve possuir também um sufixo que identifique o seu tipo de objeto. Se a classe não for direcionada para um tipo específico deve possui o prefixo *Base.*

Expressão : "Base"? + Caracteristica + **Object**

Exemplo: RedNPC, FireWatcherBoss, BlueGhostEnemy.

3.4.2 Controladores

Os controladores devem seguir as mesmas regras do objetos, porém devem adicionar o sufixo *controler.*

Expressão : "Base"? + Caracteristica + **Object** + "Controller"

Exemplo: RedNPCController, FireWatcherBossController, BlueGhostEnemyController.

3.4.3 Estados

Um estado deve mostrar qual o objeto que ele representa caso necessário deve adicionar alguma característica, indicar qual ação ele representa, o modo do jogo e deve apresentar o sufixo *State*.

Expressão : Caracteristica? + **Object**?+**State Action**+**Game Mode**+"State".

Exemplo: PlayerIdleExplorationState, FredWalkExplorationState.

3.4.4 Mapas

Devido ao grande número de *Assets* no jogo, são utilizadas muitas informações, como nome de animações, imagens ou sons. Para organizar e facilitar alterações e buscas, mapas são criados para a medida que surge a necessidade. As **Enums** dos estados e das transições também são consideradas mapas e devem seguir as mesmas regras.

Todo mapa deve indicar o objeto e o modo do jogo casa necessário, deve mostrar o que ele representa e deve ter o sufixo *Map*.

Expressão : Caracteristica? + **Object**?+**Game Mode**?+**Map Type**+"Map".

Exemplo: PlayerMeleeStateMap, InputMap, ExplorationAnimationMap.

3.4.5 Colisores

Para objetos simples que possuem apenas a função de detectar algum tipo de colisão ou barrar o trajeto de algum objeto é necessario apresentar uma caracteristica que identifique qual objeto ele representa e deve possuir o sufixo *ColliderController*.

Expressão : Caracteristica+ **Object**?+"ColliderController".

Exemplo: BlueBoxColliderController, PathNodeColliderController.

3.5 CONSTRUÇÃO DE FERRAMENTAS PRÓPRIAS

Mesmo com uma *game engine*, falta a implementação do *game core*, onde toda a estrutura do jogo ainda deve ser montada. Para que seja possível a criação de novos conteúdos para o jogo e que a manutenção no mesmo seja fácil, deve ser planejado um conjunto de ferramentas internas do jogo, que permitirão que os responsáveis por montar o *level* possam fazê-lo sem necessitar tocar em uma linha do código. Estas ferramentas devem dar o máximo de liberdade possível para o *game designer* e devem estar bem estruturadas para o caso de mudanças serem necessárias.

Esta seção apresentará as ferramentas desenvolvidas para o *The Rotfather*, tais ferramentas foram montadas de forma modular, podendo ser aplicadas em outros jogos.

3.5.1 Sistema de eventos

O objetivo do sistema de eventos é permitir que o *level designer* possa criar ações e reações de acordo com situações que ocorram dentro do jogo sem necessitar programar.

Para isso foi utilizada a estrutura básica de animação da *Unity*. O sistema de animação permite que qualquer atributo serializável de um componente possa variar de acordo com o tempo, permitindo a alteração de variáveis de todos os tipos como apresentado na figura 13.

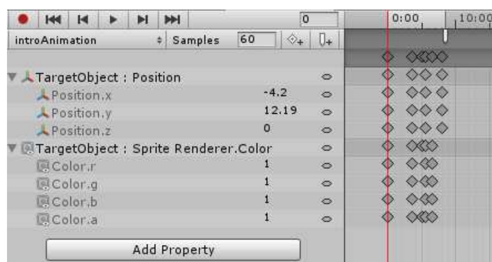

Figura 12 – Interface da ferramenta de animação

Esta ferramenta de animação já permite grande liberdade ao level designer para reproduzir ações dentro do jogo, porém, é ainda necessário os gatilhos para essas animações. O sistema de eventos permite então que dada uma situação, uma lista de objetos sofram um sequência de reações.

Para construir esta ferramenta foi então necessário definir as formas de gatilho dos eventos, para este trabalho foram utilizadas apenas

colisões, porém, poderia se ter utilizado outras formas de evento, como a vida do jogador ou algum padrão de *Input* do jogador.

Os gatilhos são representados por uma **Enum InteractiveActionMap**, que lista as possíveis formas de detecção de ações. As ações são definidas em uma classe *singleton* nomeada de **Action**, nesta classe existe uma **struct** com as funções que podem detectar colisão:

```
1  public struct ActionFunctions
2  {
3      public Func<GameObject , bool> onTriggerEnter;
4      public Func<GameObject , bool> onCollisionEnter;
5  }
```

Listing 3.3 – **Struct** com as funções de detecção da *Unity*

A definição destas funções ocorrem na inicialização da **Action**, que por ser um *singleton* ocorrerá apenas uma vez. No construtor é percorrido todos os elementos do **InteractiveActionMap**, para cada elemento é criada uma nova instância da **ActionFunctions** definido suas condições de acordo com o elemento atual da iteração, após definidas as funções elas são então salvas em um dicionário entre as **InteractiveActionMap** e a instância da **ActionFunctions**:

```
1  private Actions ()
2  {
3      actions = new Dictionary<InteractionActionMap ,
        ActionFunctions>();
4
5      foreach (InteractionActionMap action in Enum.GetValues(
        typeof(InteractionActionMap)))
6      {
7          actions.Add(action, this.GetActions(action));
8      }
9  }
10 private ActionFunctions GetActions (InteractionActionMap
        interaction)
11 {
12     ActionFunctions actions = new ActionFunctions();
13
14     actions.onTriggerEnter = gameObject => { return
        gameObject.tag.Equals(interaction.ToString())?true :
        false; };
15
16     actions.onCollisionEnter = gameObject => { return
        gameObject.tag.Equals(interaction.ToString()) ? true :
        false; };
17
18     return actions;
19 }
```

