Antonio Nascimento Lutfi

# Um Framework para Game Shows Interativos de TV com Realidade Aumentada e Segunda Tela

## DISSERTAÇÃO DE MESTRADO

DEPARTAMENTO DE INFORMÁTICA
Programa de Pós–graduação em Informática

Rio de Janeiro
Abril de 2015

Pontifícia Universidade Católica
do Rio de Janeiro

**Antonio Nascimento Lutfi**

# Um Framework para Game Shows Interativos de TV com Realidade Aumentada e Segunda Tela

**Dissertação de Mestrado**

Dissertação apresentada ao Programa de Pós–graduação em Informática do Departamento de Informática do Centro Técnico Científico da PUC–Rio como requisito parcial para obtenção do grau de Mestre em Informática.

Orientador: Prof. Bruno Feijó

Rio de Janeiro
Abril de 2015

Pontifícia Universidade Católica
do Rio de Janeiro

**Antonio Nascimento Lutfi**

# Um Framework para Game Shows Interativos de TV com Realidade Aumentada e Segunda Tela

Dissertação apresentada ao Programa de Pós–graduação em Informática do Departamento de Informática do Centro Técnico Científico da PUC–Rio como requisito parcial para obtenção do grau de Mestre em Informática. Aprovada pela Comissão Examinadora abaixo assinada.

**Prof. Bruno Feijó**
Orientador
Departamento de Informática — PUC–Rio

**Mônica Maria Ferreira da Costa**
PUC-Rio

**Prof. Luiz Eduardo Azambuja Sauerbronn**
UFRJ

**Prof. José Eugenio Leal**
Coordenador Setorial do Centro Técnico Científico — PUC–Rio

Rio de Janeiro, 17 de Abril de 2015

**Antonio Nascimento Lutfi**

Graduou-se em Sistemas de Informação pela Pontifícia Universidade Católica do Rio de Janeiro em 2010. Trabalhou com Banco de Dados para ferramentas de segmentação e seleção de mercado; Inteligência Artificial para gestão de orçamento público; e, mais recentemente, com Visão Computacional e jogos no ICAD/VisionLab. Possui experiência em múltiplas linguagens de programação e desenvolvimento em ambiente Windows e Linux. Também adquiriu experiência gerindo provas de conceito e relações com clientes.



Ficha Catalográfica

Lutfi, Antonio Nascimento

Um Framework para Game Shows Interativos de TV com Realidade Aumentada e Segunda Tela / Antonio Nascimento Lutfi; orientador: Bruno Feijó. — 2015.

94 f. : il. (color); 30 cm

1. Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2015.

Inclui bibliografia.

1. Informática – Teses. 2. Jogos de auditório em TV. 3. Segunda Tela. 4. Realidade Aumentada. 5. Videogame. I. Feijó, Bruno. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

## Agradecimentos



Ao meu orientador, Bruno Feijó, pela oportunidade e o incentivo à pesquisa realizada.

À Rede Globo e ao ICAD/VisionLab, pelo projeto que deu início a essa pesquisa.

A meus pais

À Bel

Este trabalho foi realizado com o apoio do CNPq e da PUC-Rio

## Resumo

Lutfi, Antonio Nascimento; Feijó, Bruno. **Um Framework para Game Shows Interativos de TV com Realidade Aumentada e Segunda Tela**. Rio de Janeiro, 2015. 94p. Dissertação de Mestrado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Esta dissertação apresenta um framework para a criação de jogos de auditório em TV (game shows interativos) usando Realidade Aumentada em estúdios, que permitam a participação dos telespectadores através de tablets e smartphones como segunda tela. Este trabalho de pesquisa também investiga novos paradigmas de convergência entre TV e videogames.



## Palavras–chave

Jogos de auditório em TV; Segunda Tela; Realidade Aumentada; Videogame.

## Abstract

Lutfi, Antonio Nascimento; Feijó, Bruno (Advisor). **A Framework for TV Game Shows with Second Screen**. Rio de Janeiro, 2015. 94p. MSc. Dissertation — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

This dissertation presents a framework for the development of interactive TV game shows using augmented reality in TV studios, which allows the participation of viewers using tablets and smartphones as the second screen. This research also investigates new convergence paradigms between TV and video games.



## Keywords

TV Game Shows; Second Screen; Augmented Reality; Video Game.

# Sumário


1 Introdução **10**

2 Estado da Arte **13**
2.1 Realidade Aumentada em Game Shows 13
2.2 Realidade Aumentada em Jogos 14
2.3 Dispositivos Móveis como Segunda Tela 16

3 MarkerFinder **18**
3.1 Situação inicial do processo 18
3.2 Motivação e Objetivo do Projeto 19
3.3 Especificação Funcional 21
3.4 Algoritmo 23
3.5 Modelagem de Classes 25
3.6 Detalhes de Implementação e Testes 29
3.7 Adaptabilidade 34
3.8 Resultados 34

4 Bons Jogos e bons Game Shows **35**
4.1 Bons Jogos 35
4.2 Bons Game Shows 53

5 Um Framework para Game Shows Interativos **56**
5.1 Potenciais desafios à Convergência 56
5.2 Framework em Linhas Gerais 59
5.3 Modelagem de Classes 61

6 Implementação do Framework Proposto **66**
6.1 Testes 70

7 Conclusões Finais e Trabalhos Futuros **75**
7.1 Trabalhos Futuros 76

8 Referências Bibliográficas **80**

A Anexos **86**
A.1 Descrição detalhada das classes do Framework 86

# Lista de figuras


1.1 *(Bamzooki). Times de convidados competem com criaturas virtuais em uma arena de Realidade Aumentada.* 10
1.2 *(HarryPotter, 2001). Cena do filme Harry Potter e a Pedra Filosofal. Os personagens devem desempenhar o papel de peças em um jogo de xadrez. Num Game Show Interativo, cada peça poderia ser controlada por um telespectador.* 11

2.1 *(Bamzooki). Ferramenta de contrução de Zooks à esquerda e competição no palco do programa à direita.* 13
2.2 *(Ingress, 2013). Jogadores disputam o controle de um portal que coincide com um monumento.* 15
2.3 *(Ingress, 2013). Mapa dos Estados Unidos exibindo a distribuição geográfica de membros de cada facção.* 15
2.4 *PulzAR(Wharton). A posição dos marcadores fiduciários altera o rumo do laser para desviá-lo de obstáculos, conduzindo-o ao destino certo.* 15
2.5 *Pokedex 3D Pro(Pokedex).* 16

3.1 *Exemplo de estúdio da (Globo) equipado com marcadores para o uso do LightCraft.* 19
3.2 *Marcador fiducial do (LightCraft).* 20
3.3 *Faro (Focus).* 20
3.4 *Caso de Uso - Inicialização.* 22
3.5 *Caso de Uso - Calibragem de parâmetros e detecção de círculos.* 22
3.6 *XYWZ é a menor bounding box do grid que contém o triângulo ABC.* 24
3.7 *Diagrama de Classes do MarkerFinder.* 25
3.8 *Limites do Pixel.* 27
3.9 *Detecção de círculos antes do ajuste dos parâmetros de Hough.* 30
3.10 *Detecção de círculos depois do ajuste manual dos parâmetros de Hough.* 30

4.1 *Escalada linear de desafios em cima e oscilante embaixo(Schell, 2008).* 37
4.2 *Gêneros de acordo com a expectativa de seus públicos.* 38
4.3 *World of Warcraft(WoW, 2004) à esquerda, Diablo 1 à direita, lançado em 1996 e Diablo 3 embaixo, lançado em 2012(Diablo3, 2012). Nota-se um aprimoramento de uma mesma mecânica.* 40
4.4 *(Skyrim, 2011).* 41
4.5 *(Fallout, 2010) New Vegas, passado em uma Las Vegas pós guerra nuclear.* 41
4.6 *Setup de computador pessoal para simulador de vôo(FlightSim, 2006). Embora os desenvolvedores não esperem esse nível de todos os jogadores, é exigido do jogo o máximo de realismo.* 42
4.7 *Microsoft Flight Simulator X. Avião se aproxima da cidade do Rio de Janeiro(FlightSim, 2006).* 43
4.8 *(SimCity). Cidade administrada por um jogador.* 43

4.9 *The (Sims). Jogador é responsável por gerir uma casa e a vida de seus moradores.* 43
4.10 *(StarCraft, 2010) II. Batalha em tempo real entre artilharia e infantaria.* 45
4.11 *Civilization V(Civilization, 2010). Jogo em turnos com o mapa dividido em tabuleiro no formato de colméia.* 45
4.12 *League of Legends(LoL, 2009). Cada jogador controla um exército.* 45
4.13 *Cubo de Rubik à esquerda. À direita, puzzle de madeira de seis peças* 46
4.14 *The Legend of Zelda: Skyward Sword(Zelda, 2011). Mapa de um labirinto onde cada quadrado representa uma sala. O jogador deve reposicioná-las para ganhar acesso a elas.* 46
4.15 *(Braid, 2009). Os elementos marcados de vermelho são imunes à manipulação do tempo. Todos os outros retrocedem quando o personagem também retroceder. O jogador tem que usar este tipo de mecânica para sincronizar objetos e prosseguir.* 48
4.16 *(FEZ, 2012). O cenário é tridimensional mas o movimento é realizado em uma das projeções. Isto pode revelar segredos e aproximar objetos distantes.* 49
4.17 *The Last of Us(LastofUs, 2013). Tensão gerada pela dificuldade de sobrevivência em um ambiente hostil e dos laços entre personagens. Controles inovadores em um gênero já consagrado.* 50
4.18 *Evolução dos três jogos da série (DeusEx, 2000). As habilidades do personagem são adquiridas ao longo do jogo de acordo com as abordagens preferidas do jogador, seja ele furtivo, sociável ou agressivo.* 50
4.19 *Tela de jogo do Guitar Hero. As notas correspondem aos botões de mesma cor, que devem ser apertados na ordem em que cruzam a linha inicial(GuitarHero, 2005).* 51
4.20 *(Ikaruga, 2003). O desafio é tão difícil que a nave que o jogador controla (parte inferior da tela) se camufla entre os projéteis dos quais ela deve desviar.* 51
4.21 *Super Meat Boy. O jogador controla um pedaço vermelho de carne em busca de sua namorada raptada por níveis complexos com pouquíssimo espaço para erros(SuperMeatBoy).* 52
4.22 *(Flower). O jogo conta com controles fluidos e visual atraente.* 53
4.23 *(Journey). O jogador vai em direção à montanha ao fundo.* 53

5.1 *Arquitetura de comunicação entre os módulos.* 60
5.2 *Diagrama de Classes do Framework.* 62

6.1 *Jogo de damas implementado de acordo com o framework.* 71

7.1 *Oculus Rift(Oculus) à esquerda e Omni treadmill(Omni) à direita.* 78

# 1
# Introdução

A indústria do videogame possui a maior arrecadação bruta dentre as de entretenimento(CBS, 2011)(Gartner, 2013). As vendas de dispositivos móveis já superam as de computadores e notebooks(Gartner, 2013). Neste cenário tecnológico relativamente novo, a hegemonia da Televisão como acessório indispensável ao entretenimento doméstico já não é mais tão forte.

Inspirado por estas três indústrias, este trabalho é um estudo inicial da potencial convergência entre TV e videogame, propondo, como um novo tipo de mídia, Game Shows que utilizem Realidade Aumentada no estúdio e permitam a interação dos telespectadores via dispositivos móveis. A este novo formato foi dado o nome de Game Show Interativo.

O programa de TV com o formato mais próximo ao proposto é o (Bamzooki)(figura 1.1), da inglesa (BBC). Entretanto, sua distância para um game show interativo é grande, principalmente porque falta estrutura de videogame e a qualidade da imagem está fora dos padrões de TV em alta definição.

Figura 1.1: *(Bamzooki). Times de convidados competem com criaturas virtuais em uma arena de Realidade Aumentada.*

A viabilização desse novo tipo de mídia pode possibilitar o compartilhamento de espaços virtuais por participantes de auditório e telespectadores. Programas de TV ao vivo têm suas possibilidades de interação tradicionalmente limitadas a ligações de telefone ou mensagem de texto, geralmente como uma forma de voto. A inserção de elementos virtuais

interativos via internet adiciona muitas outras maneiras de participação do público(figura 1.2). Com a provável popularização de óculos de realidade virtual, como o Oculus Rift(Oculus), esse compartilhamento de espaços virtuais pode ficar ainda mais elaborado, sendo possível até que alguns convidados do programa não estejam fisicamente no palco, mas somente virtualmente representados. A possibilidade deste exemplo é levantada e melhor elaborada na seção 7.1.2.

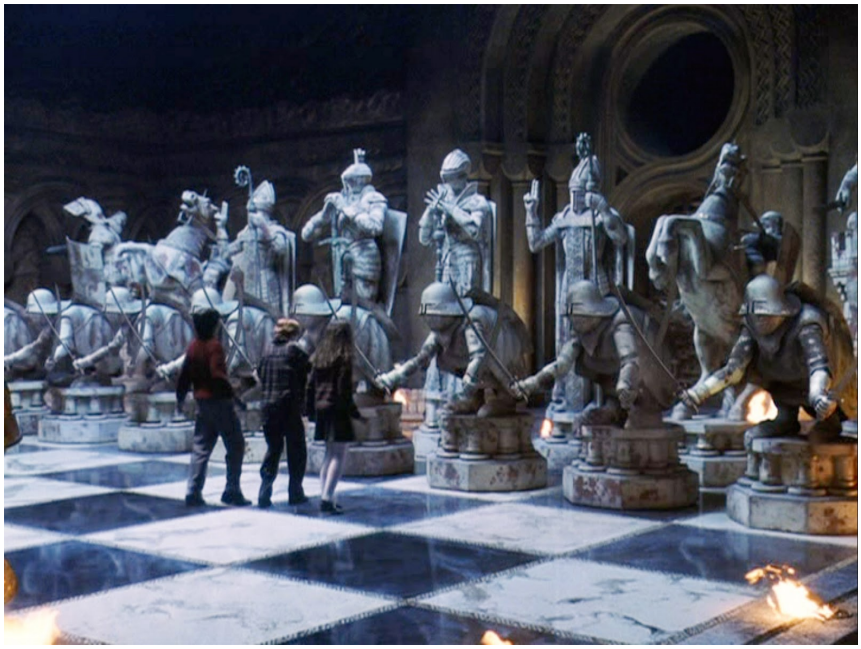

Figura 1.2: *(HarryPotter, 2001). Cena do filme Harry Potter e a Pedra Filosofal. Os personagens devem desempenhar o papel de peças em um jogo de xadrez. Num Game Show Interativo, cada peça poderia ser controlada por um telespectador.*



Como não existem relatos de trabalhos similares na literatura, um formato convergente entre TV e videogame demanda um estudo destas áreas em separado.

Além do objetivo de buscar novos paradigmas para TV e videogames, esta dissertação pretende propor um framework para o desenvolvimento deste tipo de Game Show, que lide com a interação dos telespectadores e com a exibição de elementos virtuais no estúdio. Com esta ferramenta em mãos, o desenvolvimento de jogos deste tipo e, consequentemente, o avanço da pesquisa são consideravelmente facilitados.

A dissertação está organizada como se segue. O capítulo 2 descreve o atual estado da arte da Realidade Aumentada em jogos e em TV e discute o uso de dispositivos móveis como segunda tela.

O capítulo 3 descreve, em detalhes, o projeto MarkerFinder, encomendado pela Rede Globo(Globo) para diminuir o tempo necessário de pré-produção de um estúdio onde serão usados elementos virtuais. Este projeto representa a pesquisa em seu estágio embrionário. Foi ele que serviu de inspiração para o projeto proposto. Suas funcionalidades serão incorporadas à parte relativa à RA em estúdio do framework.

O capítulo 4 relata um estudo das características responsáveis pelo sucesso de jogos de videogame(4.1), e de game shows(4.2).

A formulação do framework proposto é descrita no capítulo 5. Em 5.1, são levantados desafios que poderão ser encontrados devido a convergência entre televisão e videogame. Evidenciados estes obstáculos, o framework é descrito em linhas gerais na seção 5.2 e sua modelagem de classes em 5.3.

Os detalhes do estado atual de implementação do framework estão no capítulo 6.

O capítulo 7 encerra esta dissertação com considerações finais e propostas de trabalhos futuros.

# 2
# Estado da Arte

Como não existem aplicações que se enquadrem exatamente na área da pesquisa descrita nesta dissertação, este capítulo apresenta o Estado da Arte nas áreas que convergem para ela.

Não são apresentados trabalhos similares ao MarkerFinder(capítulo 3), pois não foi encontrada nenhuma publicação que os relatasse.

## 2.1
## Realidade Aumentada em Game Shows

Game Shows, tradicionalmente, são realizados em um estúdio onde participantes competem em um jogo, geralmente com um apresentador.

Existem poucos exemplos de Realidade Aumentada(RA) sendo usada em Game Shows. No momento, a RA em TV está mais restrita a showcases. Um destes poucos exemplos é o inglês (Bamzooki), programa da (BBC) voltado para o público infantil. Nele, crianças montam animais fictícios -Zooks- em suas casas e competem umas com as outras via internet. As melhores criações são selecionadas para competir ao vivo em auditório, onde são exibidas para o telespectador em uma arena de RA(figura 2.1). O programa possui um viés educativo, pois a montagem das criaturas é um processo de tentativa e erro que ensina noções de mecânica.

Figura 2.1: *(Bamzooki). Ferramenta de contrução de Zooks à esquerda e competição no palco do programa à direita.*

É importante ressaltar que o trabalho proposto neste documento demanda uma qualidade bem maior do que a apresentada neste exemplo. Além de serem modelados para um ambiente de alta definição -de FullHD a 4K-, os elementos virtuais devem ser posicionados no cenário real com muita

precisão. O Bamzooki exibe os modelos virtuais em uma espécie de mesa. Um Game Show Interativo no formato proposto vai além disso e tem como objetivo fazer os objetos reais e virtuais interagirem de forma mais dinâmica e livre. Para alcançar uma precisão deste tipo, é necessário um software de localização espacial da câmera em tempo real, como o (LightCraft). Este foi usado no projeto MarkerFinder e é explicado em mais detalhes no capítulo 3. Não existem relatos na literatura do uso de um software deste tipo para Game Shows.

## 2.2 Realidade Aumentada em Jogos

Embora ainda distante das abordagens convencionais, a Realidade Aumentada ganha cada vez mais espaço nos jogos de videogame. A necessidade de uma câmera que possa se mover livremente é uma restrição que faz com que jogos em RA estejam naturalmente mais presentes em dispositivos móveis, como tablets, smartphones e videogames portáteis.

Exitem diversos casos de jogos em RA que têm o papel de peças publicitárias. Estes exemplos são muito simples e não compartilham as motivações de um jogo feito como entretenimento puro. Por esta razão estes casos não serão abordados.

Um exemplo relevante é o (Ingress, 2013), um *Massively Multiplayer Online(MMO) Game* jogado em smartphones. Um jogador escolhe uma dentre duas facções que competem pelo controle de portais. Estes se localizam no mundo real, em marcos geográficos conhecidos. O GPS do dispositivo reconhece a localização do jogador e sobrepõe os elementos virtuais sobre os marcos físicos(figura 2.2). Como o jogo é jogado ao longo do mundo inteiro e o jogador tem a capacidade de escolher a qual facção pertence, a representatividade de cada facção não é globalmente uniforme. Isto enriquece o tema do jogo de um mundo dividido, pois cada área tem sua facção predominante e oferece uma experiência de competição diferente(figura 2.3).


Figura 2.2: *(Ingress, 2013). Jogadores disputam o controle de um portal que coincide com um monumento.*

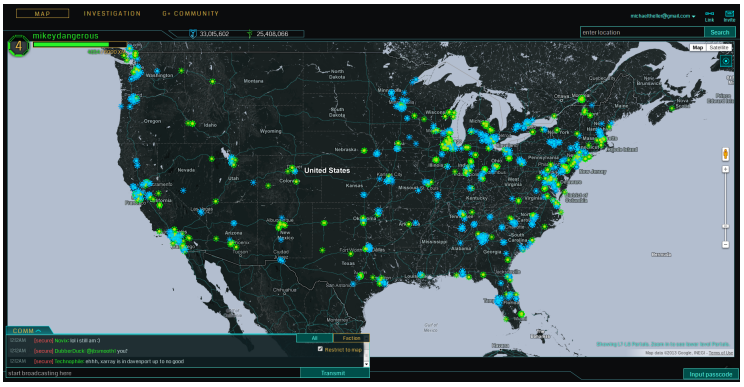


Figura 2.3: *(Ingress, 2013). Mapa dos Estados Unidos exibindo a distribuição geográfica de membros de cada facção.*


Figura 2.4: *PulzAR(Wharton). A posição dos marcadores fiduciários altera o rumo do laser para desviá-lo de obstáculos, conduzindo-o ao destino certo.*

PulzAR(Wharton) é um jogo de puzzle para o (PSVita), que utiliza uma superfície plana real como seu tabuleiro. O jogador imprime marcadores fiduciários e os usa como ferramentas para sobrepor obstáculos virtuais. Esta é uma mecânica interessante pois incorpora movimentos reais e permite, de forma natural, a participação de mais de um jogador(figura 2.4).

Um exemplo de como franquias já estabelecidas no mercado buscam a Realidade Aumentada para inovar é o Pokedex 3D Pro(Pokedex) para o Nintendo 3DS. Ele consiste em uma enciclopédia com todas as espécies da consagrada franquia (Pokemon) apresentados através de modelos 3D superpostos sobre superfiícies reais(figura 2.5).


Figura 2.5: *Pokedex 3D Pro(Pokedex).*

## 2.3 Dispositivos Móveis como Segunda Tela

Cerca de 70% das pessoas que possuem tablets ou smartphones nos Estados Unidos os utilizam enquanto assistem televisão(Nielsen, 2012). A esse uso simultâneo se dá o nome de segunda tela.

Este é só mais um dos muitos desdobramentos da popularização dos dispositivos móveis e apresenta um desafio significativo às empresas de televisão, cujo principal esforço é manter o telespectador o mais concentrado possível na programação.

Com esta preocupação em mente, os produtores de conteúdo e as emissoras de TV vêm tentando integrar o uso do dispositivo móvel à sua programação. Grosso modo, as aplicações em segunda tela sincronizam o conteúdo da TV com o do dispositivo e apresentam algum tipo de interatividade - desde propaganda até a possibilidade de participação, como votações.

Os dois principais métodos utilizadas para a sincronização das duas telas são o *Audio Watermarking* e o *Fingerprinting*.

O primeiro consiste em analisar o sinal de áudio do programa em exibição procurando por intervalos em que se possa inserir um áudio sutil sem que este possa ser ouvido. O aplicativo móvel capta estas marcas via microfone e, comparando com um banco de dados *online*, sincroniza as telas. A grande vantagem desta técnica é a precisão, dado que a marca inserida dura cerca de dois segundos; uma margem de erro de sincronização razoável. Uma de suas desvantagens é a impossibilidade de esconder estas marcas em longos períodos de silêncio ou áudio baixo.

A segunda -Fingerprinting- consiste em comparar trechos do áudio captado pelo dispositivo móvel -assinaturas- aos conteúdos armazenados no servidor do aplicativo, sincronizando os dois. Uma vantagem desta técnica é a possibilidade de analisar um conteúdo sobre o qual não se tem propriedade, desde que se tenha acesso a seu áudio. Isto é util para pesquisas de opinião, por exemplo, em que a agência pesquisadora não necessariamente tem vínculo com o produtor do conteúdo. Os custos de infraestrutura, no entanto, dificultam a escalabilidade. Gravar o áudio de um programa de TV inteiro pode ocupar muito espaço. Além disso, somente pelo áudio não é sempre possível identificar a emissora. Programas ao vivo, por não terem áudio previamente conhecido, não podem fazer uso desta técnica.

Ainda não está definida a técnica para a sincronia da segunda tela no framework proposto. Como um Game Show Interativo será desenvolvido em uma plataforma própria, provavelmente a sinconização será feita via web, sem depender de captação e processamento de sinais de áudio.

Já existem no mercado diversas ferramentas para sincronização de conteúdo. (Intrasonics, 2013) e Shazam for TV(Shazam) são exemplos que usam Audio watermarking. Já o (Beamly) é um exemplo que aplica o Fingerprinting.

A emissora americana ABC fez o uso de segunda tela através de um aplicativo que fazia publicidade de produtos usados pelos personagens da série Scandal(Tode, 2013). Outro exemplo interessante é o aplicativo da National Geographic(Tode, 2013), que, por se tratar de uma emissora de documentários, apresenta informações mais aprofundadas para quem estiver interessado no tema do programa exibido. Este aplicativo também exibe prévias do próximo bloco durante os comerciais, para desencorajar o telespectador a mudar de canal.

# 3
# MarkerFinder

O projeto surgiu com uma demanda da Rede Globo (Globo) de diminuir o tempo de um de seus processos de efeitos especiais em estúdio. Ele consiste na automatização de uma etapa deste processo, a ser descrita na seção 3.2.

Este capítulo relata, mais adiante: a especificação funcional do projeto(seção 3.3); seu algoritmo, tanto em linhas gerais quanto em detalhes(seção 3.4); a modelagem de suas classes(seção 3.5); e detalhes da implementação e documentação (seção 3.6).

É importante citar que o algoritmo desenvolvido para o MarkerFinder faz uso do método de Hough(HOUGH e MICH, 1962) de detecção automática de círculos.



## 3.1
## Situação inicial do processo

A (Globo) vem utilizando o (LightCraft) em algumas de suas gravações de estúdio. Esta tecnologia consiste em um sistema que reconhece a posição e a direção da câmera em um espaço tridimensional, para automatizar a superposição de imagens através de chroma key. Se um estúdio tiver uma janela, por exemplo, e, atrás dela pretende-se superpor uma paisagem; a posição da câmera determina que recorte da paisagem deverá aparecer.

Para a determinação dessa posição, o LightCraft necessita que a câmera seja equipada com uma lente grande-angular apontada para o teto do estúdio, além de sua lente normal. No teto do estúdio, por sua vez, há uma série de marcadores fiduciais(marcadores) previamente mapeados. O LightCraft consegue estabelecer então a posição da câmera, analisando a imagem capturada frame a frame pela grande-angular e reconhecendo tais marcadores(Figura 3.1).

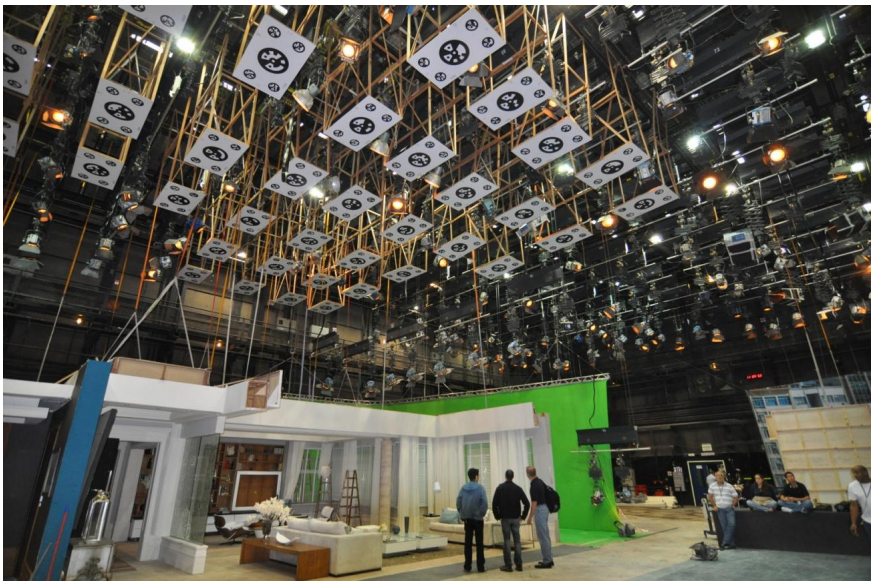

Figura 3.1: *Exemplo de estúdio da (Globo) equipado com marcadores para o uso do LightCraft.*



## 3.2 Motivação e Objetivo do Projeto

Uma das etapas mais trabalhosas e demoradas desse processo era gerar o mapa com as coordenadas tridimensionais dos marcadores. Inicialmente esse mapeamento era feito manualmente com um laser. Cada marcador possui cinco círculos (Figura 3.2), e as coordenadas do centro de cada um deles têm que ser conhecidas. Num cenário como o da figura 3.1, que contém praticamente 50 marcadores, são necessárias cerca de 250 medidas, o que leva em torno de cinco horas de levantamento manual e tem que ser repetido sempre que houver qualquer mudança no teto do estúdio, já que qualquer movimento altera a posição dos marcadores. Isso faz com que o cronograma de gravações tenha que ser muito bem planejado e executado para que não haja atrasos.

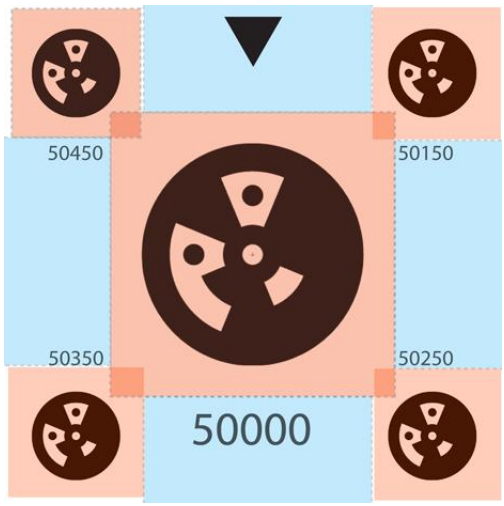


Figura 3.2: *Marcador fiducial do (LightCraft).*

Essa dificuldade foi a motivação para que a Central Globo de Engenharia(CGE) encomendasse o projeto MarkerFinder, software que tem como objetivo automatizar o levantamento do mapa de marcadores.

Para isso usa-se o (Focus), um laser scanner da FARO(Faro) usado para survey(Figura 3.3). Ele faz o escanamento girando sobre um tripé, medindo a distância e a cor de pontos do ambiente. O resultado gerado é uma nuvem de pontos que reconstrói digitalmente todo o ambiente à sua volta. No caso da Globo, o estúdio.

Figura 3.3: *Faro (Focus).*

Para automatizar a geração do mapa de marcadores é necessário, tendo em mãos essa nuvem de pontos, identificar uma subnuvem que represente a parte do teto do estúdio que contenha os marcadores. Isso é feito no SCENE(Fscene), software também da FARO para manipulação dos dados coletados. Essa subnuvem é então transformada em uma malha de

triângulos(mesh) no formato ply que, por sua vez, é o dado de entrada do MarkerFinder. Este, então, a processa e identifica os marcadores, gerando os dados necessários para que o LightCraft possa ser utilizado. Um mapa prévio dos marcadores também é fornecido como dado de entrada. Este consiste em somente um esboçco com a posição relativa entre os marcadores, sem uma escala ou unidades de distância definidas. Este mapa facilita a verificação da corretude do resultado gerado.

## 3.3 Especificação Funcional

### 3.3.1 Requisitos Funcionais

O MarkerFinder, por se tratar de um software com um propósito bastante específico, possui somente três requisitos funcionais, exigidos pela Central Globo de Engenharia:

1. Gerar uma lista de círculos. Cada círculo deverá ter o formato "X Y Z R", onde X, Y e Z são as coordenadas tridimensionais do centro do círculo e R é seu raio.

2. Levar o menor tempo possível para gerar o mapa, sem ultrapassar uma hora para cada $40m^2$ de área de teto. Este é um requisito que implica em um ganho real e muito significativo no tempo de preparação para uma filmagem.

3. O software deve ser escrito em C++ no Microsoft Visual Studio 2010, utilizando o (OpenCV) - biblioteca de Visão Computacional - e deve ser compilado para ambiente Windows.

### 3.3.2 Casos de Uso

Para atender os requisitos funcionais da seção 3.3.1, foram identificados os seguintes casos de uso:

1. **Inicialização**

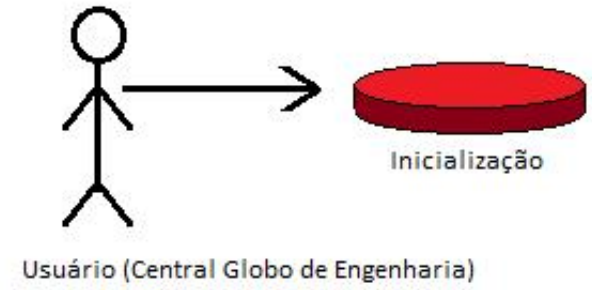


Figura 3.4: *Caso de Uso - Inicialização.*

Este caso de uso consiste em uma única ação: o usuário fornece ao software uma malha de triângulos no formato ply e um mapa (esboço) da disposição relativa dos marcadores entre si.

2. **Calibragem de parâmetros e detecção de círculos**


Figura 3.5: *Caso de Uso - Calibragem de parâmetros e detecção de círculos.*

O software exibe na tela a imagem gerada e os círculos encontrados com os valores padrão para os parametros do algoritmo de Hough(HOUGH e MICH, 1962), estes que são: raios mínimo e máximo dos círculos; distância mínima entre um círculo e outro (Accumulated Id) e tolerância a ruídos na imagem (Canny Treshold).

O usuário pode ajustar esses parâmetros enquanto o resultado não for satisfatório. Uma vez que todos os círculos correspondes aos marcadores estejam detectados em tela, o usuário aceita o resultado e o software gera o mapa de marcadores no formato exigido.

## 3.4 Algoritmo

O algoritmo do MarkerFinder, em linhas gerais, está descrito nas seções a seguir:

1 Projetar a malha tridimensional de triângulos do teto do estúdio em uma bidimensional, gerando uma imagem, mas guardando as coordenadas de profundidade (z) dos pontos;
2 Detectar círculos com o algoritmo de Hough(HOUGH e MICH, 1962);
3 **while** *usuário nao estiver satisfeito com o resultado da detecção* **do**
4 | Exibir a imagem ao usuário com os circulos encontrados;
5 | Receber novos parametros de detecção;
6 | Redetectar;
7 **end**
8 Gerar o mapa de marcadores, a partir dos círculos detectados e de um esboço do mapa dado de entrada;

### 3.4.1 Algoritmo Detalhado

Os dados de entrada do MarkerFinder são o arquivo que contém a malha de triângulos que representa o teto do estúdio e o esboço do mapa de marcadores, que é uma matriz com a disposição dos marcadores. Alem disso, também é possível definir o número médio de triângulos da malha original que se deseja ter por pixel da imagem a ser gerada para a aplicação do algoritmo de Hough. Quanto maior é este número, mais triângulos caberão dentro de um pixel e mais rápido o processamento. Evidentemente, este ganho em tempo implica em uma perda de precisão.

O algoritmo, em detalhes, é o seguinte:

1 Encontrar as coordenadas X e Y máximas e mínimas da malha recebida(MinX, MaxX, MinY, MaxY);
2 Calcular a resolução W:H da imagem sobre a qual se aplicará a detecção de círculos, usando W = MaxX - MinX, H = MaxY - MinY e o número médio de triângulos por pixel estipulado;
3 Criar uma imagem I em branco, de resolução W:H;
4 Projetar a malha 3D em uma 2D(copiar a malha 3D ignorando a coordenada Z de todos os pontos);
5 **for** *each triângulo T da malha 2D* **do**
6 Calcular a bounding box mínima BB da imagem I sobreposta que contenha T(Figura 3.6);
7 **for** *each pixel P em BB* **do**
8 **if** *o centro C de P estiver contido em T* **then**
9 coordenada Z de P ⟵ coordenada Z de C projetado na malha 3D original;
10 cor de P ⟵ cor de T;
11 **end**
12 **end**
13 Detectar círculos na imagem I com (HOUGH e MICH, 1962);
14 **while** *usuário não satisfeito com o resultado* **do**
15 Receber novos parametros de detecção de círculos;
16 Redetectar;
17 **end**
18 Gerar o mapa, comparando a lista de círculos detectados com a matriz de disposição de marcadores;
19 Retornar o mapa;

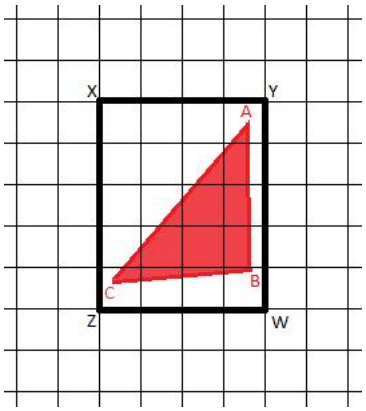


Figura 3.6: *XYWZ é a menor bounding box do grid que contém o triângulo ABC.*

## 3.5 Modelagem de Classes

O estudo da demanda pelo projeto frente ao algoritmo da seção 3.4 identificou seis classes, como mostra o diagrama de classes da figura 3.7. Os argumentos das funções foram omitidos no diagrama por questão de espaço, mas estão descritos em detalhes na subseção 3.5.1.