Listing 3.4 – Inicialização dos gatilhos das ações

Com as ações definidas, foi criado um componente **InteractiveObject** para definir as ações, conectar as funções de detecção com os componentes físicos do objeto e chamar uma lista de objetos que sofrerão reações da interação.

```
this.onTriggerEnter = Actions.Instance().ActionsDictionary[
    actionType].onTriggerEnter;
```

Listing 3.5 – Configurando os métodos de um **InteractiveObject**

Utilizando os próprios métodos da *Unity* é detectado a colisão e então validado pelo método definido pelas **Actions**

```
private void OnTriggerEnter2D (Collider2D collision)
{
    if (this.onTriggerEnter(collision.gameObject))
    {
        this.Interact();
    }
}
```

Listing 3.6 – Detecção da colisão pela *Unity* e validação pela **Action**

Ao detectar uma interação o **InteractiveObject** deve alertar a todos os objetos da lista de reações. Foi criado um componente **InteractionReactionObject** para definir e configurar as reações, este componente além de possuir uma animação que irá representar sua reação ele possui uma lista de efeitos que podem ser definidos para ocorrer no início ou ao final da animação da reação.

```
public void React ()
{
    this.reactinAnimationName = InteractionAnimationMap.
    REACTION;

    this.animator.Play(InteractionAnimationMap.REACTION);
    foreach (AbstractReaction reaction in this.preReactions)
    {
        reaction.React(this.gameObject);
    }
    this.OnReaction = true;
}
```

Listing 3.7 – Execução do início de uma reação

Para a definição destes efeitos foi criada uma classe abstrata **AbstractReaction** que possui um método abstrato para o início da reação, para o final da reação e para a inicialização da classe. Estes métodos são chamados pelo **InteractionReactionObject** de acordo com o tempo da sua animação de reação.

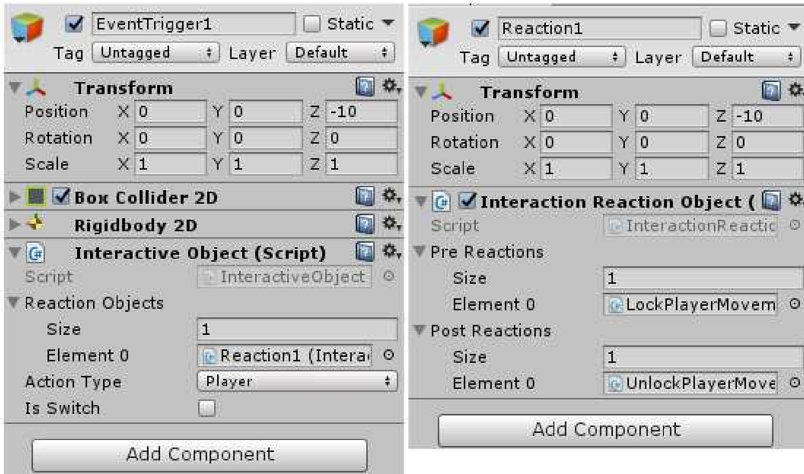

Figura 13 – Exemplo de uso do sistema dentro da *Unity*.

O *level designer* ao montar o level irá utilizar apenas dois componentes, **InteractiveObject** e **InteractionReactionObject** como apresentado na figura, para adicionar comportamentos que envolvam programação ele deve utilizar as **Reactions** que herdam da **AbstractReaction**, novas **Reactions** são implementadas a medida que surge a necessidade e a implementação de uma delas costuma ser pontual, não necessitando de mais de cinco linhas de código.

```
1 public class RemoveColliderReaction : AbstractReaction
2 {
3     public override void React (GameObject gameObject)
4     {
5         gameObject.GetComponent<Collider2D >().enabled =
      false;
6     }
7 }
```

Listing 3.8 – Exemplo da implementação de uma **Reaction**

3.5.2 Ferramenta para o controle de múltiplas *viewports*

Como o jogo é 2D, ele utiliza uma projeção ortográfica, onde a distância no eixo Z não interfere no tamanho da imagem que será apresentada na tela. Na figura 14 é possível perceber a diferença de profundidade, já na figura 15 é como se estivessem alinhadas, a diferença de uma para a outra é apenas o modo de projeção.

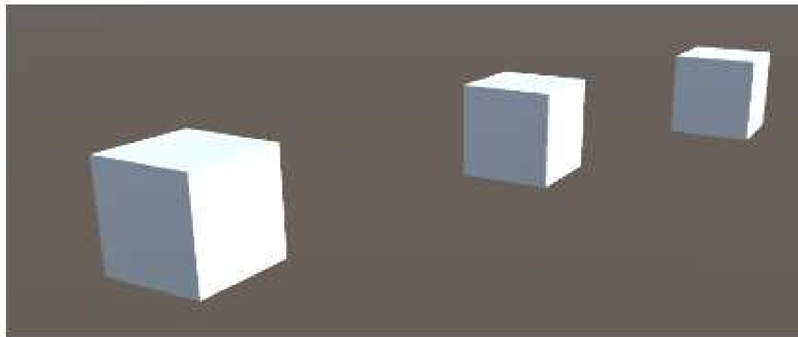

Figura 14 – Visão de uma câmera com perspectiva.

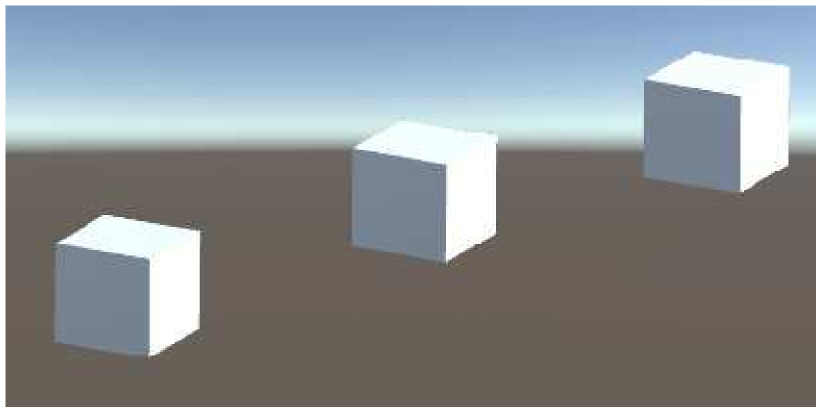

Figura 15 – Visão de uma câmera ortogonal.

Considerando uma proporção *pixel perfect*, onde um pixel em uma imagem equivale à um pixel na *viewport*, foram criados dois tipos de câmeras.

3.5.2.1 *Main Camera*

Esta câmera é a principal, sua *viewport* é a própria *window*, diferente das câmeras secundárias ela deve recalcular sua altura e largura de acordo com a resolução da *window*, de modo que ela sempre irá mostrar a mesmas imagens independente da resolução do jogador.

A resolução padrão para o jogo é de 1920x1080 pixels, uma proporção de 16:9, de modo que resoluções com proporções diferentes terão uma tarja preta que irá manter a imagem de apresentação igual à resolução padrão.

Na figura 16 é mostrado a visão do jogo na resolução esperada, já na figura 17 é utilizado uma resolução de 800x600 pixels com proporção 4:3. Como a proporção é maior no eixo y, se limita a imagem nos limites do eixo x e para não distorcer a imagem se aumenta a altura da câmera em y, para a imagem ficar de acordo com a medida definida pelo *game designer* a main camera coloca tarjas nos cantos superiores e inferiores.

Figura 16 – Monitor com resolução 1920x1080 pixels.

Figura 17 – Monitor com resolução 800x600 pixels.

3.5.2.2 *Secondary Camera*

A *secondary camera* é composta por dois objetos diferentes, o primeiro é a própria câmera que possui bordas, funções de *character follow* e cálculos de altura e largura em coordenadas de mundo, o segundo é a *viewport*, que irá aplicar sobre uma câmera definida na interface da ferramenta, transformações de acordo com a resolução da *window*, uma mesma *viewport* pode alternar entre várias câmeras.

A figura 18 apresenta um exemplo de uso de duas *secondary cameras*, para configurar estas câmeras os componentes de câmera e *viewport* foram configurados como se apresenta na figura 19.

O processo de configurar a câmera de acordo com a configuração da *viewport* exige uma sequência extensa de cálculos, como mostrado abaixo os valores de comprimento, largura e posição são calculados sem restrições e a cada etapa é limitados e acrescentado variáveis no cálculo.

```
1 public void LimitDimensions ()
2 {
3       this.UpdateValues ();
```

```
4         this.LimitingPositiveValues ();
5         this.LimitZoom ();
6         this.LimitHorizontalValues ();
7         this.LimitVerticalValues ();
8 }
```

Listing 3.9 – Sequência de métodos para o cálculo da *viewport*

Figura 18 – Exemplo de uso de múltiplas *viewports in-game*.

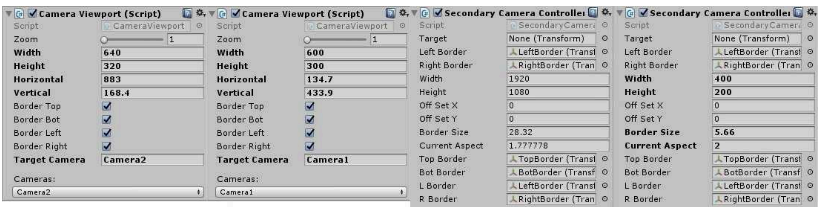

Figura 19 – Interface de configuração de duas *Viewports*.

3.5.3 Ferramenta para a criação e manipulação de grafos

Como o foco do jogo está nos diálogos e no desenvolver da história, o *design* do jogo limita as movimentações dos personagens em locais específicos. Desta maneira é possível saber onde o jogador irá e quais cenas ele verá, este é um padrão adotado pela maioria dos *adventure games*.

Para permitir a criação, manipulação e intersecção de caminhos, foi criada a ferramenta de *path plot*, que permite que o *level designer* crie nodos e conecte-o com outros nodos.

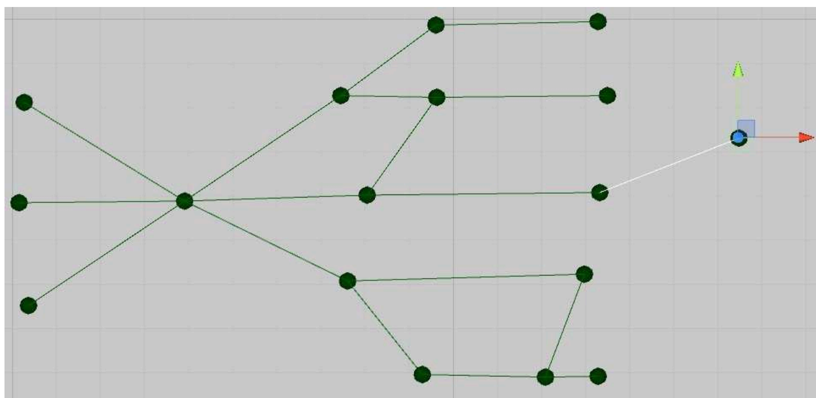

Figura 20 – Exemplo de grafo criado com a ferramenta *path plot*.