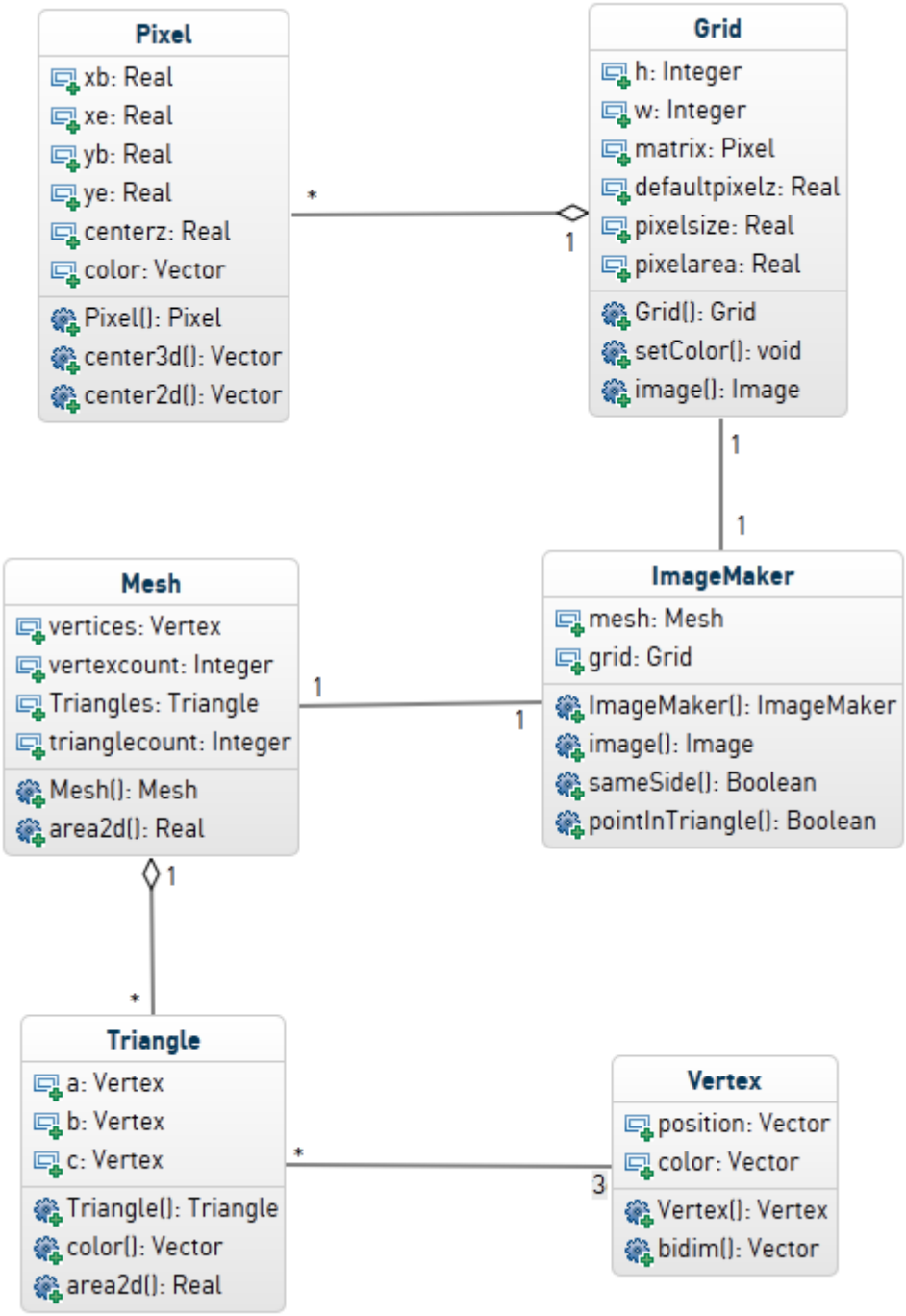


Figura 3.7: *Diagrama de Classes do MarkerFinder.*

### 3.5.1 Descrição detalhada de Classes e Atributos

A descrição detalhada de cada classe, atributo e método(função) segue abaixo. É importante ressaltar que, pela motivação e maior prioridade do projeto serem minimizar o tempo do processo de qualquer maneira, todos os atributos e métodos das classes são públicos.

Os nomes dos atributos estará em azul e, os dos métodos e seus argumentos, em vermelho.

- **Vertex**
  É um vértice (ponto) no espaço 3D.

  **Position** é o vetor (X,Y,Z).
  **Color** é a cor RGBA do ponto.
  **Vertex(x, y, z, r, g, b, a)** é o método construtor. **x**, **y** e **z** são as coordenadas tridimensionais do vértice. **r**, **g**, **b**, **a** são as componentes RGBA da cor do vértice.
  **Color()** é a cor RGB.
  **bidim()** retorna as coordenadas bidimensionais do ponto (X e Y), ou seja, o ponto projetado no plano onde a coordenada Z é zero.

- **Triangle**
  É um triangulo no espaço 3D.

  **A**, **B** e **C** são os vértices (Vertex) do triângulo.
  **Triangle(a, b, c)** é o Método construtor. **a**, **b** e **c** são os tres vertices que definem um triangulo.
  **Color()** é a cor RGB do triangulo.
  **area2d()** é a área do triângulo projetado em 2d, ou seja, ignorando a coordenada Z dos vértices.

- **Mesh**
  É uma malha de triângulos gerada previamente em formato ply.

  **vertices** é a lista dos vértices da malha.
  **vertexcount** é a quantidade de vertices em vertices.
  **triangles** é a lista dos triangulos da malha.
  **trianglecount** é a quantidade de triangulos em vertices.
  **Mesh(meshfile)** é o método construtor. **meshfile** é o arquivo ply que contém os dados da malha.
  **area2d()** é a área da malha, quando projetada em um plano, ou seja,

quando se ignoram as coordenadas Z de todos os pontos. Assume-se que, por já se tratar de um teto de estúdio razoavelmente plano, não exitem dois pontos com as mesmas coordenadas X e Y. Sendo assim, não há risco significativo dessa projeção ter pontos coincidentes.

– **Pixel**

Representa um pixel da imagem sobre a qual será aplicada o algoritmo de Hough.

**xb** significa "x begin", é a coordenada x mínima do pixel.
**xe** significa "x end", é a coordenada x máxima do pixel.
**yb** significa "y begin", é a coordenada y mínima do pixel.
**ye** significa "y end", é a coordenada y máxima do pixel.
**centerz** é a coordenada Z do centro do pixel. Corresponde à coordenada z do ponto da malha de triangulos que coincide com o centro do pixel.
**color** é a cor RGB do pixel.
**Pixel(xb, xe, yb, ye, color, z)** é o método contrutor. **xb**, **xe**, **yb** e **ye** são as coordenadas dos limites do pixel(figura 3.8), **color** é um vetor de quatro dimensões que contem as componentes RGBA da cor do pixel e **z** é a coordenada de profundidade do centro do pixel.
**center3d()** é a posição (X,Y,Z) do centro do pixel.
**center2d()** é a posição (X,Y) do centro do pixel.

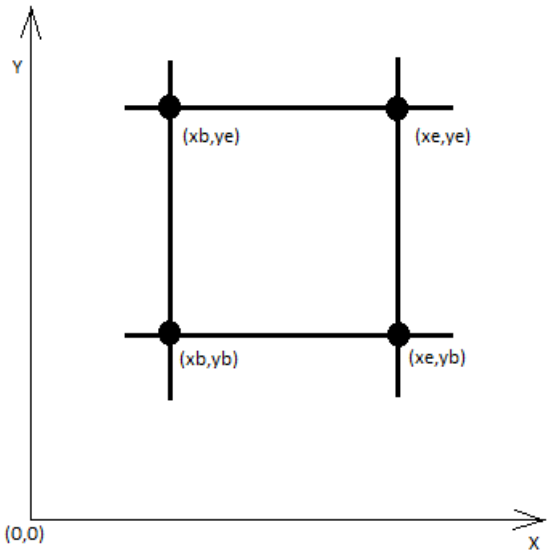


Figura 3.8: *Limites do Pixel.*

– **Grid**

É a matriz bidimensional de pixels que dá origem à imagem sobre a qual se aplicará o Hough.

**h** é a altura, em pixels, do grid.
**w** é a largura, em pixels, do grid.
**matrix** é a matriz de fato.
**defaultpixelz** é a coordenada Z atribuída inicialmente aos pixels quando estes são inicializados.
**pixelsize** é o tamanho do lado do pixel (**Pixel.xe** - **Pixel.xb**).
**pixelarea** é a área do pixel (**pixelsize** * **pixelsize**).
**Grid(h, w, Pixelsize, xmin, ymin, defaultcolor, defaultz)** é o método construtor. **h** e **w** são as dimensões da matriz. **xmin** e **ymin** são as coordenadas do início da matriz, ou seja, sua Origem. **defaultcolor** é a cor inicial para os pixels a serem instanciados. **defaultz** é a coordenada z inicial que os pixels receberão na sua instanciação.
**setColor(i, j, color)** atribui a cor **color** ao pixel de índice [**i**][**j**] da matriz.
**image()** constroi a imagem a partir do grid pronto.

– **ImageMaker**

É a classe responsável por receber a malha (Mesh) e gerar um Grid. Com o grid pronto é possível gerar a imagem.

**mesh** é a malha recebida
**grid** é o grid a ser gerado
**ImageMaker(mesh, defaultcolor, trianglesperpixel)** é o método construtor. **mesh** é a malha de triangulos a ser processada; **defaultcolor** é a cor inicial dos pixels do Grid a ser instanciado e **trianglesperpixel** é o numero médio estipulado de triangulos da malha por pixel do Grid. Quanto maior este número, pior é a qualidade da imagem gerada e mais rápido o processamento.
**image()** gera a imagem
**sameSide(p1, p2, a, b)** determina se os pontos **p1** e **p2** estão do mesmo lado da reta **ab**.
**pointInTringle(p, t)** determina se o ponto **p** está dentro do triângulo **t**.

## 3.6 Detalhes de Implementação e Testes

Seguindo os requisitos funcionais(seção 3.3.1), o software foi desenvolvido em C++ com o framework (OpenCV) no Microsoft Visual Studio 2010. Cada uma das seis classes foi declarada e implementada em um par de arquivos .cpp e .h. Além delas, existe um módulo main.cpp, que contem a função principal. O código está descrito e comentado na íntegra em `www.icad.puc-rio.br/~gameshow/MarkerFinder/sources/`. Os arquivos XML de configuração do Visual Studio 2010 que ele necessita para ser compilado estão em `icad.puc-rio.br/~gameshow/MarkerFinder/configFilesVStudio/`.

Para cada uma das classes foi feita uma classe de mesmo nome seguido de "Tester"(GridTester para a classe Grid, por exemplo), que contém os casos de teste para sua classe de origem. Esses casos de teste foram usados juntamente com o módulo de UNIT Testing BOOST Test(Boost). Mais sobre testes em `icad.puc-rio.br/~gameshow/MarkerFinder/MarkerFinderTester/`. Foi usado o módulo time.h para a tomada de tempo da execução do software.

Pelo fato do usuário ter conhecimento técnico avançado na área, não foi necessário o desenvolvimento de nenhuma interface gráfica para o software, com exceção da janela para a detecção dos círculos. O software é uma aplicação de console. Durante o curso de uma execução sem erros, o usuário é informado das seguintes etapas, em ordem:

1. Hora do início da execução
2. Início e final de leitura de arquivo de Mesh
3. Início e final de cálculo de área da projeção bidimensional da malha
4. Inicio e final de cálculo das mínimas e máximas coordenadas X e Y da malha
5. Inicio e final do cálculo das cores dos pixels da imagem correspondente à malha
6. Início e final do processo de geração da imagem a partir do Grid pronto
7. Hora de final de execução
8. Tempo total de execução

Depois destas mensagens, é exibida na tela a imagem e o usuário inicia o caso de uso Calibragem de parâmetros e detecção de círculos (seção 3.3.2). As figuras 3.9 e 3.10 ilustram a imagem com os círculos detectados antes e depois do ajuste, respectivamente.

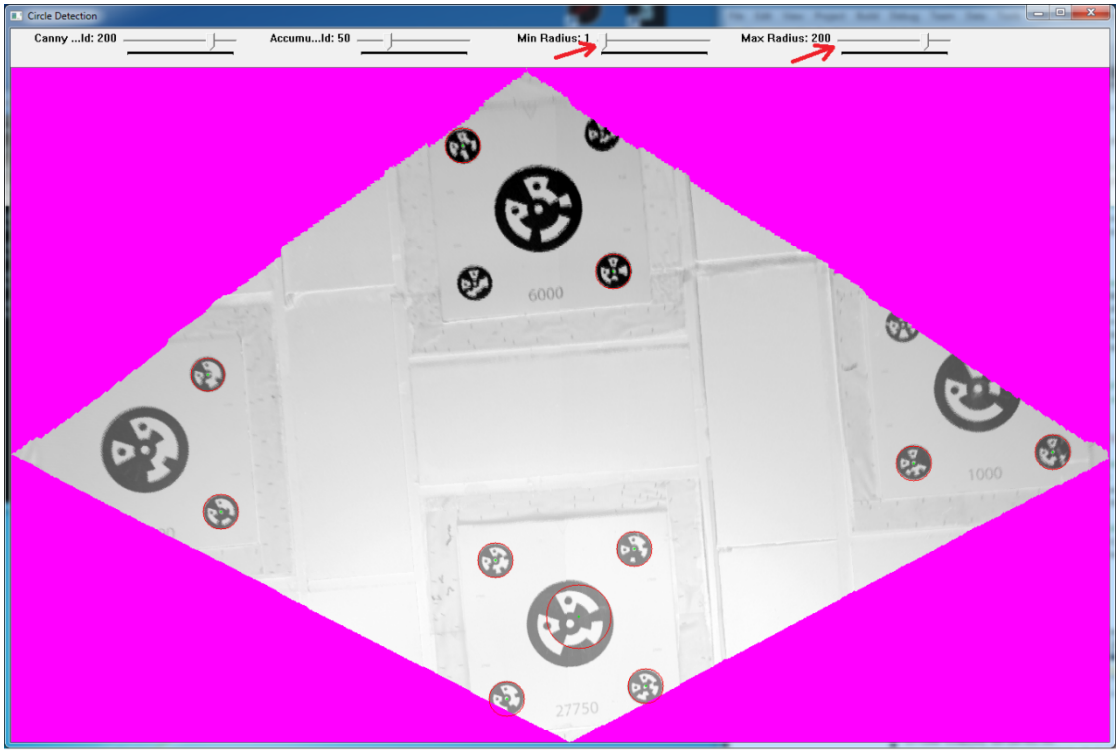


Figura 3.9: *Detecção de círculos antes do ajuste dos parâmetros de Hough.*

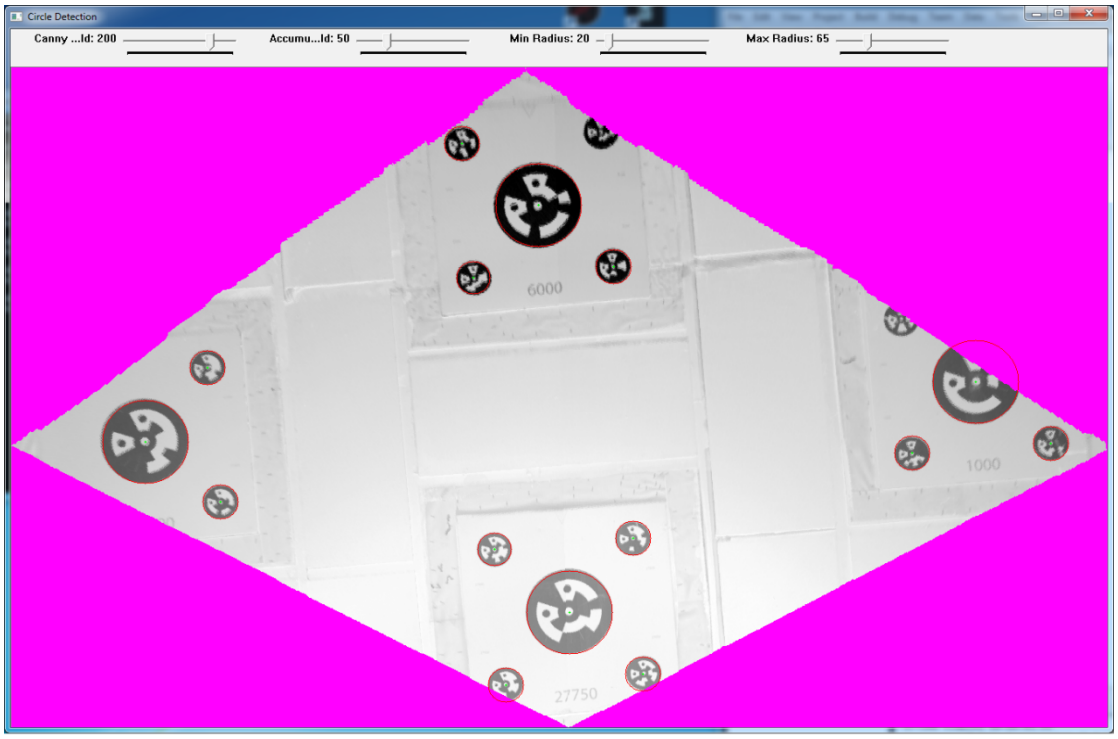


Figura 3.10: *Detecção de círculos depois do ajuste manual dos parâmetros de Hough.*

As únicas duas funções implementadas que não foram descritas na seção 3.5.1 são:

– std::vector<Vec3f>HoughDetection(): recebe a imagem convertida para tons de cinza e os parametros exigidos pelo algoritmo de Hough e retorna a lista de círculos encontrados. É esta função que é executada a cada novo ajuste de parâmetros feito pelo usuário.

– int main(int argc, char** argv): é a função principal da aplicação. Recebe o nome do arquivo ply com a malha de triângulos como entrada e exibe a lista dos círculos detectados depois do processamento.

### 3.6.1 Testes

Foram implementados, como um projeto em Visual Studio a parte, os testes dos módulos do MarkerFinder, que foi chamado de MarkerFinderTester. É neles que estão definidos os módulos de testes para cada classe.

Foi utilizada a ferramenta BOOST Test do framework BOOST(Boost) em conjunto com esses módulos para execução do script de testes formulado.

Foram identificados 36 casos de teste relevantes. Por toda a execução da aplicação depender da integridade do arquivo de entrada, existem vários pontos no fluxo de execução que demandam atenção à consistência dos dados. Estes casos de teste forçam os erros possíveis para assegurar que a aplicação seja capaz de identificá-los e, se necessário, abortar a execução; nunca gerando um mapa de coordenadas inconsistente.

Vale ressaltar que as mensagens geradas pela aplicação em caso de erro são membros da classe ExceptionMessages. Os códigos dos módulos de teste estão em `icad.puc-rio.br/~gameshow/MarkerFinder/MarkerFinderTester/`. Os casos de teste estão a seguir:

– **Testes da classe Pixel**

1. pixel_x_null_interval_test: tenta instanciar um Pixel de largura nula.
2. pixel_x_negative_interval_test: tenta instanciar um pixel de largura negativa.
3. pixel_y_null_interval_test: tenta instanciar um Pixel de altura nula.
4. pixel_y_negative_interval_test: tenta instanciar um Pixel de altura negativa.
5. pixel_null_color_test: tenta instanciar um Pixel passando um ponteiro NULL como vetor de cor.

6. pixel_not_square_test: tenta instanciar um Pixel não quadrado, ou seja, de larguras e alturas diferentes.

– **Testes da classe Grid**

1. grid_w_zero_test: tenta instanciar um Grid de largura nula.
2. grid_w_negative_test: tenta instanciar um Grid de largura negativa.
3. grid_h_zero_test: tenta instanciar um Grid de altura nula.
4. grid_h_negative_test: tenta instanciar um Grid de altura negativa.
5. grid_pixelsize_zero_test: tenta instanciar um Grid passando tamanho de pixel nulo como parâmetro.
6. grid_pixelsize_negative_test: tenta instanciar um Grid passando tamanho de pixel negativo como parâmetro.
7. grid_defaultcolor_null_pointer_test: tenta instanciar um Grid passando um ponteiro NULL como vetor de cor inicial de Pixels.

– **Testes da classe Triangle**

1. triangle_a_null_pointer_test: tenta instanciar um Triangle com ponteiro NULL para vértice a.
2. triangle_b_null_pointer_test: tenta instanciar um Triangle com ponteiro NULL para vértice b.
3. triangle_c_null_pointer_test: tenta instanciar um Triangle com ponteiro NULL para vértice c.
4. triangle_a_b_identical_test: tenta instanciar um Triangle com ponteiros idênticos para os vértices a e b.
5. triangle_a_c_identical_test: tenta instanciar um Triangle com ponteiros idênticos para os vértices a e c.
6. triangle_b_c_identical_test: tenta instanciar um Triangle com ponteiros idênticos para os vértices b e c.
7. triangle_a_b_c_identical_test: tenta instanciar um Triangle com ponteiros idênticos para os vértices a, b e c.

– **Testes da classe Mesh** Os testes da classe Mesh precisam de arquivos de entrada propositalmente corrompidos para sua realização.

1. mesh_empty_filename_test: tenta instanciar uma Mesh passando uma string vazia como nome de arquivo de entrada.

2. mesh_corrupted_missing_file_test: tenta instanciar uma Mesh com um nome de arquivo inexistente.
3. mesh_invalid_header_test: tenta instanciar uma Mesh com um arquivo com cabeçalho fora do formato estipulado.
4. mesh_line_number_mismatch_test: tenta instanciar uma Mesh com um arquivo com dados inconsistentes com o cabeçalho.