Como a principal plataforma que se pretende lançar o jogo é o PC, os controles básicos de movimentos devem considerar um teclado e mouse. A movimentação do personagem pelo teclado é limitada então em seis direções, resultado entre a combinação de direita e esquerda com cima, baixo e nada. Cada nodo possui um mapa onde a chave para ele é uma das seis direções e o valor corresponde ao nodo seguinte àquela direção.

O uso da ferramenta é simples, foi criado um atalho (ctrl+f) que cria um nodo, este nodo ainda não está posicionado e irá seguir a posição do mouse, por padrão ele irá considerar que está conectado ao nodo que estava selecionado quando o atalho foi pressionado, esta conexão é representada pela aresta branca. Quando pressionado, a ferramenta irá detectar a posição do nodo e os nodos vizinhos para definir uma chave, caso a posição já esteja ocupada ele irá destruir o novo nodo, caso o novo nodo esteja sobre um nodo já existente, em

vez de se criar um novo nodo é criada uma nova conexão entre o dois nodos antigos. As conexões criadas entre os nodos são representadas pelas arestas verdes, na figura 21 é possível ver a interface gerada em um nodo após ter todas suas direções preenchidas.

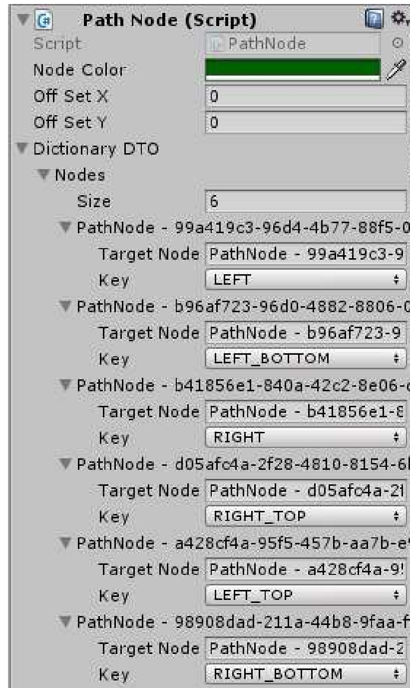

Figura 21 – Interface criada com a ferramenta *path plot*.

Todos os campos criados em objetos dentro da Unity têm seus valores serializados e salvos. Mas isso só ocorre com algumas variáveis básicas, para pode serializar o mapa de direções de cada nodo, foi criado um DTO, responsável por guardar o nome de cada nodo e sua respectiva chave, durante o processo de desserialização cada nodo presente no dicionário é buscado, utilizando o nome dele em formato de string, ao ser encontrado é inserido no dicionário com a chave relacionada ao nome.

4 CONCLUSÃO

A proposta deste trabalho é apresentar o processo e as técnicas aplicadas e desenvolvidas na construção de um jogo que fosse apto a ser comercializado.

Ao final deste trabalho foi construída uma *demo* no formato de um *vertical slice* onde todas as mecânicas e ferramentas inicialmente planejadas estivessem implementadas, permitindo assim a apresentação de como seria o jogo.

A demo foi exposta no *Play Test VII* que ocorreu no estúdio Cafundó em parceria com a IGDA Florianópolis, o *feedback* recolhido foi muito positivo, foi possível encontrar algumas melhorias a serem feitas em relação ao *game design*, mas como um todo o projeto alcançou seu objetivo.

Foi possível ver também grande parte das dificuldades enfrentadas por estúdios menores, a dificuldade de conseguir mão de obra qualificada e financiamento é muito grande.

Este trabalho nunca teria sido concluído sem o apoio do G2E e da perseverança da professora Mônica, todos os recursos visuais e sonoros , o design do jogo e a programação foram feitos por membros do grupo, todos unidos pelo desejo de produzir um grande jogo. Apesar do jogo não estar completo a estrutura e as ferramentas montadas serão utilizadas na conclusão dele e na construção de outros projetos.

4.1 TRABALHOS FUTUROS

Neste documento foram apresentadas algumas das ferramentas construídas e a estrutura principal para a construção do jogo, porém, parte do conteúdo não foi apresentado, o funcionamento de todas as inteligências artificiais, o sistema de diálogo e carregamento do mesmo, a estrutura de navegação pelos grafos e transições de ambientes, o modo *stealth* e o funcionamento de menus internos e externos do jogo já estão implementados e funcionando, mas foram omitidos para que o outro programador que os implementou possa apresentá-los em seu trabalho de conclusão de curso.

Além disso é possível fazer melhorias no sistema de eventos, que se limita a detectar por colisões e utiliza uma classe abstrata para atribuir as reações, de modo que obriga a criação de um *prefab* para cada reação. O modo de tiroteio não foi concluído e funcionaria em conjunto

o modo *stealth*. Outras mecânicas também poderiam ser implementadas e correções nas atuais podem ser feitas utilizando o *feedback* obtido.

O G2E está sempre com suas portas aberta para pessoas determinadas, qualquer um que tenha se interessado pelo projeto e pelo universo transmídia criado é bem-vindo a conhecê-lo de mais perto.

5 REFERÊNCIAS

WASHBURN, Michael.IEEE INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING COMPANION, 38., 2016, Rochester. What Went Right and What Went Wrong: An Analysis of 155 Postmortems from Game Development. [s. L.]: Ieee, 2016. 9 p.

LEWIS, Chris .THE INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 33., 2011, New York. The whats and the whys of games and software engineering. [s. L.]: Acm, 2011. 4 p.

BLOW, Jonathan. Game Development: Harder Than You Think. Magazine Queue, New York, v. 1, p.29-37, 10 fev. 2004.