– **Testes da classe ImageMaker**

1. imagemaker_zero_triangles_pixel_test: tenta instanciar um ImageMaker com zero como número médio de triangulos por pixel.
2. imagemaker_negative_triangles_pixel_test: tenta instanciar um ImageMaker com número médio de triangulos por pixel negativo.
3. imagemaker_mesh_null_pointer_test: tenta instanciar um ImageMaker com um ponteiro NULL para Mesh.
4. imagemaker_color_null_pointer_test: tenta instanciar um ImageMaker com um ponteiro NULL para cor inicial dos Pixels.
5. imagemaker_same_side_null_p1_test: testa o comportamento da função sameSide se ela receber um ponteiro NULL para o ponto p1.
6. imagemaker_same_side_null_p2_test: testa o comportamento da função sameSide se ela receber um ponteiro NULL para o ponto p2.
7. imagemaker_same_side_null_a_test: testa o comportamento da função sameSide se ela receber um ponteiro NULL para o ponto a.
8. imagemaker_same_side_null_b_test: testa o comportamento da função sameSide se ela receber um ponteiro NULL para o parâmetro b.
9. imagemaker_same_side_identical_p1_p2_test: testa o comportamento da função sameSide se ela receber ponteiros idênticos para p1 e p2.
10. imagemaker_same_side_identical_a_b_test: testa o comportamento da função sameSide se ela receber ponteiros idênticos para a e b.
11. imagemaker_point_triangle_null_pointer_p_test: testa o comportamento da função pointInTriangle se ela receber ponteiro NULL para o ponto p.

12. imagemaker_point_triangle_null_pointer_t_test: testa o comportamento da função pointInTriangle se ela receber ponteiro NULL para o triângulo t.

O script de testes executa cada um dos casos anteirores em ordem e compara o resultado com a mensagem de erro esperada. No momento, todos os testes estão tendo o resultado esperado.

### 3.6.2 Documentação

O código foi amplamente comentado usando o formato aceito para geração automática de documentação com o (Doxygen).

## 3.7 Adaptabilidade

Para acomodar possíveis mudanças nos requisitos ou exigências novas de implementação, o projeto está atualmente sendo desenvolvido com ferramenta (Git) para o controle de versão. Os módulos de testes foram implementados pensando em minimizar o retrabalho no caso de mudanças no código que impliquem em alterações, remoções ou inserções de casos de teste.

## 3.8 Resultados

O projeto está em produção e cumpre os requisitos especificados na seção 3.3.1. O desempenho ficou em níveis considerados ótimos, mesmo quando se tem milhões de vértices.

A aplicação demora cerca de vinte minutos para processar uma malha de 3,4 milhões de vértices e 6,75 milhões de triângulos, o que corresponde a uma área de 33 metros quadrados, aproximadamente. Esse tempo escala linearmente com o número de triângulos da malha, ou seja, uma malha de 13,5 milhões de triângulos demoraria cerca de 40 minutos para ser processada. O computador usado para a medição foi um Intel Core i7-3770K, com quatro núcleos a 3.5GHz cada e 32GB de memória RAM, rodando Windows 7.

# 4
# Bons Jogos e bons Game Shows

O que determina o sucesso e o apelo de uma obra está diretamente relacionado à mídia em que ela é transmitida. As frequentes adaptações de obras literárias para o cinema e, mais raramente, o contrário são provas de que é impossível mudar o meio sem mudar a mensagem.

Por este documento tratar da convergência entre jogos de videogame e game shows e por existir pouco estudo e literatura sobre esta convergência, é impossível ter certeza sobre o sucesso de uma potencial obra que se transmita através destas duas mídias. Um estudo do que faz uma obra ter sucesso em cada uma destas áreas é um ponto inicial importante, mesmo que se prove no futuro que os requisitos do sucesso da convergência não se relacionem tão intimamente com os das áreas em separado.

Este capítulo investiga os aspectos que fazem videogames e game shows de TV obterem sucesso de público. O estudo de aplicativos móveis foi deixado de lado pois não se enquadra. O aplicativo, no caso deste projeto, é somente uma interface para um telespectador participar do jogo. Por esta razão, o que é necessário para ele é ter boa usabilidade. O aplicativo móvel, dentro do projeto proposto, ocupa o lugar do controle do videogame.



## 4.1
## Bons Jogos

Um dos conceitos da Psicologia que mais se aplica na criação de apelo de um jogo é o *Flow*, termo que será mantido em inglês pela ausência de uma tradução precisa. Proposto por Mihaly Csikszentmihalyi(Csikszentmihalyi, 1990), este conceito diz respeito à realização de uma atividade qualquer em um estado de imersão completa, causada pela motivação. Uma pessoa que, ao realizar uma atividade, está tão em foco e tão imerso a ponto de não prestar atenção em mais nada, encontra-se neste estado.

Para identificar, ou ainda, diagnosticar que uma pessoa está em *Flow*, Csikszentmihalyi propõe a observação de seis fatores:

- Pensamento focado no momento presente, com a mente sem conjecturas.
- Dissolução e mistura da conciência com a realização da ação.

– Silenciamento da auto-consciência, ou do pensamento em si mesmo.
– Sensação de se estar no controle da atividade realizada.
– Percepção de tempo alterada. Geralmente o executor da atividade percebe o tempo passar mais devagar do que quando fora de *Flow*.
– A realização da atividade é, em si, recompensadora. O estado de *Flow* é marcado pelo prazer na realização da atividade pela atividade.

O sexto fator, embora não suficiente por si só, é o mais relevante para o desenvolvimento de jogos. O ato de jogar um jogo tem que ser sua própria recompensa. Por mais que jogos possam ter prêmios em sua conclusão, um jogador que só os persegue sem se divertir com o jogo em si não está jogando, mas sim trabalhando. A palavra *play*, que em inglês siginifica algo entre jogar e brincar, descreve uma atividade que é recompensa de si proópria(Schell, 2008).

Saber diagnosticar uma atividade como provocadora de *Flow* pode ser suficiente para seu estudo. Para desenvolver um jogo, todavia, é necessário saber criar uma experiência que o provoque. Ainda em (Csikszentmihalyi, 1990), é sugerido que, para uma atividade provocar este estado, são necessárias três condições: ter um conjunto claro de objetivos; responder imediatamente às interações do realizador, para que este possa se ajustar e se manter imerso; e um equilíbrio entre as habilidades do executor e a dificuldade das tarefas. A dificuldade deve ser um pouco maior que a média das atividades do cotidiano, ela deve representar um desafio. Quanto maior o desafio, mais intenso é o estado de *Flow*. Entretanto, se muito difícil, a atividade acaba gerando frustração.

Ao realizar uma tarefa, ou mais especificamente ao se jogar um jogo, o jogador melhora suas habilidades com a prática e o desafio diminui. Cabe ao desenvolvedor do jogo fazer com que o desafio escale com o tempo. Jesse Schell(Schell, 2008) examina duas maneiras de escalar o desafio(figura 4.1). A primeira propõe uma escalada linear de acordo com o aumento de habilidade do jogador, de modo que ele perceba o jogo sempre como um desafio constante. A segunda maneira, que o autor prefere, sugere uma subida oscilante, em que a média dos desafios seja linear. Desta maneira o jogador alternaria entre momentos de dificuldade seguidos de momentos de sensação de superioridade.

A noção de desafio, no entanto, não corresponde aos mesmos traços em todos os gêneros. Gêneros distintos possuem públicos com objetivos muito diferentes.

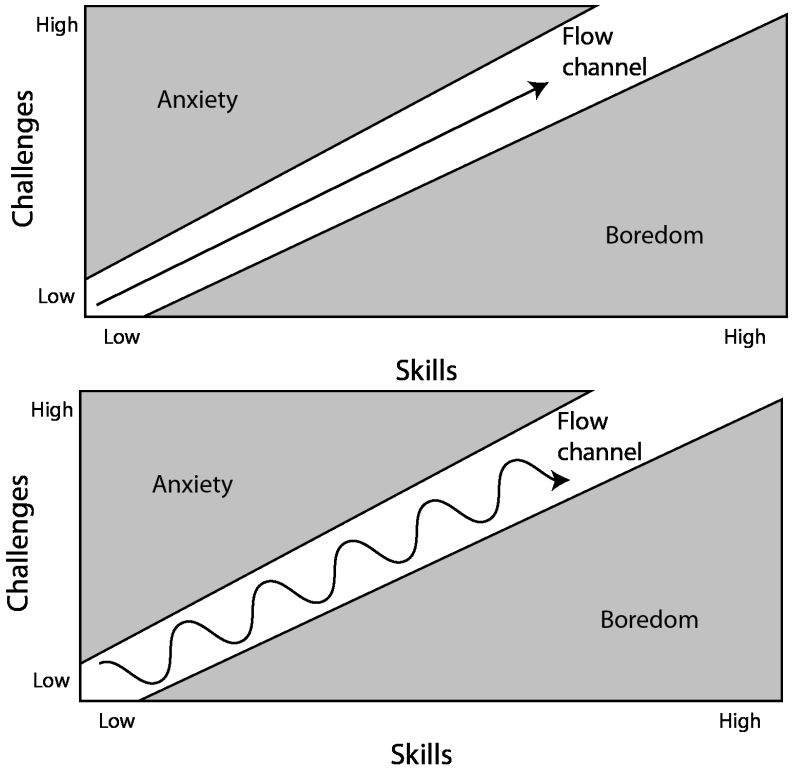


Figura 4.1: *Escalada linear de desafios em cima e oscilante embaixo(Schell, 2008).*

Em (Apperley, 2006), T. H. Apperley propõe uma divisão dos videogames em quatro gêneros: Simulação; Estratégia; Ação e RPG. O artigo tem como objetivo classificar os jogos em gêneros que sejam compatíveis com ambas as opostas correntes ludologista e narratologista; cada uma com a sua ótica de estudo dos jogos. A primeira corrente é o estudo do jogo e da atividade de jogar, onde ele é considerado como tendo uma forma própria de narrativa. Ludologistas acreditam que a teoria de narrativa não é inteiramente relevante para se entender jogos. A segunda corrente parte da teoria de narrativa para entendê-los. A posição desta dissertação está praticamente no meio termo entre as duas correntes, tendendo mais para a ludologia. Esta postura intermediária também é defendida por pesquisadores de transmídia (Jenkins 2006).

Por esta abordagem, além destes quatro gêneros propostos em (Apperley, 2006), cabe a consideração de mais dois: puzzle e jogos motores. O primeiro apresenta desafios de ordem mais mental ao jogador. O segundo consiste em jogos cujo principal apelo é entreter quase que exclusivamente pela interação motora.

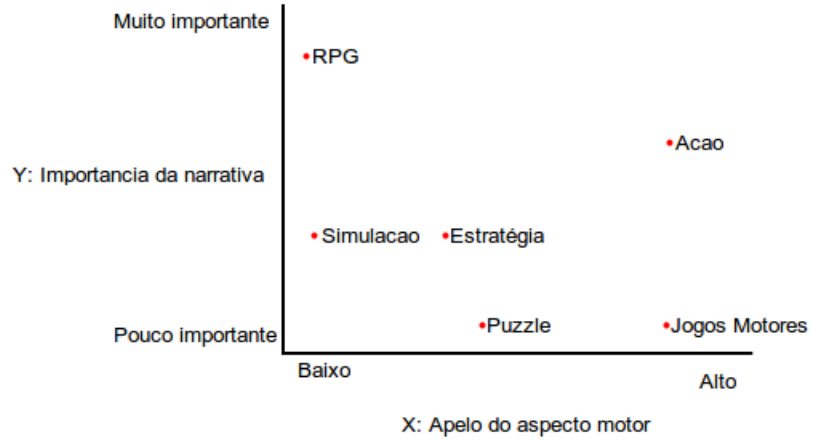


Figura 4.2: *Gêneros de acordo com a expectativa de seus públicos.*

A figura 4.2 mostra a importância que o aspecto motor e a narrativa tem em cada um destes gêneros. É importante ressaltar que qualquer jogo pode se beneficiar de uma boa narrativa e de uma boa experiência motora, mas nem sempre uma ou outra é crítica para seu sucesso.

Não é importante abordar a influência dos aspectos visuais dos jogos neste estudo. Estes são importantíssimos para o apelo de um jogo mas são consequências de escolhas criativas feitas anteriormente. Uma vez escolhido o gênero e a modelagem inicial, escolhe-se a identidade visual. Os elementos gráficos se adequam ao jogo e não o contrário. A constante evolução da computação gráfica e o barateamento do poder computacional fizeram com que eles deixassem de ser uma limitação. O que antes podia ser uma escolha limitada pela performance, a identidade visual de um jogo, hoje, é uma escolha quase que puramente estética.



### 4.1.1 Gêneros e seus casos de sucesso

Esta seção apresenta uma descrição de cada gênero introduzido na seção anterior com exemplos de jogos de sucesso, bem como uma análise dos atributos determinantes de tal sucesso. Dentro de cada um é avaliada a narrativa e a qualidade de seus aspectos motores de interação.

#### RPG

Gênero que já fazia sucesso antes dos videogames, o Role-Playing Game tem a narrativa em primeiro plano. Diferente de jogos em que os feitos do jogador simplesmente avançam uma história linear, no RPG a ênfase é em

construir a narrativa de acordo com as personalidades dos protagonistas. Tendo seu início no final da década de 1960 e com sua principal franquia, Dungeons and Dragons(Gygax e Arneson), lançada em 1974; o RPG de mesa consiste em um grupo de jogadores e um mestre. Cada jogador tem seu personagem com habilidades próprias e, mais importante, personalidade própria. O mestre tem o papel do narrador, descrevendo a aventura para os jogadores que tomam decisões de acordo com as características de seus personagens. Embora a narrativa seja escrita para permitir ampla escolha dos jogadores, a liberdade deles não raramente leva a situações imprevistas. Ao mestre cabe adaptar a aventura frente a tais ocorrências e mediar qualquer questão entre jogadores ou personagens.

Com a popularização dos videogames, naturalmente o RPG eletrônico foi se tornando um gênero de grande presença e grande número de apreciadores. Todos os eventos no RPG de mesa são decicidos por uma mistura de dados e habilidades de cada personagem. Ao atacar um monstro, por exemplo, um jogador joga um dado. O resultado, comparado com o fator de defesa do monstro, determina se o ataque acertou ou não. Este processo pode ser longo e complexo, de acordo com a quantidade de fatores a serem levados em consideração. Já no RPG eletrônico esta barreira é vencida silenciosamente e o jogo parece fluir mais. A desvantagem é a ausência de um narrador humano, que diminui bastante as possíveis bifurcações narrativas. Enquanto o mestre pode adaptar a história como achar melhor, o RPG de computador está preso a um número de possibilidades finitas e predeterminadas. Experimentos em geração automática de narrativa são notáveis e possuem grande potencial para melhorar esta questão, mas esta tecnologia ainda não está muito presente no mercado. Exemplos destes experimentos são o LogTell(Ciarlini et al., 2005), o OzProject(Bates et al., 1994) e o PaSSAGE(Thue et al., 2007).

Evidentemente, os fatores de sucesso para um videogame de RPG são ter uma narrativa atraente e alto poder de escolha, tanto estratégica quanto moral. Quanto mais opções existirem, e quanto mais destas opções complementarem as possíveis personalidades dos protagonistas modelados pelos jogadores, mais imersos estes ficarão.

Dois exemplos de sucesso de RPGs eletrônicos são a franquia Diablo, que atualmente está em seu terceiro jogo(Diablo3, 2012) e o World of Warcraft(WoW, 2004), ambos da (Blizzard). O segundo é um *Massively Multiplayer Online RPG*, ou MMORPG. Ele é jogado exclusivamente online por milhares de jogadores desde 2004. Estes dois jogos são exemplos de um estilo mais clássico, com um enredo medieval e uma câmera isométrica que filma de cima pra baixo, como mostra a figura 4.3.


Figura 4.3: *World of Warcraft(WoW, 2004) à esquerda, Diablo 1 à direita, lançado em 1996 e Diablo 3 embaixo, lançado em 2012(Diablo3, 2012). Nota-se um aprimoramento de uma mesma mecânica.*

Outros dois exemplos com uma mecânica diferente do padrão são as séries Elder Scrolls(ElderScrolls), que teve o jogo Skyrim(Skyrim, 2011) como seu lançamento mais recente(figura 4.4) e Fallout(Fallout, 2010). Ambas são desenvolvidas pela (Bethesda) e são jogos em primeira pessoa, ou seja: a câmera do jogo simula os olhos do protagonista. Com um passo um pouco mais rápido que os RPGs clássicos, estes exemplos também contam com elementos de ação. Em relação à narrativa e ao poder de escolha, o Fallout New Vegas(figura 4.5), que é o lançamento mais recente da série, conta com elementos de narrativa interativa bastante inovadores.

Figura 4.4: *(Skyrim, 2011).*

Figura 4.5: *(Fallout, 2010) New Vegas, passado em uma Las Vegas pós guerra nuclear.*



## Simulação

Tendo sua origem nas simulações de computador para o estudo de sistemas e modelos sérios, os jogos de simulação abordam algum aspecto do mundo modelado por uma ótica menos compromissada com a realidade e mais com a diversão. Simulações sérias tipicamente emulam as consequências de um cenário com parâmetros específicos. Já os jogos deste gênero não contam com tais parâmetros no início de sua execução, mas ao longo do tempo, fornecidos como os comandos do jogador.

Este gênero foi subdividido neste documento em duas grandes correntes: atividade e administração. A primeira diz respeito à simulação de uma atividade normalmente executada por uma só pessoa. Simuladores de vôo e de outros veículos se enquandram nesta categoria. Ja os jogos de administração abordam atividades exercidas por um grupo de pessoas, ou até uma sociedade inteira. O jogador administra cidades, hospitais, times de futebol e famílias, por exemplo. Neste tipo, muitas vezes o papel dado ao jogador não corresponde ao de uma pessoa real. Na série (SimCity), por exemplo, cabe ao jogador simular uma cidade de sucesso. As ferramentas disponíveis para tal vão muito além

das que um administrador real de cidades possui. Um prefeito não escolhe diretamente o valor das propriedades residenciais nem o tipo específico de indústria de sua cidade, ao contrario do jogador. Este ponto é uma distinção importante entre os jogos de atividades e administração. Enquanto os jogadores esperam o máximo de realismo ao simular um avião(figura 4.6), o sucesso de um jogo de administração não é necessariamente maior quanto mais realista ele for.

Figura 4.6: *Setup de computador pessoal para simulador de vôo(FlightSim, 2006). Embora os desenvolvedores não esperem esse nível de todos os jogadores, é exigido do jogo o máximo de realismo.*

Manter o jogador no estado de *Flow* é especialmente difícil em um jogo de administração. Por não contar com controles imersivos e por não haver um roteiro definido, o desenvolvedor tem pouco controle sobre o usuário e sobre a sequência de eventos do próprio jogo. Como dito no início da seção 4.1, para manter uma pessoa em *Flow* é necessária uma tarefa com objetivos claros e resposta rápida. Neste caso, um jogador pode se frustar em meio às inúmeras mecânicas e eventos possíveis em uma simulação de cidade, por exemplo. O número de elementos realistas ideal para o sucesso deste sub-gênero é um que seja pequeno o suficiente para não confundir o jogador, mas grande o suficiente para que não se perca do tema proposto.