Component-Oriented Programming By Andy Ju An Wang, Kai Qian

DEALS, Tnw. This engine is dominating the gaming industry right now. 2016. Disponível em: ¡https://thenextweb.com/gaming/2016/03/24/engine-dominating-gaming-industry-right-now¿. Acesso em: 24 mar. 2016.

OSMAN, Kamisah. Chemistry Learning Through Designing Digital Games. 4. ed. Malaysia: Igi Global, 2017. 4 p. (Encyclopedia of Information Science and Technology).

MARSTON, Hannah R.; HALL, Amanda K.. Gamification: Applications for Health Promotion and Health Information Technology Engagement. 1: Igi Global, 2015. 27 p.

MCSHAFFRY, Mike; GRAHAM, David. Game Coding Complete. 4. ed. Boston: Course Technolog, 2012. 881 p.

https://thenextweb.com/gaming/2016/03/24/engine-dominating-gaming-industry-right-now

APÊNDICE A – Artigo

The Rotfather: Desenvolvimento de um adventure game comercial

Gabriel Capeletti[1]

[1]Departamento de Informática e Estatátisca – Universidade Federal de Santa Catarina (UFSC)
Florianópolis – SC – Brazil

gabriel.capeletti@outlook.com

Abstract. *The game industry has grown fast in the last decades, while it promotes a billionaire creative and entertainment market it also promotes technological advance. Even though Brazil is a great game consumer he does not has much games developed. At the end of this work a adventure game will be developed with capabilities of competing in the actual game market. It will also encourage brasilians developer aspirants and guide then in the game development process giving a small step in the long journey of bringing games closer to Brazil's universities.*

Resumo. *A indústria de jogos digitais têm crescido de uma maneira muito rápida nas últimas décadas, ao mesmo tempo que ela promove um mercado bilionário criativo e de entretenimento também incentiva o avanço tecnológico. Apesar de o Brasil ser um grande consumidor de jogos digitais ele ainda está muito atrás no quesito de desenvolvimento. Ao final deste trabalho terá sido construído um jogo do estilo adventure, com capacidade de competir no mercado de jogos digitais atual e também irá incentivar aspirantes brasileiros e guiá-los no desenvolvimento de um jogo que por consequência contribuirá para que o meio acadêmico se aproxime da indústria dos jogos digitais.*

1. Introdução

Dada a grande demanda de jogos digitais no Brasil e a pouca produtividade do mesmo, é possível encontrar um grande mercado pouco explorado, para que esse mercado seja explorado é preciso aproximar o desenvolvimento de jogos aos brasileiros.

Uma parte considerável dos lucros da indústria de *softwares* é destinado ao jogos eletrônicos, consequentemente parte dos profissionais formados nas áreas de TI acabam por trabalhar com jogos (WASHBURN 1 autor et al.,2016). O preparo deste profissionais acaba sendo incompleto no atual estado dos ensinos nas univerdades. Este trabalho é apenas um pequeno passo para a aproximação do desenvolvimento de jogos digitais e o meio acadêmico brasileiro.

2. Escolha de ferramentas

Mcsaffry e Graham (2012) nomeiam como camada de aplicação todo tipo de programa que é responsável por estabelecer a comunicação entre o jogo e o computador que está sendo usado, podendo variar de um *pc windows* à um *playstation 4* ou um celular. Esta camada possui grande responsabilidade para o funcionamento do jogo e engloba várias funções. Para agilizar o desenvolvimento dos jogos inúmeras ferramentas foram criadas, para iniciar o desenvolvimento é preciso primeiro definí-las tendo em mente a plataforma ao qual o jogo será desenvolvido.

2.1. Bibliotecas gráficas

As bibliotecas gráficas são responsáveis por apresentar toda a visão do jogo ao jogador. Este processo pode ou não envolver as placas gráficas e muitas vezes é utilizado na apresentação de jogos 3D. Elas permitem que a aplicação se comunique com o SO ou *hardware*, dando acesso e liberdade para a aplicação apresentar imagens ao usuário.

Existem várias bibliotecas gráficas no mercado, porém duas delas se destacam por popularidade e performance. A Direct3D (integrada na DirectX) é uma API exclusiva da Microsift que dominou por muitos anos o mercado formando um monopólio dos jogos com os SOs da Microsoft.

A OpenGL apesar de existir desde 1992, está apenas agora recuperando espaço no mercado sobre a DirectX, mas apresenta um futuro promissor já que as novas tecnologias que surge estão utilizando-a.

2.2. Motores de física

Os motores de física ou *physics engines* permitem simulações físicas sobre um grupo de objetos. Os jogos que utilizam alguma forma de cálculo físico acabam por utilizar um desses motores.

Existem motores de física para várias situações diferentes, cada qual com seus pontos fortes e fracos. Diferente das duas APIs gráficas apresentadas, eles são de fácil uso e para cada plataforma existe uma variedade diferente.

2.3. *Game engines*

Foram apresentados dois tipos de bibliotecas que auxiliam o desenvolvimento de jogos, mas existem outros tipos, como as *engines* de áudio, gerência de memória, comunicação em rede, uso de periféricos entre outros.

Ao se desenvolver um jogo, serão escolhidas algumas dessas ferramentas que serão ainda integradas. Visando poupar os desenvolvedores de passar por esse processo, surgiram as *Game engines*. Elas agrupam várias funcionalidades de outras bibliotecas em uma mesma *engine*.