No que diz respeito à importância dos aspectos motores, os dois sub-gêneros também são muito distintos. Jogos de atividade, evidentemente, tem que apresentar controle fiéis à realidade. Os de adminsitração, que são maioria, geralmente são controlados com mouse e atalhos de teclado, bastando que estes não dificultem a experiência.

Um exemplo de jogo de atividade de grande sucesso é a série Flight Simulator(FlightSim, 2006) da Microsoft(figura 4.7). Se tratando de jogos

de administração, da mesma empresa do SimCity(figura 4.8), a série The Sims(Sims) é um sucesso de vendas desde 2000. O primeiro jogo da série, quando lançado, se tornou o jogo eletrônico mais vendido de todos os tempos, com 11.3 milhões de cópias(figura 4.9).

Figura 4.7: *Microsoft Flight Simulator X. Avião se aproxima da cidade do Rio de Janeiro(FlightSim, 2006).*


Figura 4.8: *(SimCity). Cidade administrada por um jogador.*

Figura 4.9: *The (Sims). Jogador é responsável por gerir uma casa e a vida de seus moradores.*

### Estratégia

Desde o Xadrez e de outros jogos de tabuleiro que os jogos de estratégia estão no cotidiano das pessoas. Um jogo de estratégia é marcado pela necessidade de se pensar à frente e no grande impacto que cada decisão tem no rumo do jogo. Geralmente ele é jogado por dois ou mais competidores.

Jogos eletrônicos de estratégia se dividem em dois grandes grupos: os jogos em tempo real(*Real Time Strategy*, ou RTS) e os jogos em turnos(*Turn Based Strategy*, ou TBS). Enquanto, no primeiro, todos os competidores jogam ao mesmo tempo; no segundo, os competidores alternam os movimentos em turnos.

(Jones, 2011) trata das características que fazem um jogo de estratégia ser excitante, o que, para o autor, é a característica essencial para se ter sucesso de público. São elas:

- **Ter um conflito ativo com um resultado imprevisível.** A excitação ao se jogar um jogo vem da incerteza dos jogadores.
- **Incentivar o início deste conflito o mais cedo possível.** O jogo tem que deixar claro para os participantes que evitar o conflito, ou a preparação para ele, se traduzirá em uma desvantagem futura.
- **Quanto mais próximo do final do jogo, mais imprevisível ele deve ser.** Embora esta regra não seja uma unanimidade nos jogos de estratégia, ela é bastante relevante. Se no final do jogo, o vencedor ja estiver claro, ele deixa de ser excitante. No caso específico do trabalho proposto nesta dissertação, isto é importantíssimo. Como a mídia é um programa de TV que precisa de público tanto jogador quanto espectador, o final do jogo não pode ser anticlimático.
- **A vitória tem que poder ser alcançada pelo jogador que está atrás.** Em muitos jogos de estratégia os jogadores tem que acumular poder de algum tipo, seja militar, seja financeiro, entre outros. O que este preceito diz é que o jogo não pode obrigar um jogador que está para trás no acúmulo de poder a vencer esta diferença para ganhar o jogo. Ou seja, deve haver sempre uma condição de vitória que dependa menos do poder acumulado do que a condição mais obvia.

Um jogo de estratégia que possuir estas características, segundo esta abordagem de (Jones, 2011), fará com que o jogador se envolva mais, facilitando sua entrada em *flow*.

Nota-se pela classificação dada a este gênero na figura 4.2 que ele não depende de narrativa nem de controles sofisticados.

Exemplos de jogos de sucesso são o StarCraft II(StarCraft, 2010)(figura 4.10), a série Civilization(Civilization, 2010)(figura 4.11) e o League of Legends(LoL, 2009)(figura 4.12). Todos eles podem ser jogados por vários participantes. StarCraft II foi o RTS que vendeu mais cópias em seu mês de lançamento: mais de três milhões. League of Legends é jogado exclusivamente online.

Figura 4.10: *(StarCraft, 2010) II. Batalha em tempo real entre artilharia e infantaria.*


Figura 4.11: *Civilization V(Civilization, 2010). Jogo em turnos com o mapa dividido em tabuleiro no formato de colméia.*

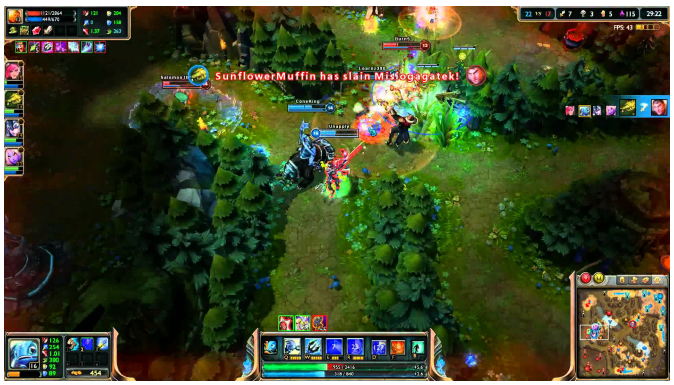

Figura 4.12: *League of Legends(LoL, 2009). Cada jogador controla um exército.*

### Puzzle

Puzzles são problemas cujo raciocínio para resolvê-los é divertido. Desde o famoso Cubo de Rubik e das complexas formas de madeira(figura 4.13) até suas versões nos videogames, qualquer cenário de entretenimento que apresente um problema com solução não evidente é um puzzle.

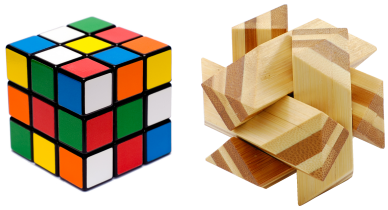

Figura 4.13: *Cubo de Rubik à esquerda. À direita, puzzle de madeira de seis peças*

Elementos deste gênero estão presentes em jogos de vários outros(figura 4.14), como ação e estratégia. Para o presente estudo, são considerados similares os fatores que definem o sucesso de um puzzle específico ou os de um jogo que se classifique inteiramente como puzzle. Este sucesso, no entanto, varia de acordo com a capacidade esperada do público alvo. Ao contrário dos jogos de outros gêneros, que tém sua dificuldade na realização de uma tarefa com solução visível, o puzzle tem sua dificuldade justamente no fato de sua solução ser desconhecida. Uma vez compreendido, o jogador vence aquele obstáculo sem muita dificuldade. Logo, o tempo esperado para cada puzzle é algo difícil de ser estimado por seu desenvolvedor, pois a capacidade de raciocínio dedutivo não é necessariamente homogênea em um grupo demográfico.


Figura 4.14: *The Legend of Zelda: Skyward Sword(Zelda, 2011). Mapa de um labirinto onde cada quadrado representa uma sala. O jogador deve reposicioná-las para ganhar acesso a elas.*

Em (Bates, 1997) são definidos tipos de puzzles, especificamente para jogos eletrônicos. São estes:

- **Uso de um objeto de uma maneira padrão.** Puzzle simples para apresentar objetos ao jogador. Usar uma lâmpada em um bocal para iluminar um quarto escuro, por exemplo.
- **Uso de um objeto de maneira atípica.** Fazer o jogador usar um objeto para algum propósito alem do padrão. Usar a cera que pinga de uma vela para vedar uma fresta, por exemplo.
- **Puzzles de construção.** Neste tipo, o jogador deve construir um objeto a partir de objetos mais simples para vencer um obstáculo.
- **Puzzles de informação.** Obstáculos que somente são vencidos se o jogador fornece uma informação que está faltando, como a senha de um computador, por exemplo.
- **Cadeia de eventos.** Este tipo exige que o jogador esteja muito familiarizado com as mecânicas do jogo. Ele consiste em uma série de outros puzzles mais simples que, quando resolvidos, desencadeiam eventos que solucionam o problema imposto.
- **Puzzles de personagens.** Focado mais em jogos que possuem foco em narrativa, este tipo tem como obstáculo um personagem. Ele é resolvido através da interação social. Seja dar um brinquedo a uma criança em troca de algo, seja subornar um guarda que bloqueia um caminho importante.
- **Puzzles que dependem de tempo.** A solução para este tipo depende de ações que não dão resultados imediatos quando realizadas. Ela exige do jogador a noção de uma ação feita em um momento impactará o estado do jogo algum tempo adiante. Acionar o freio de mão de um caminhão antes de carregar sua carroceria é um exemplo primitivo deste tipo.
- **Charadas.** Perguntas dúbias com respostas não evidentes.
- **Puzzles de diálogo.** Muito usado em RPGs, envolve escolher as falas certas dentro de um diálogo, de acordo com o personagem que serve de obstáculo.
- **Puzzles geográficos.** Situações em que o jogador precisa chegar em um lugar específico, mas a rota não é clara. Labirintos, sejam em duas ou três dimensões, são bons exemplos.

É importante ressaltar que um puzzle pode se enquadrar em mais de uma categoria, e mais, que um passo para sua solução pode ser outro puzzle.

Ainda em (Bates, 1997), o autor descreve falhas a serem evitadas pelo desenvolvedor: puzzles que necessitam que o jogador falhe a primeira vez para descobrir sua solução; eventos que acontecem sem uma causa clara e soluções que só fazem sentido para quem as desenvolveu.

Em contrapartida, as características de sucesso de um puzzle são: ter informações transparentes para o jogador; encaixar nos outros elementos de forma suave, no caso de jogos que não são exclusivamente do gênero; ter, em sua solução, elementos que se encaixem com o tema do jogo e provocar uma sensação de satisfação quando solucionados. Com estas preocupações em mente, é possível criar um puzzle que seja um desafio justo, inserindo e mantendo o usuário em *flow*, de acordo com os requisitos levantados na seção 4.1.

A importância dos aspectos motores varia de acordo com as escolhas do desenvolvedor mas, geralmente, os puzzles interessantes tem controles interessantes. Quando usado como elemento dentro de outro jogo, os controles de um puzzle têm que ser coerentes com os controles do resto do jogo.

O peso da narrativa também varia. Um puzzle isolado geralmente não necessita dela para ser bom.

Este é um gênero onde empresas e desenvolvedores independentes se destacam. Dois exemplos de extraordinário sucesso de público são Braid(Braid, 2009)(figura 4.15), onde o jogador volta e avança no tempo para passar dos obstáculos; e FEZ(FEZ, 2012)(figura 4.16), que se passa em um mundo tridimensional, mas em que o personagem se move nas projeções bidimensionais para progredir.

Figura 4.15: *(Braid, 2009). Os elementos marcados de vermelho são imunes à manipulação do tempo. Todos os outros retrocedem quando o personagem também retroceder. O jogador tem que usar este tipo de mecânica para sincronizar objetos e prosseguir.*

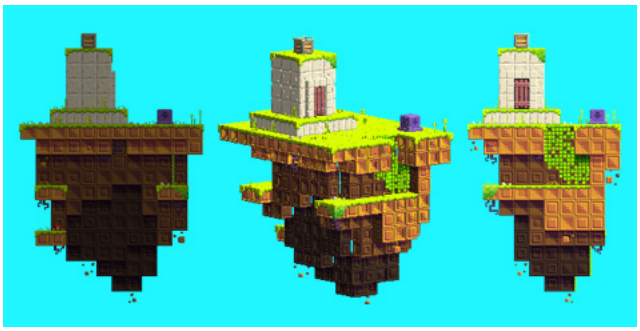

Figura 4.16: *(FEZ, 2012). O cenário é tridimensional mas o movimento é realizado em uma das projeções. Isto pode revelar segredos e aproximar objetos distantes.*

### Ação

Em um gênero bastante amplo, jogos de ação se dividem de acordo com a perspectiva de câmera. Jogos em primeira pessoa apresentam a perspectiva do protagonista. Jogos em terceira pessoa possuem uma câmera que exibe o corpo do protagonista e, muitas vezes, permite movimento independentemente dele.

Por englobar vários sub-gêneros, definir os aspectos que fazem um jogo de ação ter sucesso é uma tarefa mais subjetiva do que fazer isto para outros gêneros. Provavelmente por este motivo, existe consideravelmente menos literatura formal a respeito do que faz um jogo de ação ser bom.

O aspecto que é provavelmente o mais crucial é a tensão. Bons jogos de ação despertam em seu público a mesma sensação de bons filmes de ação: uma tensão constante, seja em um momento calmo demais, indicativo de iminência de conflito; seja no conflito em si.

Esta característica é atingida, predominantemente, por seus aspectos motores. O controle de um jogo de ação tem que se encaixar perfeitamente com a mecânica proposta, para passar esta tensão ao jogador. A narrativa também é de grande importância neste aspecto. A razão da classificação deste gênero na figura 4.2 ser uma onde a narrativa é menos importante que os controles se deu pelo fato de que existem bons jogos de ação sem destaque para narrativa, mas nunca sem controles bons.

A franquia Call of Duty(CallofDuty, 2003), que tem enorme sucesso de vendas, não é conhecida por ter narrativas especialmente envolventes, por exemplo. Já The Last of Us(LastofUs, 2013), jogo que foi sucesso de vendas e crítica em 2013, conta com controles considerados revolucionários devido a sua simplicidade e com uma narrativa impecável ao criar momentos de tensão e imersão(figura 4.17).

Figura 4.17: *The Last of Us(LastofUs, 2013). Tensão gerada pela dificuldade de sobrevivência em um ambiente hostil e dos laços entre personagens. Controles inovadores em um gênero já consagrado.*

A franquia Deus Ex(DeusEx, 2000) é um bom exemplo de um jogo de ação com elementos de RPG. A série sempre teve um foco considerável na narrativa e nas diferentes abordagens possíveis na realização de missões(Figura 4.18).

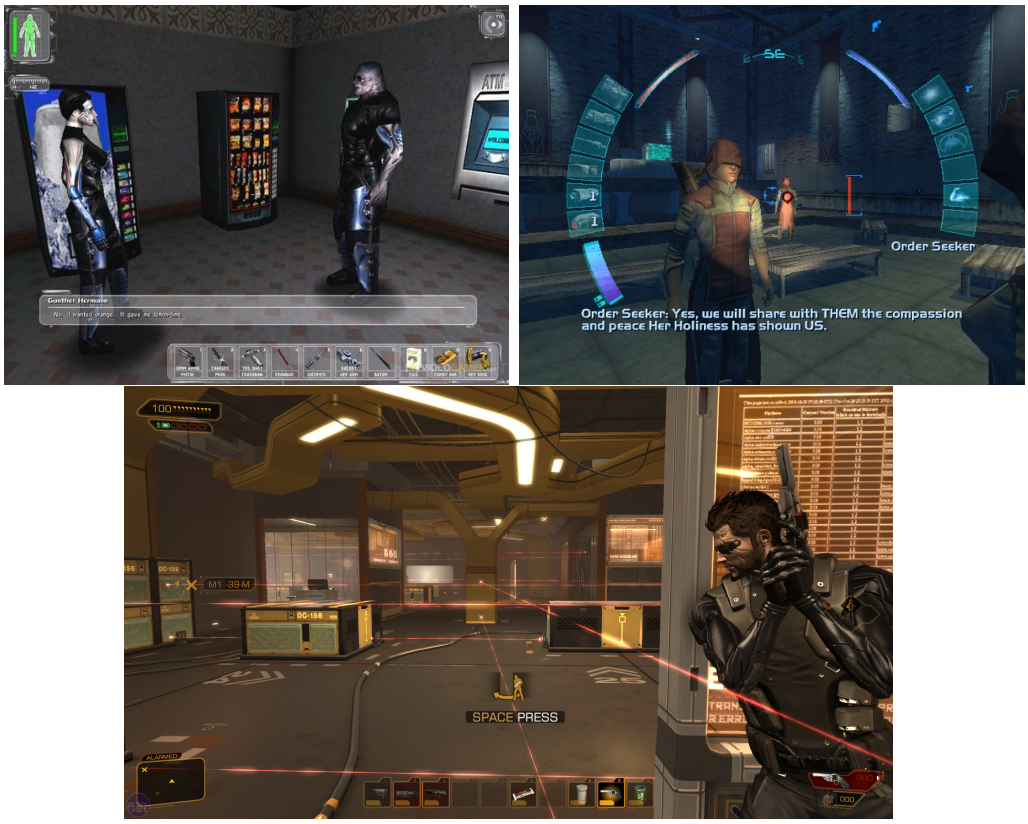


Figura 4.18: *Evolução dos três jogos da série (DeusEx, 2000). As habilidades do personagem são adquiridas ao longo do jogo de acordo com as abordagens preferidas do jogador, seja ele furtivo, sociável ou agressivo.*

### Jogos Motores

Quando se menciona jogos focados na experiência motora, se pensa em reflexos rápidos e objetivos desafiadores. Isto é verdade para alguns sub-gêneros. Jogos de música, como as franquias Guitar Hero(GuitarHero, 2005) e Rock Band(RockBand, 2007), onde o jogador simula um instrumento musical e tem que acertar as notas no ritmo certo em que elas aparecem na tela são exemplos disso (figura 4.19). Outro subgênero que possui uma legião de aficcionados são os jogos de nave, como o (Ikaruga, 2003) que, embora já antigo, representa bem seus semelhantes(figura 4.20). Os fãs deste tipo procuram um desafio especialmente difícil. Ambos os jogos de música e de nave possuem um foco muito pequeno em narrativa. Até pode existir uma história de ficção, mas o jogo pode ser aproveitado integralmente mesmo por quem a ignora.

Figura 4.19: *Tela de jogo do Guitar Hero. As notas correspondem aos botões de mesma cor, que devem ser apertados na ordem em que cruzam a linha inicial(GuitarHero, 2005).*


Figura 4.20: *(Ikaruga, 2003). O desafio é tão difícil que a nave que o jogador controla (parte inferior da tela) se camufla entre os projéteis dos quais ela deve desviar.*

Um jogo independente que fez bastante sucesso pela qualidade de seus controles e pelos níveis difíceis e que exigem extrema precisão é o Super Meat Boy(SuperMeatBoy). Este também é um exemplo em que a narrativa desempenha um papel com quase nenhuma relevância(figura 4.21).

Figura 4.21: *Super Meat Boy. O jogador controla um pedaço vermelho de carne em busca de sua namorada raptada por níveis complexos com pouquíssimo espaço para erros(SuperMeatBoy).*

Existem também outros tipos de jogos motores cujo prazer não advém do desafio, mas sim da fluidez dos controles. Este tipo de abordagem é mais recente e os melhores exemplos são (Flower) e (Journey), ambos desenvolvidos pelo estúdio independente (Thatgamecompany). O primeiro tem como objetivo polinizar um campo de flores e vê-las crescer. Para isso o jogador usa somente o acelerômetro para guiar uma esfera de pólen pelo ar(figura 4.22). Sem limites de tempo nem momentos de tensão, o jogo provoca imersão pela adequação entre os controles e os aspectos visuais.

Já o Journey conta um pouco mais com elementos de narrativa. Nele o jogador se vê controlando um ser humanóide de capa em um deserto rumo a uma montanha no horizonte(figura 4.23). Embora esse objetivo seja narrativo, a narrativa em si é muito sutil. Não há animações e os objetivos são explicados com movimentos de câmera. O jogo tem este nome pois enfatiza que o objetivo final é menos importante que a jornada, e foca em criar esta jornada com visuais impactantes e controles fluidos.

Figura 4.22: *(Flower). O jogo conta com controles fluidos e visual atraente.*

Figura 4.23: *(Journey). O jogador vai em direção à montanha ao fundo.*



Com exemplos diametralmente opostos, a impressão inicial de que este gênero tem como público alvo apenas aqueles que gostam de desafios intensos é contrariada. O sucesso deste tipo de jogo advém quase que exclusivamente da excelência e adequação dos controles aos outros elementos.

## 4.2 Bons Game Shows

Pouca literatura acadêmica existe sobre o que faz um Game Show de sucesso. Em 2008, houve um simpósio(Whannel et al., 2008) para discuti-los, mas nenhum documento oficial sobre seu conteúdo foi gerado. (Holmes, 2008) apresenta um estudo sobre o Quiz Show, que é um Game Show de perguntas e respostas. O traço identificado como mais relevante em seu sucesso de audiência

é a participação passiva dos telespectadores. Os temas e a dificuldade das perguntas devem ser compatíveis com o público alvo.

Ainda sobre o Quiz Show, (McQuail et al., 1972) argumenta sobre os quatro tipos de gratificação buscados pelo público. São eles:

- **Auto Referência.** O telespectador se compara com os participantes e se imagina participando, o que concorda com a opinião supracitada de (Holmes, 2008). Além disso, este aspecto se refere ao sentimento de vitória quando o participante para o qual se está torcendo ganha e, analogamente, a satisfação em ver o oponente perder.
- **Material para interação social.** Seja comentando sobre o episódio no dia seguinte, seja assistindo-o acompanhado de outras pessoas, um bom Quiz Show agrega pessoas socialmente.
- **Exitação.** Toda a tensão gerada por um jogo é compatível com um Game Show de qualquer tipo. Um final emocionante com vencedor indefinido ou a sensação de acertar uma pergunta, por exemplo. A incerteza é uma predominante causa de tensão. Esta característica é comum a qualquer cenário que induza um espectador a torcer ou ficar curioso com um resultado.
- **Apelo educacional ou informativo.** Seja aprendendo, confirmando conhecimento ou estimulando o estudo posterior sobre os temas das perguntas, Quiz Shows tem um forte apelo educacional.

A literatura acadêmica sobre o Reality Show também é escassa. (Millan, 2006) oferece uma dura crítica sob a ótica da psicologia a este formato, onde enfatiza que as principais características que fazem o telespectador ser capturado são o poder de decisão -no caso de Reality Shows participativos- e o voyerismo. Este último diz respeito à possibilidade do telespectador de julgar os participantes em situações cotidianas.

O pouco estudo a respeito destes tipos de programa leva a uma abordagem menos específica: analisar os Game Shows pela ótica de um programa de televisão genérico. Em (McQuail, 1987), o autor tenta responder por que as pessoas assistem televisão, e identifica quatro grandes motivos: informação; identidade pessoal; integração e interação social; e entretenimento. O primeiro e o último são auto-explicativos e evidentes. Já identidade pessoal e integração social necessitam de maiores explicações.

Identidade pessoal diz respeito a como um telespectador usa a televisão para se medir e se aprimorar. Ele busca reforço para seus valores pessoais e identificação com pessoas e modelos de comportamento que sejam compatíveis

com sua visão de mundo. Além disso, um telespectador pode ter novas perspectivas sobre traços que já possui, ao vê-los representados por uma outra pessoa.

Integração e interação social se referem a experiências que o telespectador tem que causam impacto na maneira como ele enxerga e participa da sociedadade a sua volta. A televisão pode trazer uma nova perspectiva sobre questões alheias a ele, criando uma proximidade, ou até mesmo empatia. Analogamente, pode revelar uma identificação dele com um grupo semelhante, fortalecendo-a. A televisão pode se apresentar até como substituto social, servindo de companhia.

A edição é uma ferramenta muito valiosa capaz de dar ênfase e criar momentos de tensão. Uma boa edição não altera só a forma do que vai ser apresentado, mas também a mensagem. No caso de Reality Shows, a edição cumpre um papel crucial. É importante ressaltar que esta ferramenta não é disponível para programas ao vivo, como é o caso de jogos do formato proposto.

As características levantadas nesta seção auxiliam a identificar um programa que tenha apelo de público. Isto não é sufuciente para que as pessoas o assistam. Existem cada vez mais canais de televisão disponíveis, e a concorrência pela audiência é acirrada. Um programa tem pouquíssimo tempo para despertar a curiosidade de um telespectador que esteja mudando de canal. Uma vez conquistado, o telespectador tem que ser mantido, ou ele desiste e volta a mudar. O programa tem que ser compreensível em pouco tempo. Se ele não se fizer entender rápido, vai perder audiência. Este ponto é de extrema importância para Game Shows do formato proposto. Um desenvolvedor só criará um jogo de sucesso se este permitir rápido entendimento.

# 5
# Um Framework para Game Shows Interativos

O MarkerFinder(capítulo 3), juntamente com o os exemplos descritos no capítulo 2, indicaram uma possível área relevante e pouco explorada de pesquisa: a convergência entre videogames e televisão. Existe, no momento, tecnologia avançada o suficiente para integrar programas de TV ao vivo à Realidade Aumentada com qualidade. Dentro de um estúdio, uma ferramenta como o (LightCraft) não é somente capaz de compor o plano de fundo de um cenário. Uma vez que o espaço de um estúdio esteja mapeado, é possível adicionar elementos virtuais a ele. Como a posição e orientação da câmera são conhecidas, exibir tais elementos como parte da imagem final é relativamente simples.

O elemento que falta para unir o videogame à televisão, uma vez que a Realidade Aumentada de qualidade é possível, é o jogador. Para que ele seja inserido, o uso da segunda tela no lugar do controle é uma abordagem promissora. Um game show com elementos de Realidade Aumentada e possibilidade de participação dos telespectadores com disposivos móveis é um estudo de caso bastante adequado para a pesquisa proposta.

Embora seja o foco desta dissertação, o framework é só uma ferramenta para o estudo da convergência. É necessário estudar o apelo de público que este tipo de jogo terá. Por esta razão, a seção 5.1 prevê os possíveis desafios que um jogo convergente terá para seu sucesso.

O framework, em linhas gerais, é descrito em 5.2. Sua especificação de classes está em 5.3.

O capítulo 6 detalha o estado atual de sua implementação.

## 5.1
## Potenciais desafios à Convergência

O estudo apresentado no capítulo 4 ajuda a entender o que faz um jogo ou um programa de sucesso. Esta dissertação estuda o potencial de uma mídia que é as duas coisas. Por esta razão, existem diversas possíveis decisões de desenvolvimento que podem levar um Game Show Interativo a ser um jogo melhor mas um programa de TV pior e vice-versa. As duas óticas não podem

ser aplicadas isoladamente sob o risco de perda considerável de apelo. Esta seção descreve os cenários onde se esperam encontrar desafios advindos da convergência. É importante ressaltar que os desafios descritos podem não ser impedimentos, mas somente obstáculos. Cabe ao desenvolvedor do jogo a engenhosidade de superá-los. Também é relevante apontar que estes potenciais desafios são levantados sob as óticas dos videogames e game shows atuais. É possível que os desafios do formato proposto sejam de outra ordem.

O primeiro desafio é relativo ao papel do participante. Quanto mais poder de decisão ele tiver, menos poder de decisão o telespectador terá. Uma das soluções pensadas para isso é que o participante fosse um apresentador ou um ator contratado, o que eliminaria a necessidade de diverti-lo. Isso permite aumentar a interatividade e o poder do telespectador que joga. Além disto, por estar em estúdio, o participante não verá os objetos virtuais de forma tão realista quanto os telespectadores. Isto tem que ser levado em conta pelo desenvolvedor enquanto este formula o jogo.

Outro potencial problema é o quanto de poder um telespectador que joga tem. Nem todo mundo que assiste um Game Show Interativo vai estar participando dele como jogador. Sendo assim, se muito do jogo depender do que acontece nos tablets e smartphones dos jogadores, aqueles que somente assistem podem não entender o que se passa. Uma boa prática no desenvolvimento de um jogo para evitar este problema é minimizar a quantidade de informação exclusiva para dispositivos móveis, exibindo o máximo possível na televisão. Existe uma demanda relativamente nova por parte de pessoas que gostam de assistir a outras jogando. Videos de *let's play*, onde um jogador de videogame captura sua experiência e a publica online ou faz *streaming* têm um público cada vez maior. O (TwitchTV), adquirido pela (Amazon) é um site com somente este tipo de vídeo. Esta tendência mostra que não é necessariamente um problema para um telespectador assistir a outras pessoas jogando, mas é um problema se ele não possuir informações suficientes para entender o que estiver acontecendo.

Outra dualidade que gera dificuldades é o número de telespectadores que jogam em oposição ao poder que cada um destes jogadores tem de influenciar o jogo. Quanto mais telespectadores participarem, menos impacto cada um deles terá no jogo. Um número alto de jogadores exige, quase que invariavelmente, que haja algum tipo de votação para decidir quais interações irão ocorrer. Analogamente, um número menor de jogadores possibilita uma interatividade individual maior, mas gera um problema em relação ao método de escolha deles. Escolher a proporção ideal entre telespectadores ativos e passivos é um problema que, ao que tudo indica, deverá ser resolvido empiricamente,

caso a caso. De qualquer maneira, há pesquisa sobre formas mais sofisticadas de interferência da audiência em programas de TV interativa(Baffa e Feijó, 2015). Estas novas formas de participação da audiência podem definir grupos com interesses comuns -identificados por redes sociais- que estabelecem novos mecanismos de influência e de maximização de resultados. Até mesmo mecanismos de leilão podem ser explorados. O projeto do framework leva em conta a possibilidade desses tipos de expansão e permite futuras adaptações minimizando o retrabalho.

Ainda se tratando do fato de que muito provavelmente haverá menos vagas para jogadores domésticos do que telespectadores que desejam jogar, é imprescindível que fique bastante claro se uma pessoa não consegue se conectar por falta de vagas ou por alguma falha de comunicação. A falta de transparência acerca deste tipo de informação pode gerar frustração e perda de interesse.

O fato do framework poder ser usado somente para desenvolver jogos em turnos -restrição melhor elaborada na seção 5.2-, já elimina alguns gêneros descritos no capítulo 4 do universo de possibilidades. A maioria dos formatos convencionais para jogos de ação e jogos de estratégia ou simulação em tempo real já são conceitualmente descartados.

Já a limitação de tempo de exibição na televisão limita, a princípio, a complexidade do jogo. Um RPG é inviável, a menos que seja muito bem desenvolvido e adaptado. O trabalho necessário para criar uma campanha de RPG que se enquadre nos moldes da televisão adicionaria muitas variáveis ao estudo e não é adequada para uma abordagem inicial. O principal problema, além do tempo e da necessidade de uma narrativa complexa pela exigência do próprio gênero, diz respeito à dificuldade de capturar um telespectador que esteja mudando de canal. RPGs são jogos de mecânica complexa e exigem muito tempo para serem compreendidos.

Quanto à simulação, jogos de administração são especialmente difíceis. Por não possuirem uma progressão clara, ou mais importante, um final evidente, eles não cabem em um programa de TV. O Game Show Interativo não pode ser interrompido a qualquer momento, pois a continuidade dele depende de um novo episódio. Já simulações de atividade cabem melhor no formato.

Um jogo de estratégia em turnos(TBS) se adequa bem ao formato proposto. Ele permite muitos participantes no estúdio com telespectadores ajudando-os, por exemplo.

Puzzle é um gênero mais complexo. Por depender de capacidade de raciocínio dedutivo dos jogadores e por este não ser um atributo de distribuição demográfica simples, estabelecer o grau de desafio ideal para os problemas

é um obstáculo. Outro aspecto que pode ser motivo de frustração é o caso de um telespectador entender a solução para um problema, mas seu poder de participação via dispositivo móvel ser limitado demais para que ele possa influenciar tanto quanto gostaria.

Jogos motores, a princípio, não se enquandram muito bem em um jogo de turnos. A espera entre um turno e outro pode interromper o estado de *Flow*. Elementos que exijam coordenação motora mais complexa ou sincronizada, no entanto, podem ser incorporadas. A presença de desafios motores dentro de um turno, tanto a participantes quanto a telespectadores, adiciona tensão e, consequentemente, apelo ao jogo.

## 5.2 Framework em Linhas Gerais

O framework proposto tem como propósito possibilitar o desenvolvimento de Game Shows Interativos. Ele foi modelado para que alguém que queira desenvolver um jogo deste tipo possa fazê-lo de forma prática. É possível dividir o jogo em times e decidir o quanto de participação cada jogador terá, esteja ele em casa ou no estúdio. Um jogador com dispositivo móvel pode participar do jogo através desde votações até uma interferência mais impactante. O framework foi concebido para deixar todas as decisões deste tipo a cargo do desenvolvedor.

A ferramenta se divide em três grandes módulos(figura 5.1). O principal deles, chamado de Core ou módulo central, é responsável por receber os inputs de todos os jogadores -domésticos ou de estúdio- e processá-los de acordo com as regras definidas pelo desenvolvedor. É a máquina de estados do jogo, que calcula um novo estado de jogo a cada turno e o envia a todos os participantes. Pode-se enxergar o Core como o *backend* do framework.

O módulo Studio é responsavel por tudo que tange o estúdio de TV onde o jogo é exibido. Ele recebe o estado de jogo enviado pelo módulo Core e atualiza os elementos de Realidade Aumentada e a situação dos participantes. Ele também coleta os inputs destes participantes e os envia ao Core.

O módulo Mobile recebe o estado de jogo do Core e atualiza o que é exibido na tela dos dispositivos móveis dos jogadores domésticos, bem como coleta os inputs destes jogadores e os envia ao módulo central.

Quanto à terminologia utilizada ao longo deste documento, denomina-se jogador qualquer pessoa que jogue o jogo, seja em casa ou no estúdio. Participante é o jogador no estúdio. Telespectador, a menos que explicitamente descrito de forma diferente, diz respeito ao jogador que participa de casa com a segunda tela. Estado de jogo é o conjunto de informações necessárias e

suficientes para definir um momento do jogo. Desenvolvedor é a pessoa que usa o framework proposto para desenvolver Game Shows Interativos.

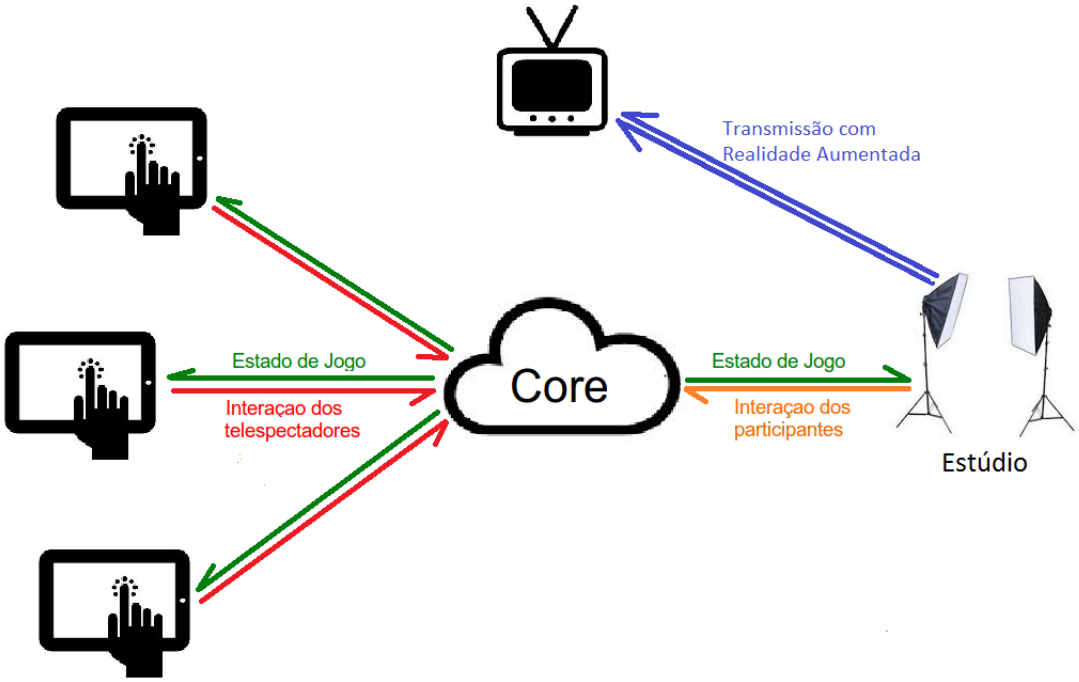


Figura 5.1: *Arquitetura de comunicação entre os módulos.*

O grande gargalo de tempo é a comunicação entre todos os jogadores. Por esta razão, o módulo central tem que transmitir o estado do jogo da forma mais resumida possível. Todos os objetos podem possuir representações gráficas diferentes quando exibidos na TV ou na segunda tela. Estes modelos devem ser transmitidos a seus respectivos módulos antes do início, para que a comunicação durante sua execução seja leve. A cada novo estado de jogo, o Core transmite somente a diferença entre este e o anterior, pois isto diminui consideravelmente o tamanho do pacote enviado. Além disso, por se tratar de um ambiente assíncrono, os módulos de estúdio e de segunda tela podem solicitar o estado atual inteiro ao Core, no caso de atrasos ou falhas de comunicação.

Ainda por questões de comunicação -que incluem *delay* na transmissão-, em um primeiro momento, a concepção do framework não permite jogos em tempo real, mas somente em turnos. Devido a um número potencialmente alto de jogadores que podem estar espalhados geograficamente, lidar com as interações de cada um deles em tempo real seria um desafio técnico elevado que foge ao escopo de um teste inicial. Em tempo real, uma demora de três segundos no processamento das interações, por exemplo, é crítica e quebra o

fluxo do jogo. A mesma demora para um turno tem um impacto bem menor, possivelmente até imperceptível.

O Core é o módulo mais importante e complexo do framework, devido às suas funções centralizadoras. Na presente dissertação, apenas ele foi implementado, em (Python). Cada um dos módulos tem uma linguagem de implementação que é mais adequada. O módulo de estúdio, por tratar de computação gráfica em tempo real, deve ser implementado em uma linguagem que consuma menos recursos computacionais, provavelmente C++ com a biblioteca de Realidade Aumentada do (OpenCV). O módulo mobile deve ser implementado, possivelmente, em mais de uma linguagem, pois os principais sistemas operacionais dos dispositivos móveis têm restrições quanto às linguagens de programação que suportam. Por esta diversidade de linguagens, a transmissão de informações está sendo feita com (JSON), notação que atende bem às questões de um ambiente multi-linguagem. Como um trabalho futuro, deverá ser analisada a possibilidade de compactação deste pacote JSON, porque isto pode diminuir ainda mais o overhead de rede.

Quanto aos métodos de sincronia de segunda tela, provavelmente não será usado nenhum dos descritos na seção 2.3. Estes são baseados em reconhecimento de sinais de áudio do programa exibido na televisão. Isto é feito, muitas vezes, porque o produtor da aplicação de segunda tela não tem relação com a produção do conteúdo em si. O Game Show Interativo é um formato que considera a segunda tela desde fases iniciais de seu desenvolvimento. Além disso, já utiliza um servidor para transmitir dados aos telespectadores e receber suas respostas. Por estas razões, a sincronia de conteúdo provavelmente será realizada independentemente do áudio do programa de TV.



## 5.3 Modelagem de Classes

As classes identificadas para atender a proposta do framework estão no diagrama de classes(figura 5.2) e descritas abaixo. Para poupar espaço, os argumentos dos métodos foram omitidos do diagrama. O anexo A.1 contém uma descrição em detalhes de cada atributo, método e argumento.