É possível encontrar *Game engines* para vários tipos de plataformas e em várias linguagens. De acordo com uma pesquisa realizada pelo site thenextweb existem duas *engines* que dominam o mercado:

- Unreal Engine
  A Unreal é uma engine desenvolvida pela Epic Games, teve sua estréia com o jogo Unreal. O foco dela é em jogos 3D com alto processamento gráfico, por estas razões sempre foi a favorita pelas grande empresas de jogos. Atualmente ela utiliza C++ como linguagem de programação, facilitando seu uso, já que a grande maioria dos desenvolvedores utilizam C++ por permitir uma melhor otimização do código.
- Unity Engine
  A Unity foi lançada em 2005 e era exclusiva para OS X. Por ser uma engine mais nova e menos poderosa em relação a seus competidores(Unreal, CryEngine), possuía uma curva de aprendizado menor, facilitando a entrada de novos desenvolvedores no mercado. Como já visto anteriormente, a maioria dos jogos indies

acabam por serem feitos em 2D, já que 3D consome muito tempo e trabalho, se tornando inviável para pequenos grupos sem financiamento. Tendo este como seu maior cliente a Unity foi se moldando aos poucos, sua versão 4 foi a primeira a vir com vários suportes para jogos 2D e atualmente ela é uma engine recomendada tanto para jogos 2D quanto 3D.

Como este trabalho será um jogo 2D a Unity se torna a melhor opção.

3. Estrutura da Unity

Primariamente a Unity utiliza C# como sua principal linguagem de programação, sendo ela uma linguagem que aceita o paradigma orientado a objetos permite que os jogos dentro dela sejam programados utilizando este paradigma, porém, ela possui uma API que aprimora a estrutura básica disponível pelo C#, permitindo o uso do paradigma orientado a componentes.

3.1. Programação Orientada a Componentes

Programação orientada a componentes permite que programas sejam construídos a partir de componentes de *software* pré-definidos, que são reutilizáveis e independem um do outro. Estes componentes devem seguir alguns padrões predefinidos incluindo interface, conexões, versões e entregas.(WANG e QIAN, 2005).

Dentro da Unity estes componentes devem herdar de um *MonoBehaviour* que é responsável por implementar todas as comunicações entre os componentes e inserir o componente dentro do ciclo de vida da Unity. A estrutura da Unity se resume de forma superficial à uma simulação de uma lista de *GameObjects*, de forma que a cada *frame* ou *loop* cada um destes objetos realizarão uma sequência de operações.

A figura 1 mostra a sequência que os métodos de um *MonoBehaviour* são chamados, para cada um desses métodos a Unity irá percorrer a sua lista de *GameObjects* e realizar a chamada do respectivo método. Apesar da grande quantidade de métodos normalmente são utilizados principalmente dois deles, que já são implementados ao se criar uma classe de componentes dentro da Unity, o método *Start* e *Update*. O *Start* como pode ser visto na figura 1 é chamado apenas uma vez, quando é criado, enquanto o *Update* é chamado uma vez a cada ciclo.

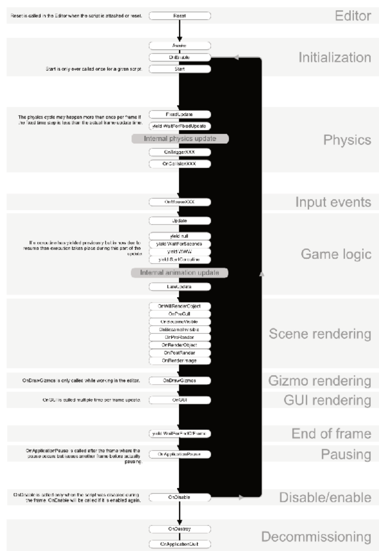

Figura 1. Ciclo de vida da Unity

4. Arquitetura

Para a programação dos objetos na Unity é necessário a criação de componentes que serão adicionados à eles. A comunicação entre componentes ocorre através da referencia do objeto e o método de busca **GetComponent<T>()**, que pertence aos objetos do tipo *MonoBehaviour*.

Apesar de útil este método é custoso e portanto costuma-se utilizá-lo na inicialização do objeto. Como mostrado anteriormente no ciclo de vida da Unity existem dois métodos para a inicialização dos objetos, **Awake()** e **Start()**, ambos são chamados apenas uma vez logo após a inicialização, sendo que o método **Awake()** é chamado primeiro.

Dada esta ordem de procedência foi escolhido o método **Awake()** para a inicialização e busca de referências, e o método **Start()** para o uso destas referências e configurações dos objetos. Deste modo é possível evitar a ocorrência de erros por referências nulas.

Visando facilitar a legibilidade do código e a modularização do mesmo, foi criada uma arquitetura base dos componentes, que buscam as referências dos componentes mais utilizados.

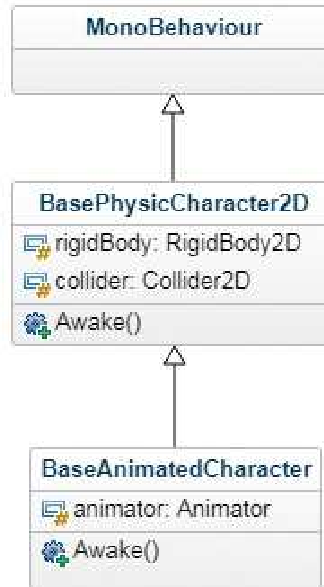

Figura 2. Arquitetura base.

4.1. Máquina de estados

Nos jogos digitais o uso de máquinas de estado é muito comum, cada ação do jogador gera inúmeras transições de estados para cada objeto, por exemplo se uma parede é destruída ele teve uma alteração de estado, seus colisores foram alterados e ela apresentou uma animação, se um objeto do chão é carregado pelo jogador ele não pode ser carregado novamente e deve estar na mão do jogador. O personagem principal é quem geralmente acaba por ter o maior número de estados, já que cada *input* acaba por realizar uma alteração no estado do jogo e cada estado possui uma regra diferente e se comporta de maneira diferente em relação aos *inputs*.