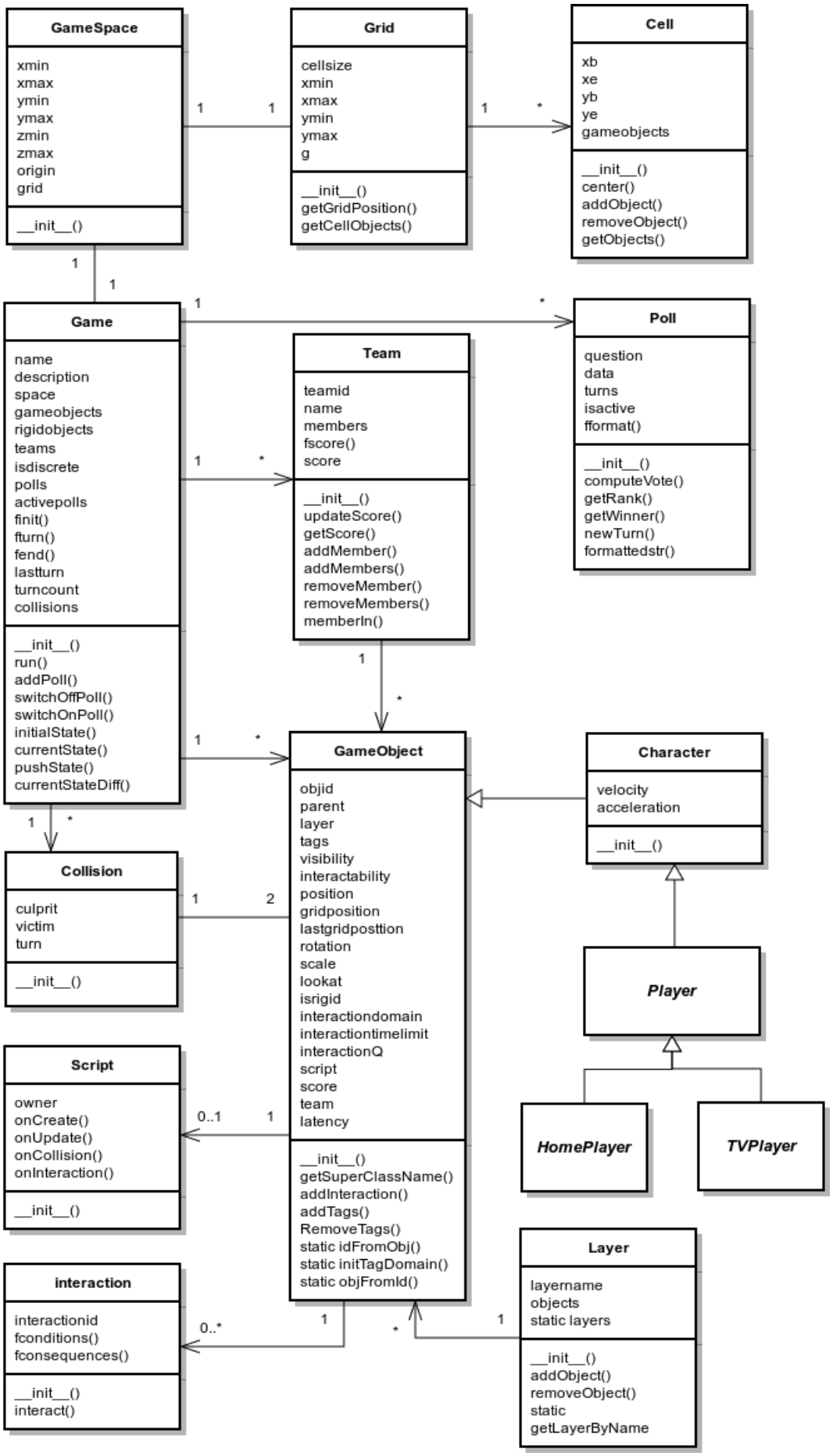


Figura 5.2: *Diagrama de Classes do Framework.*

– **GameSpace**
Define o espaço tridimensional do jogo. Via de regra é o espaço mapeado do estúdio.

– **Grid**
Divide o chão do GameSpace -plano onde a coordenada Z é zero- em um Grid. Dependendo do jogo desenvolvido, pode-se poupar tempo de processamento considerando o chão como um tabuleiro em vez de um espaço contínuo.

– **Cell**
É uma célula, ou casa, do Grid. Ela possui uma lista com todos os objetos -classe GameObject- que estejam localizados nela.

– **GameObject**
Representa todos os objetos dentro de um jogo, desde corpos inanimados a personagens e jogadores. Classe para ser herdada por outras mais específicas. É uma das classes principais do framework. Um GameObject pode ser customizado para só ser visto ou interativo a conjuntos específicos de jogadores. Isto possibilita que o desenvolvedor crie objetos que podem sofrer interações originadas por um só time, por exemplo. Ou ainda, pode criar objetos que não sejam exibidos na TV ou no dispositivo móvel.

– **Character(GameObject)**
Um GameObject que pode se mover pelo espaço e ter um comportamento. Uma especificação de GameObject que exclui objetos inanimados.

– **Player(Character)**
Um Jogador. No momento, ou jogador de estúdio ou doméstico. A classe player existe somente como classe intermediária entre Character e os tipos mais específicos de jogadores.

– **HomePlayer(Player)**
Jogador doméstico: o telespectador que joga com o dispositivo móvel. No momento esta classe ainda não foi modelada em detalhes por depender de escolhas e limitações tecnológicas que ainda não foram levantadas

– **TVPlayer(Player)**
Jogador de estúdio, participante. Aquele jogador que está no palco. Pelos mesmos motivos da classe anterior, esta também ainda não teve seus detalhes modelados.

– **Layer**
Um conjunto de GameObjects. Agrupar objetos em layers pode melhorar o tempo de processamento sempre que se tem que iterar por eles. O desenvolvedor utiliza objetos desta classe como desejar.

– **Collision**
Representa uma colisão entre dois GameObjects. Contém informações sobre os dois objetos e o número do turno em que a colisão ocorreu. É possível atribuir a cada objeto o papel de culpado ou vítima. No caso de um jogo de xadrez, por exemplo, a peça capturada pode ser considerada vítima daquela que a capturou.

– **Script**
Um script para ditar o comportamento de um GameObject. O desenvolvedor escreve o script e o anexa a um objeto. Quase todos os atributos de um Script são funções que serão chamadas em momentos específicos do jogo. O desenvolvedor fica livre para modelar as funções da maneira mais conveniente. A primeira destas funções -onCreate- é chamada no momento da criação do GameObject. onUpdate é chamada uma vez por turno. onCollision e onInteraction são chamadas quando o objeto participa de uma colisão ou sofre uma interação, respectivamente.

– **Interaction**
Trata a interação entre jogadores(Player) e outros GameObjects. Basicamente um conjunto de condições e consequências. Quando o objeto sofre um tipo de interação específica, ou seja, quando as condições são verdadeiras, as consequências são executadas. Existe um conjunto com todas as interações possíveis do jogo. Cada GameObject possui um subconjunto -interactiondomain- daquelas interações que são aplicáveis a ele.

– **Team**
Um time de GameObjects. Inicialmente, membros de um time deveriam ser da classe Player. Essa restrição foi abolida para dar mais liberdade ao

desenvolvedor. É possível customizar a função que calcula a pontuação do time. Por padrão, esta pontuação é a soma das pontuações dos membros.

– **Poll**

Uma votação, ou enquete. Pergunta múltipla escolha dirigida aos jogadores. Como se trata de um programa de TV, uma forma de interação possível é fazer perguntas aos telespectadores e usar as respostas para influenciar no jogo. Esta é uma maneira de prover interatividade mesmo a quem não participa via dispositivo móvel.

– **Game**

Classe que contém todos os atributos e funções necessárias para rodar o jogo, ou seja, estabelecer um estado e suas transições. É o jogo propriamente dito. Centraliza e administra todas as informações de todos os elementos. É responsável por receber as interações dos jogadores e as transmitir o estado do jogo.

# 6
# Implementação do Framework Proposto

Dos três módulos propostos(figura 5.1), somente o Core está implementado. Ele foi escrito em (Python) e comentado seguindo os padrões do (Doxygen) para geração automática de documentação em html e (LaTeX). Um jogo de damas foi implementado como teste deste módulo e está detalhado em 6.1.

Sua implementação foi dividida em onze arquivos, cujo código se encontra em `icad.puc-rio.br/~gameshow/frameworkGSI/coreSources/` e sua documentação em `icad.puc-rio.br/~gameshow/frameworkGSI/htmlDocumentation/`. Via de regra, cada arquivo .py contém somente uma classe: aquela com o mesmo nome do arquivo. **Team.py** contém a classe **Team**, por exemplo. **Game.py** contém as classes **Game** e **Poll**; **GameObject.py** contém **GameObject** e **Layer**; e **Space.py** contém **Cell**, **Grid** e **GameSpace**.

Neste capítulo, os atributos de objetos estão escritos em azul e os métodos e seus argumentos em vermelho. Tudo o que for estático, seja atributo ou método, estará em verde.

Como este é o módulo que centraliza a comunicação entre todas as partes envolvidas no jogo, cabe a ele receber todos os inputs de todos os usuários, estejam eles em seus dispositivos móveis ou no estúdio. Uma vez processados todos estes inputs, um novo estado de jogo será gerado e tem que ser enviado de volta para os mesmos participantes. Como cada módulo é executado em ambientes diferentes, cada um com suas metas e limitações, é muito provável que suas implementações sejam em linguagens de programação diferentes. Por esta razão, esta comunicação entre o Core e os outros módulos tem que ser em uma linguágem genérica e largamente conhecida. Optou-se pelo (JSON), que já conta com bibliotecas de escrita e leitura para todas as grandes linguagens e também é compreensível para seres humanos.

Em Python, para se transcrever as informações de uma classe para JSON é necessário usar um dicionário, que é uma lista imutável em que cada elemento é um par composto por uma chave e um valor. Neste caso, a chave é uma *string* com o nome do atributo, seguida do valor daquele atributo.

Para cada classe cujas informações precisam ser transmitidas ao estúdio e dispositivos móveis foi declarado um par de funções que montam JSONs com tais informações. A função **introduction()** monta e retorna um JSON inicial, com os valores de todos os atributos relevantes. Ela apresenta o objeto para quem ainda não sabe de sua existência. Já a função **catchingUp()** é chamada quando já se tem uma referência ao objeto, e deseja-se atualizar suas informações. Isto foi elaborado com o objetivo de minimizar o tamanho do JSON a ser enviado, o que influencia na banda consumida.

As classes **Poll**, **GameObject**, **Character**, **Collision**, **Layer**, **Interaction** e **Team** possuem esse par de funções como métodos de seus objetos.

A classe **Script** não possui nenhum atributo que precise ser enviado para fora do módulo central.

**HomePlayer** e **TVPlayer** não foram implementadas até o momento, pois sua implementação depende dos dois outros módulos.

**Game** é um caso especial. Por centralizar todos os elementos do jogo, é ela que monta o JSON que contém seu estado completo. Foram implementados quatro métodos:

- **initialState()**, que retorna o JSON do estado inicial do jogo;
- **currentState()**, que retorna o JSON do estado atual completo do jogo;
- **pushState()**, que insere o estado atual no buffer de estados do jogo (**statebuff**).
- **currentStateDiff()**, que retorna um dicionário com a diferença entre dois estados em **statebuff**. Conhecendo um estado antigo e a diferença entre ele e um estado novo, é possível deduzir o novo. Este dicionário é potencialmente muito menor do que um que contenha um estado inteiro, e diminui o overhead de comunicação.

As funções que transmitirão essas informações aos outros módulos não foram implementadas. Como as tecnologias a serem usadas para eles ainda não estão totalmente definidas, a comunicação entre eles também não está.

Outros detalhes de como cada classe foi implementada, quando relevantes, estão descritos abaixo:

- **GameSpace**

  No método construtor **__init__()**, se omitido o argumento **origin** ele tera o ponto (0,0,0) como valor padrão. Já o argumento **cellsize** tem o valor zero como padrão. Este valor indica uma célula de tamanho zero, o que é interpretado como inexistencia de um Grid.

– **GameObject**

A lista estática **tagdomain** foi implementada como um dicionário.

Fez-se necessária a criação de uma variável estática **game** para referenciar o jogo (objeto da classe **Game**).

No método construtor **__init__()**, vários argumentos podem ser omitidos em favor de seus valores padrão. **parent**, **layer**, **script** e **team** têm None como padrão. **score** e **latency** recebem zero. **interactiontimelimit** recebe zero, o que indica um tempo limite indeterminado para interação. **isrigid** recebe False.

O atributo **gridposition** foi implementado como uma função sem argumentos. O GameObject possui uma referência para o objeto da classe Space que contém o Grid. Através desta referência é calculada a célula. Já **lastgridposition**, por sua vez, é um vetor com a posição anterior. Ele é o valor do retorno de **gridposition** para a posição anterior.

O atributo **interactionQ** foi implementado como uma fila de tuplas no formato **(interacion, interactor, lcond, lcons)**. **interaction** é uma referência ao objeto que representa a interação; **lcond** à lista de parâmetros a serem passados para **interaction.fconditions** e **lcons** à lista a ser passada para **interaction.fconsequences**. **interactor** é o autor da interação.

– **Script**

No método construtor **__init__()**, os argumentos **onCreate** e **onUpdate** tem como valor padrão *lambda x: None*, que representa uma função que recebe um parâmetro e não faz nada. Similarmente, **onCollision** e **onInteraction** recebem por padrão *lambda x, y: None*; uma função que recebe dois parâmetros e não faz nada. Por esses quatro argumentos serem funções programadas livremente pelo desenvolvedor e pelo Script ser o que governa o comportamento de um GameObject, é recomendavel que elas tenham como argumento o GameObject ao qual o Script está atrelado (**owner**). No caso de **onCollision** e **onInteraction**, recomenda-se que, alem de **owner**, o outro argumento seja o GameObject contra o qual se colidiu ou com o qual se interagiu, respectivamente.

– **Interaction**

Os atributos **fconditions** e **fconsequeces** são funções sem restrição alguma por parte do Framework. Suas implementações ficam totalmente a cargo do usuário.

O tipo da variável estática **possibletypes** pode ser ou um dicionário ou uma lista, ficando a cargo do desenvolvedor.

– **Team**

No método construtor **__init__()**, o argumento **fscore**, se omitido, assume como valor padrão *lambda members: sum( [member.score for member in members] )*, que é uma função que simplesmente soma **member.score** de cada GameObject membro do time e atribui o resultado ao **score** do time. Foi escolhido este padrão por ser muito simples e atender grande parte dos jogos a serem desenvolvidos. O desenvolvedor é livre para usar a função de cálculo de pontuação que melhor lhe convier.

O método **addMembers()** recebe uma lista de GameObjects a serem incluídos no time. Nenhum erro é disparado se um elemento dessa lista ja estiver no time em questão. Se esse for o caso, este elemento simplesmente não é inserido e o método continua tentando inserir os outros elementos da lista. Tambem não há nenhuma restrição que impeça um GameObject de estar em mais de um time. A identificação da necessidade desta limitação e sua implementação ficam a cargo do desenvolvedor.

O método **removeMembers()** tenta remover todos os GameObjects da lista que recebe como argumento do time. Nenhum erro ocorre se um desses objetos não for membro do time, ele apenas é ignorado.

– **Poll**

No método construtor **__init__()**, o valor padrão para **turns** é -1, o que indica validade indefinida. **isactive** por padrão é True e **fformat** é *lambda x: None*, uma função que recebe um argumento e não faz nada.

No método **computeVote()**, o argumento **votes** tem como valor padrão 1. Isso quer dizer que, se omitido o argumento, será contado um voto para **answer**.

Analogamente, o argumento **turnnumber** tem como padrão o mesmo valor. Se omitido, somente um turno é computado.

– **Game**

Por serem funções de livre implementação pelo desenvolvedor, recomenda-se que **finit**, **fturn** e **fend** recebam um argumento: o próprio objeto da classe Game ao qual elas pertencem. Desta maneira elas tem acesso aos atributos e métodos do objeto e conseguem alterá-los.

## 6.1 Testes

Os pontos mais importantes de se testar em qualquer framework são a facilidade que um desenvolvedor tem para usá-lo e a consistência do resultado de acordo com seu escopo. Para o projeto em questão -mais especificamente para o módulo Core- o primeiro ponto se traduz na facilidade e tamanho do código gerado no desenvolvimento do arcabouço de um jogo. Já o segundo diz respeito à capacidade da plataforma de manter a consistência do jogo. Um outro tipo importante de teste, que não foi realizado nesta dissertação, é do uso do framework na criação de vários jogos, para comprovar sua generalidade e robustez.

Levando isso em consideração, o teste escolhido foi o desenvolvimento de um jogo de Damas, que é uma espécie de caso mínimo em que estes aspectos são evidentes.

Como os outros dois módulos não foram implementados, a transmissão de interações de usuários foi abstraída. O jogo lê uma fila de comandos e executa o primeiro válido. Isto simula uma abordagem possível na resolução de interações conflituosas num ambiente assíncrono e multiusuário. Analogamente, a transmissão do estado atual do jogo para os usuários é simulada com a inserção, movimento a movimento, do estado atual em uma fila.

O jogo de damas implementado segue as regras inglesas. Doze peças brancas e doze pretas, cada cor em um lado de um tabuleiro padrão 8x8 usando somente as casas pretas. As peças só se movem uma casa em uma das diagonais para a frente, ou seja, na direção do time oposto. Quando uma peça cruza o tabuleiro inteiro ela é promovida a um rei e, a partir deste ponto, ela pode andar uma casa em qualquer diagonal, seja para frente ou para trás. Uma peça captura uma adversária se esta estiver em uma casa na qual a primeira conseguiria normalmente se mover, e se a próxima casa nesta mesma direção estiver vaga. Uma peça que acabou de realizar uma captura deve continuar capturando, no mesmo turno, até que não existam mais possibilidades. Se uma captura é possível, o jogador é obrigado a realizá-la. Se mais de uma for possível, cabe ao jogador escolher qual realizará. O jogador com as peças pretas começa. O primeiro time que não tiver nenhum movimento possível na sua vez perde o jogo.