Como a programação na Unity utiliza o paradigma de orientação à componentes, grande parte dos *Design Patterns* da orientação à objetos não podem ser aplicados diretamente, eles devem ser adaptados à estrutura da Unity e ao paradigma. No caso da máquina de estados foi utilizada uma versão já adaptada para Unity e em cima dela foram feitas alterações para otimizar e facilitar o uso da ferramenta.

Esta estrutura faz o uso de duas **Enums**, uma para representar os estados e outra para representar as transições entre os estados, a implementação dos estados ocorre em uma classe referente àquele estado e que herda de uma classe base genérica para todos os estados, a gerência dos estados ocorre por uma outra classe também genérica que realiza o processo de transição e atualização dos estados.

4.1.1. Estados

Cada estado de um objeto é independente do outro, porém, todos eles herdam da classe base **FSMState** que é genérica, de modo que o estado sempre possa usar a referencia de seu controlador base.

Os estados estão fora da estrutura de componentes da Unity mas possuem sempre a referencia ao componente que os gerencia. Um estado possui como atributos além da referencia ao componente, um mapa entre as transições e os estados, ambos em formas de **Enum** e por último uma referencia à **Enum** de estados para identificar qual estado esta classe representa.

Os métodos responsáveis por configurar as transições do objeto permitem a adição e remoção de transições e informam qual estado resulta de uma dada transição. Estes métodos são apenas utilizados para a configuração dos estados que ocorre na inicialização dos objetos. Para o controle do objeto existem três métodos, um chamado quando ocorre uma transição para este estado, outro quando ocorre uma transição deste estado e por último um método de *update*, que é chamado a cada frame e permite que ações contínuas sejam realizadas no objeto. Qualquer um deste métodos pode realizar uma chamada à máquina de estados para realizar uma transição, porém, existe um outro método apenas para conferir se alguma transição deve ocorrer, este método é chamado também a cada *update* e é abstrato na classe base, obrigando a implementação em todos os estados.

4.1.2. Sistema gerenciador de estados

Como falado anteriormente os estados são independentes, de modo que um estado não possui a referencia para outro. Eles utilizam apenas **Enums** para realizarem as transições, logo para que a máquina de estados funcione é necessário um sistema responsável por conectar e gerenciar estes estados.

Para guardar as referências e acessá-las rapidamente é utilizado um mapa entre a **Enum** de estados e o estado, além do mapa, existe uma referencia ao estado atual que é atualizado a cada frame.

Esse sistema permite então a inserção e remoção de estado e a transição entre os estados. Durante a transição ele irá através do mapa do estado atual, verificar se é possível realizar a transição desejada, caso possível, irá fazer a chamada para o método de saída do estado atual, atualizar o estado atual e fazer a chamada para o método de entrada.

A construção desta máquina e a conexão dela com a estrutura da *Unity* ocorre por meio de um controlador que irá montar os estados e serializar o dados necessários para a máquina construída, permitindo que o *game designer* possa configurá-la pela interface da

Unity.

5. Construção de ferramentas próprias

Mesmo com uma *game engine*, falta a implementação do *game core*, onde toda a estrutura do jogo ainda deve ser montada. Para que seja possível a criação de novos conteúdos para o jogo e que a manutenção no mesmo seja fácil, deve ser planejado um conjunto de ferramentas internas do jogo, que permitirão que os responsáveis por montar o *level* possam fazê-lo sem necessitar tocar em uma linha do código. Estas ferramentas devem dar o máximo de liberdade possível para o *game designer* e devem estar bem estruturadas para o caso de mudanças serem necessárias.

5.1. Sistema de eventos

O objetivo do sistema de eventos é permitir que o *level designer* possa criar ações e reações de acordo com situações que ocorram dentro do jogo sem necessitar programar.

Para isso foi utilizada a estrutura básica de animação da *Unity*. O sistema de animação permite que qualquer atributo serializável de um componente possa variar de acordo com o tempo, permitindo a alteração de variáveis de todos os tipos como apresentado na figura 3.

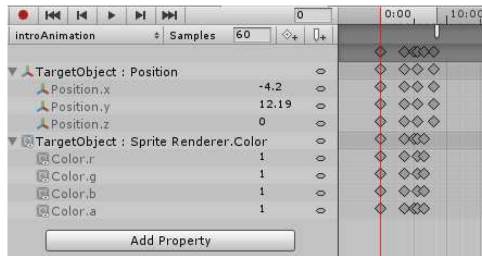

Figura 3. Interface da ferramenta de animação

Esta ferramenta de animação já permite grande liberdade ao level designer para reproduzir ações dentro do jogo, porém, é ainda necessário os gatilhos para essas animações. O sistema de eventos permite então que dada uma situação, uma lista de objetos sofram um sequência de reações.

Para construir esta ferramenta foi então necessário definir as formas de gatilho dos eventos, para este trabalho foram utilizadas apenas colisões, porém, poderia se ter utilizado outras formas de evento, como a vida do jogador ou algum padrão de *Input* do jogador.

Os gatilhos são representados por uma **Enum InteractiveActionMap**, que lista as possíveis formas de detecção de ações. As ações são definidas em uma classe *singleton* nomeada de **Action**, nesta classe existe uma **struct** com as funções que podem detectar colisão.

A definição destas funções ocorrem na inicialização da **Action**, que por ser um *singleton* ocorrerá apenas uma vez. No construtor é percorrido todos os elementos do **InteractiveActionMap**, para cada elemento é criada uma nova instância da **ActionFunctions** definido suas condições de acordo com o elemento atual da iteração, após definidas as funções elas são então salvas em um dicionário entre as **InteractiveActionMap** e a instância da **ActionFunctions**.

Com as ações definidas, foi criado um componente **InteractiveObject** para definir as ações, conectar as funções de detecção com os componentes físicos do objeto e chamar uma lista de objetos que sofrerão reações da interação.

5.2. Ferramenta para o controle de múltiplas *viewports*

Para permitir que fossem criados efeitos de mais de uma *viewport*, foi necessário uma ferramenta de manipulação de cameras *viewports*, este sistema ficou responsável por gerenciar todas as câmeras em cena.

5.2.1. *Main Camera*

Esta câmera é a principal, sua *viewport* é a própria *window*, diferente das câmeras secundárias ela deve recalcular sua altura e largura de acordo com a resolução da *window*, de modo que ela sempre irá mostrar a mesmas imagens independente da resolução do jogador.

A resolução padrão para o jogo é de 1920x1080 pixels, uma proporção de 16:9, de modo que resoluções com proporções diferentes terão uma tarja preta que irá manter a imagem de apresentação igual à resolução padrão.

5.2.2. *Secondary Camera*

A *secondary camera* é composta por dois objetos diferentes, o primeiro é a própria câmera que possui bordas, funções de *character follow* e cálculos de altura e largura em coordenadas de mundo, o segundo é a *viewport*, que irá aplicar sobre uma câmera definida na interface da ferramenta, transformações de acordo com a resolução da *window*, uma mesma *viewport* pode alternar entre várias câmeras.

O processo de configurar a câmera de acordo com a configuração da *viewport* exige uma sequência de cálculos. Os valores de comprimento, largura e posição são calculados sem restrições e a cada etapa é limitados e acrescentado variáveis no cálculo.

5.3. Ferramenta para a criação e manipulação de grafos

Como o foco do jogo está nos diálogos e no desenvolver da história, o *design* do jogo limita as movimentações dos personagens em locais específicos. Desta maneira é possível saber onde o jogador irá e quais cenas ele verá, este é um padrão adotado pela maioria dos *adventure games*.

Para permitir a criação, manipulação e intersecção de caminhos, foi criada a ferramenta de *path plot*, que permite que o *level designer* crie nodos e conecte-o com outros nodos.

Como a principal plataforma que se pretende lançar o jogo é o PC, os controles básicos de movimentos devem considerar um teclado e mouse. A movimentação do personagem pelo teclado é limitada então em seis direções, resultado entre a combinação de direita e esquerda com cima, baixo e nada. Cada nodo possui um mapa onde a chave para ele é uma das seis direções e o valor corresponde ao nodo seguinte àquela direção.

O uso da ferramenta é simples, foi criado um atalho (ctrl+f) que cria um nodo, este nodo ainda não está posicionado e irá seguir a posição do mouse, por padrão ele irá considerar que está conectado ao nodo que estava selecionado quando o atalho foi pressionado, esta conexão é representada pela aresta branca. Quando pressionado, a ferramenta irá detectar a posição do nodo e os nodos vizinhos para definir uma chave, caso a posição já esteja ocupada ele irá destruir o novo nodo, caso o novo nodo esteja sobre um nodo já existente, em vez de se criar um novo nodo é criada uma nova conexão entre o dois nodos antigos. As conexões criadas entre os nodos são representadas pelas arestas verdes, na figura 4 é possível ver a interface gerada em um nodo após ter todas suas direções preenchidas.

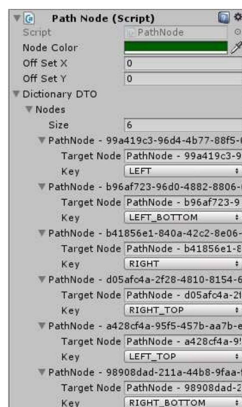

Figura 4. Interface criada com a ferramenta *path plot*.

Todos os campos criados em objetos dentro da Unity têm seus valores serializados e salvos. Mas isso só ocorre com algumas variáveis básicas, para pode serializar o mapa de direções de cada nodo, foi criado um DTO, responsável por guardar o nome de cada nodo e sua respectiva chave, durante o processo de desserialização cada nodo presente no dicionário é buscado, utilizando o nome dele em formato de string, ao ser encontrado é inserido no dicionário com a chave relacionada ao nome.

6. Conclusão

A proposta deste trabalho é apresentar o processo e as técnicas aplicadas e desenvolvidas na construção de um jogo que fosse apto a ser comercializado.

Ao final deste trabalho foi construída uma *demo* no formato de um *vertical slice* onde todas as mecânicas e ferramentas inicialmente planejadas estivessem implementadas, permitindo assim a apresentação de como seria o jogo.

A demo foi exposta no *Play Test VII* que ocorreu no estúdio Cafundó em parceria com a IGDA Florianópolis, o *feedback* recolhido foi muito positivo, foi possível encontrar algumas melhorias a serem feitas em relação ao *game design*, mas como um todo o projeto alcançou seu objetivo.

Foi possível ver também grande parte das dificuldades enfrentadas por estúdios menores, a dificuldade de conseguir mão de obra qualificada e financiamento é muito grande.

Este trabalho nunca teria sido concluído sem o apoio do G2E e da perseverança da professora Mônica, todos os recursos visuais e sonoros , o design do jogo e a programação

foram feitos por membros do grupo, todos unidos pelo desejo de produzir um grande jogo. Apesar do jogo não estar completo a estrutura e as ferramentas montadas serão utilizadas na conclusão dele e na construção de outros projetos.

Referências

WASHBURN, Michael. IEEE INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING COMPANION, 38., 2016, Rochester. What Went Right and What Went Wrong: An Analysis of 155 Postmortems from Game Development. [s. L.]: Ieee, 2016. 9 p.

BLOW, Jonathan. Game Development: Harder Than You Think. Magazine Queue, New York, v. 1, p.29-37, 10 fev. 2004.

MCSHAFFRY, Mike; GRAHAM, David. Game Coding Complete. 4. ed. Boston: Course Technolog, 2012. 881 p.