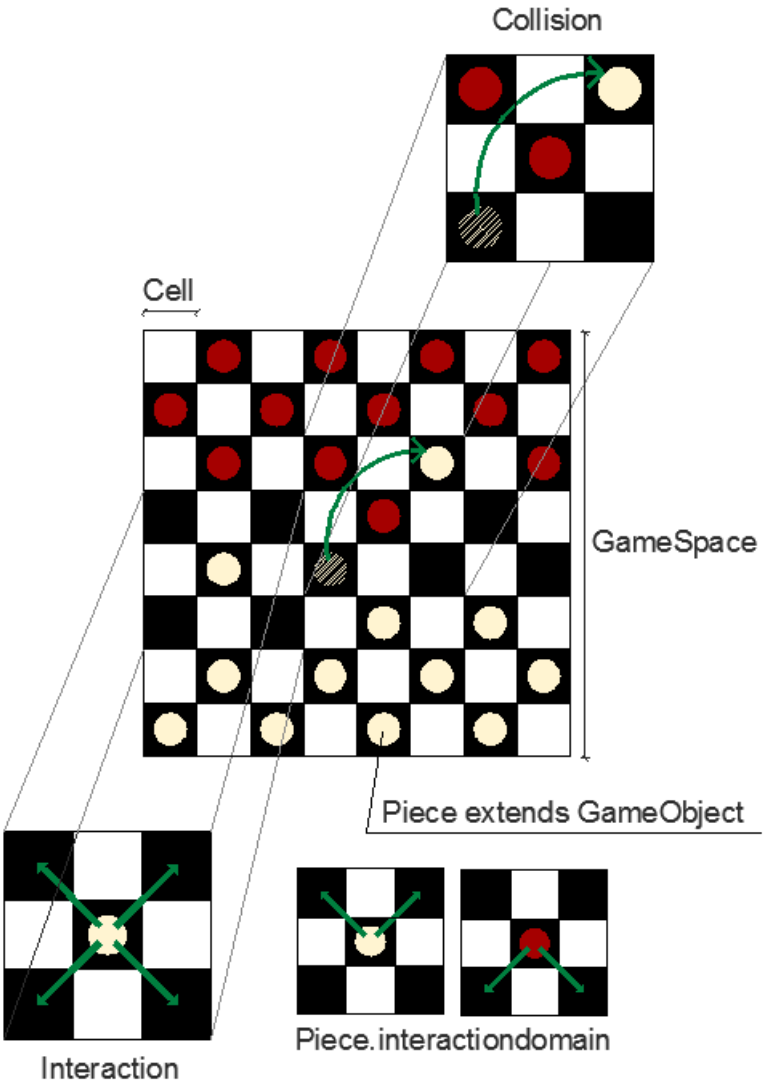


Figura 6.1: *Jogo de damas implementado de acordo com o framework.*

A implementação, em python, do Core deste jogo usando o framework foi dividida em quatro módulos: **Piece.py**, que implementa a peça como extensão da classe **GameObject**; **piecebehavior.py**, que contém o **Script** do comportamento da peça; **pieceinteractions.py**, que contém as possíveis interações dos jogadores com as peças; e **Checkers.py**, que contém a representação do jogo como uma classe que herda da classe **Game**(figura 6.1). Os detalhes de cada um destes módulos seguem abaixo:

– **Piece.py**

A classe **Piece** é basicamente um **GameObject**. A única diferença é o método **promotion()**, que é chamado quando a peça chega ao lado oposto do tabuleiro e é promovida a um rei. Este método inclui a capacidade de se mover para trás em seu dominínio de interações (**GameObject.interactiondomain**). A lista estática **pieces** é so

uma referência à lista **GameObject.gameobjects**, para facilitar o desenvolvimento.

– **piecebehavior.py**
O comportamento de uma peça é definido por um membro da classe **Script**.
A função *onCreate()* inicializa a peça, centralizando-a na casa do tabuleiro em que ela estiver.
*onCollision()* trata as colisões. Neste caso, é considerada que a peça sofreu uma colisão quando ela é capturada. Logo está função a remove do tabuleiro.
*onUpdate()* é chamada uma vez por turno pra cada peça. Executa a primeira interação válida da fila de interações da peça(**interactionQ**). Em seguida, confere se a peça mudou de posição faz as alterações necessárias no tabuleiro. Por fim, promove a peça a um rei se a nova posição exigir isto.
*onInteraction()* recebe uma tentativa de interação e a insere em **interactionQ**, se esta interação for permitida para a peça em questão; ou seja, se ela estiver presente na lista **interactiondomain**.

– **pieceinteractions.py**
Neste módulo foram definidas funções para a captura e movimentação de peças.
A função *capture()* é responsável por atualizar a pontuação, que se traduz no número de peças capturadas, e por chamar a função *onCollision()* da peça capturada.
*isMoveClear(piece, direction)* tem uma peça e uma direção como argumentos. Se o movimento da peça na direção for possível, de acordo com as regras, a função retorna True. Caso contrário, False.
*move(piece, direction)* move a peça *piece* na direção *direction*. Se o movimento for uma captura, ela chama *capture()* passando a peça que se moveu e a que foi capturada, nesta ordem, como parâmetros.
A partir destas duas últimas funções, foram criadas quatro interações(objetos da classe Interaction), uma para cada direção possível. Isto foi feito usando funções lambda, como mostra o trecho de código a seguir.

```
cond = lambda direction: lambda piece:
                         isMoveClear(piece, direction)
cons = lambda direction: lambda piece:
                         move(piece, direction)

leftup = Interaction(  1, cond( (LEFT, UP) ),
           cons( (LEFT, UP) )  )
```

– **Checkers.py**

Este é o módulo que importa todos os outros três e roda o jogo.

*broadCastState()* é a função que simula a transmissão do estado atual do jogo para os usuários. Ela insere tanto o estado atual do jogo em uma lista de estados, como insere a diferença entre os dois últimos estados em uma outra lista. Esta diferença é obtida através da função **currentStateDiff()** da class *Game*. Parte deste teste envolve comprovar que a diferença entre dois estados é suficiente para deduzir o estado atual, desde que se tenha o anterior.

*finit()* é a função responsável por inicializar o jogo. Ela monta o tabuleiro e chama a função *onCreate* de cada uma das peças.

*fturn()* é chamada no início de cada turno. Um turno, via de regra, consiste em um movimento. Quando um time move uma peça e passa a vez ao próximo, inicia-se um novo turno. A exceção a esta regra é o caso de múltiplas capturas pela mesma peça, que ocorrem todas dentro de um mesmo turno. Por esta razão, é importante ressaltar que uma mudança de turno pode equivaler a mais de uma mudança de estado. Os estados contém o número do turno, que é o atributo **turncount** da classe *Game*. Quando ocorrem múltiplas capturas, a fila de estados conterá multiplos estados com o mesmo número de turno. *fturn()* é responsável por receber cada interação e executar um movimento assim que houver uma interação válida.

*fend()* é chamada quando o jogo acaba, e retorna uma string com o nome do time vencedor.

A classe *Checkers* extende a classe *Game*. Seu método construtor(**__init__()**) cria um espaço bidimensional(classe *Space*) quadrado com lado medindo 8 unidades, dividido em um *Grid* 8x8 de casas(*Cells*) de tamanho 1. Cria os times(*Team*) whites e blacks e as *Layers* active e inactive. A primeira contém as peças que estão em jogo e a segunda, as que ja foram capturadas. Em seguida todas as peças são instanciadas, atribuídas a um dos dois times e associadas a um objeto

piecebehavior(classe *Script*). O jogo então é instanciado com as funções de finit, fturn e fend como parâmetros para os argumentos de mesmo nome da classe *Game*.

Os resultados da execução deste cenário de testes foi bastante positivo. O código gerado teve menos de quinhentas linhas e é bastante claro, cumprindo a exigência de ser uma ferramenta de fácil utilização. Ele está em `icad.puc-rio.br/~gameshow/frameworkGSI/checkers/`. Quanto à consistência entre estados, o teste também foi bem sucedido. Com a lista de estados foi possível replicar o jogo, a cada movimento, em outro ambiente. Foram simuladas diversas interações simultâneas, o que imita um ambiente multijogador. O software não apresentou problemas para identificar quais delas representavam movimentos válidos e, dentre estes, eleger qual seria executado.

A lista de diferenças comprovou que, se forem transmitidas somente as diferenças entre dois estados, é possível reconstruir o mais recente em sua integridade, desde que de posse do mais antigo. Sendo assim, não se faz necessária a transmissão do estado inteiro para os usuários, a menos que um deles o solicite por algum motivo de interrupção de comunicação que advenha da assincronicidade. No caso específico do teste, um estado completo ocupa 13753 bytes em formato *plaintext string*, enquanto a diferença entre dois estados consecutivos ocupa somente 385 bytes. Evidentemente que, no momento da transmissão, tais dados podem ser compactados, mas a disparidade entre os dois é notável. É importante observar, ainda, que o estado fornecido pelo framework detalha informações de todos os objetos. No caso específico de um possível jogo, algumas destas informações podem ser redundantes, o que faz com que este tamanho possa ser diminuido ainda mais.

# 7
# Conclusões Finais e Trabalhos Futuros

Esta dissertação apresenta uma área de pesquisa ainda não explorada na literatura que trata da convergência entre televisão e videogame, propondo um Game Show em Realidade Aumentada com interatividade dos telespectadores via dispositivos móveis como segunda tela. Nesta dissertação, denomina-se esta área de Game Show Interativo

Foi relatado todo o processo de surgimento desta pesquisa e sua inspiração no projeto MarkerFinder(capítulo 3), onde a tecnologia (LightCraft) foi apontada como promissora para a garantia de qualidade dos elementos de Realidade Aumentada. Isto permitiu a modelagem de um framework para o desenvolvimento de Game Shows Interativos.

Dos três módulos propostos para o framework, somente um está implementado: o módulo central -Core-, responsável por regular as regras, o estado de jogo e a comunicação entre os jogadores. Os outros dois módulos foram simulados, para finalidades de testes e de análise dos conceitos propostos.

O teste do módulo Core, descrito na seção 6.1, indica que a modelagem inicial do framework está correta e que ele permite o desenvolvimento de Game Show Interativos de forma fácil para alguém que tenha conhecimento de programação. Além disso, este teste foi implementado de uma maneira que colocou à prova a capacidade do módulo central de lidar com múltiplas interações simuladas e decidir como proceder em caso de conflitos. Conforme a implementação do framework progredir em trabalhos futuros, testes poderão ser feitos relativos à computação gráfica gerada em estúdio; às interfaces e transmissões de informação dos dispositivos móveis; e à performance e custo computacional do conjunto.

O andamento atual da pesquisa ainda não permite afirmar sobre a viabilidade dos Game Shows Interativos como entretenimento. O único experimento capaz de fornecer esta informação é um teste de público. O trabalho relatado neste documento indica que é tecnologiamente possível desenvolvê-los. Estudos futuros atestarão para o valor social de tal formato.

## 7.1 Trabalhos Futuros

O estado atual da pesquisa apresentada nesta dissertação aponta alguns rumos possíveis. A conclusão da implementação do framework é, evidentemente, uma condição necessária para que estes sejam perseguidos. A próxima etapa da pesquisa é implementar o módulo de estúdio, pois a segunda tela -módulo mobile- depende da primeira para ser testada. Além disso, os desafios técnicos deste módulo serão de grande calibre. É importante desenvolvê-lo antes pois as restrições encontradas durante sua implementação poderão limitar certas possibilidades no módulo mobile.

Durante a implementação dos módulos faltantes -estúdio e mobile- será necessário decidir, definitivamente, as interfaces que serão utilizadas para a comunicação entre as partes e os requisitos de infraestrutura para a execução do framework. Existem dois grandes pontos de consumo de recursos computacionais que necessitam de atenção: a composição da imagem em alta definição com Realidade Aumentada em tempo real; e o overhead de comunicação, que cresce com o número de usuários de segunda tela conectados.

O framework, como citado no capítulo 5.2, é uma ferramenta para construção de Game Shows Interativos, que, por sua vez, são a mídia escolhida para estudar a convergência entre TV e jogos. Sendo assim, ele não é o produto final da pesquisa. Uma vez que seja de fato possível construir um jogo neste formato, o estudo de sua viabilidade como meio de entretenimento e ferramenta educativa pode prosseguir.

A seção 7.1.1 apresenta algumas propostas de Game Shows Interativos, possíveis uma vez que o framework esteja pronto. Em 7.1.2 são levantados possíveis trabalhos a serem realizados no futuro, que também estão na convergência entre televisão e videogame.

### 7.1.1 Possibilidades de Game Show Interativos

Uma característica peculiar do formato proposto é a disparidade entre a informação acessível ao participante e aos telespectadores. O participante está no estúdio e não tem acesso á imagem em Realidade Aumentada com a mesma qualidade do que a exibida na TV, a menos que se use algum aparato como óculos de Realidade Virtual. Além disso, os telespectadores possuem dispositivos móveis, que podem exibir ainda mais informação. Um desenvolvedor deve incorporar esta diparidade como mecânica de jogo. Alguns esboços de Game Shows Interativos estão a seguir.

Um jogo onde o participante tenha que descobrir em que lugar ou época está, ou que personalidade ele é. Os telespectadores o ajudariam a descobrir vendo os elementos que ele não vê. Este seria um jogo educativo mais voltado para crianças em idade escolar.

É possível colocar o telespectador no papel de detetive, procurando pistas no cenário que seriam elementos virtuais, às quais o participante não tem acesso. Este conceito ainda está muito pouco desenvolvido, principalmente no que diz respeito à participacão de quem estiver no palco.

Baseado no modo *multiplayer* dos jogos da franquia Assassins Creed(AssassinsCreed) e na consagrada série de livros "Onde está Wally?"(Handford, 1987), um jogo onde o participante se escondesse em meio a personagens virtuais e os telespectadores tivessem que o encontrar tem um potencial considerável de diversão. O problema notado nessa ideia é que se trata de um jogo que só funciona se houver poucos telespectadores jogando, e não necessariamente seria interessante apenas assistir como espectador passivo. Isto originou a preocupação com a proporcionalidade inversa entre o poder individual de um telespectador e o número de telespectadores que podem participar. Esta dualidade foi melhor elaborada no capítulo 5.1.

A possibilidade de criar uma disparidade de informação não somente entre participante e telespectador, mas dentro do próprio grupo de telespectadores foi levantada mais recentemente durante esta pesquisa. Se telespectadores tiverem informações diferentes e tiverem que trabalhar em equipe para solucionar um problema, por exemplo, comportamentos interessantes podem emergir.

Muitos dos Reality Shows de sucesso utilizam um modelo recorrente: um grupo de concorrentes é apresentado ao público e este os vai eliminando até sobrar um vencedor. Uma adaptação de um Game Show Interativo para este modelo tem chances de capturar o público. O voto pode ser substituído por interações, como uma prova de obstáculos onde o telespectador escolha quem atrapalhar e quem ajudar, por exemplo. Quem já se diverte votando e interferindo no rumo do programa terá uma diversão a mais.

### 7.1.2 Possibilidades da Pesquisa

O Oculus Rift(Oculus) e seus similares vêm indicando uma tendência para o campo de Realidade Virtual. Ele consiste em um periférico que se acopla da cabeça de um usuário, como óculos, e gera imagens estereoscópicas. Além disso ele pode ser usado com um fone de ouvido. Com um acelerômetro e outros sensores, ele é capaz de captar a orientação da cabeça e projetar uma

imagem de um universo virtual nos visores, bem como gerar áudio estéreo que corresponda a esta orientação. Atualmente, esse tipo de acessório não possibilita a livre movimentação do usuário pelo cenário virtual sem o uso de algum outro periférico, como o (Omni), por exemplo. Este consiste em uma esteira que permite passos em todas as direções. Quando usados em conjunto, a esteira determina a posição do jogador no espaço virtual, e o Oculus, a direção de seu olhar(figura 7.1).


Figura 7.1: *Oculus Rift(Oculus) à esquerda e Omni treadmill(Omni) à direita.*



As duas razões pelas quais o Oculus, sozinho, não consegue lidar com a mudança da posição do jogador são a ausência de sensores para tal e, mais importante, a diferença entre o formato geográfico de um ambiente virtual e do ambiente real onde o jogador de fato está. Se uma câmera direcionada para cima for acoplada a um par de óculos de realidade virtual e o teto de um ambiente real for mapeado com marcadores fiduciários, o uso de uma tecnologia como o (LightCraft) permitiria o livre movimento de um jogador por um cenário real que coincida com um espaço virtual. Um Game Show Interativo que se utilizasse disto seria muito mais imersivo para o participante, que se veria no mesmo ambiente de RA que os telespectadores. Esta proposta trata de uma pesquisa multidisciplinar, pois exige um trabalho tanto em software quanto em hardware.

Outra possibilidade de pesquisa é um cenário um pouco diferente do formato proposto. O know-how técnico adquirido ao final de uma implmentação completa possibilitará o desenvolvimento de uma aplicação com potencial para a área de educação. Como uma alternativa interessante, pode-se imaginar um cenário onde os participantes estivessem no estúdio e o público, ao invés de em casa, na platéia. Diferentes membros do público poderiam visualisar diferentes elementos virtuais, e resolver um problema trabalhando em equipe e combinando as informações adquiridas. Este experimento não é origatoramente um programa de TV, mas pode ser realizado em algum outro tipo de ambiente.

Uma futura linha completamente diferente de investigações envolve a pesquisa de aspectos sociais e comportamentais quando televisão e jogos se misturam, representando a convergência de dois dos mais impactantes e abrangentes veículos audiovisuais de cultura atualmente.Entretanto, esta linha de pesquisa é essencialmente interdisciplinar, envolvendo aspectos sociológicos. culturais, eticos e tecnológicos.

# 8
# Referências Bibliográficas


AMAZON. **Amazon.com, Inc.** www.amazon.com. Último acesso em 25/03/2015. 5.1

APPERLEY, T. H. Genre and game studies: Toward a critical approach to video game genres. **Simulation & Gaming**, Sage Publications, v. 37, n. 1, p. 6–23, 2006. 4.1

ASSASSINSCREED. **Assassins Creed Series**. [S.l.]: Ubisoft. assassinscreed.ubi.com. Último acesso em 25/03/2015. 7.1.1

BAFFA, A.; FEIJó, B. Automated storytelling based on dynamic audience interaction. 2015. 5.1

BAMZOOKI. **Bamzooki**. [S.l.]: BBC. www.bbc.co.uk/cbbc/shows/bamzooki. Último acesso em 25/03/2015. (document), 1, 1.1, 2.1, 2.1

BATES, B. Designing the puzzle. In: **Proceedings of the 1997 Game Developers Conference**. [S.l.: s.n.], 1997. 4.1.1

BATES, J.; LOYALL, B.; REILLY, W. S. **An architecture for action, emotion, and social behavior**. [S.l.]: Springer, 1994. 4.1.1

BBC. **BBC TV**. www.bbc.com/tv. Último acesso em 25/03/2015. 1, 2.1

BEAMLY. **Beamly**. about.beamly.com/. Último acesso em 25/03/2015. 2.3

BETHESDA. **Bethesda**. http://bethsoft.com/en-us. Último acesso em 25/03/2015. 4.1.1

BLIZZARD. **Blizzard**. http://us.blizzard.com/en-us/. Último acesso em 25/03/2015. 4.1.1

BOOST. **Boost Test Library**. http://www.boost.org/doc/libs/1_55_0/libs/test/doc/html/index.html. Último acesso em 25/03/2015. 3.6, 3.6.1

BRAID. **Braid**. [S.l.]: Number None Inc & Microsoft Game Studios, 2009. http://braid-game.com/. Último acesso em 25/03/2015. (document), 4.1.1, 4.15

CALLOFDUTY. **Call of Duty**. [S.l.]: Activision and Square Enix, 2003. www.callofduty.com. Último acesso em 25/03/2015. 4.1.1

CBS. **Video game ratings made by anonymous panel.** [S.l.]: CBS News, 2011. cbsnews.com/news/video-game-ratings-made-by-anonymous-panel/. Último acesso em 25/03/2015. 1

CIARLINI, A. E. et al. A logic-based tool for interactive generation and dramatization of stories. In: **Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology**. [S.l.]: ACM's Special Interest Group on Computer–Human Interaction, 2005. p. 133–140. 4.1.1

CIVILIZATION. **Civilization V**. [S.l.]: 2K Games & Aspyr, 2010. http://www.civilization5.com/. (document), 4.1.1, 4.11

CSIKSZENTMIHALYI, M. **Flow: The Psychology of Optimal Experience**. [S.l.]: Harper & Row, 1990. ISBN 0060162538. 4.1

DEUSEX. **Deus Ex**. [S.l.]: Eidos Interactive and Square Enix, 2000. www.deusex.com/. Último acesso em 25/03/2015. (document), 4.1.1, 4.18

DIABLO3. **Diablo 3**. [S.l.]: Blizzard, 2012. http://us.battle.net/d3/en/. (document), 4.1.1, 4.3

DOXYGEN. **Generate documentation from source code**. http://www.stack.nl/~dimitri/doxygen/. Último acesso em 25/03/2015. 3.6.2, 6

ELDERSCROLLS. **Elder Scrolls, The**. [S.l.]: Bethesda. http://www.elderscrolls.com/. Último acesso em 25/03/2015. 4.1.1

FALLOUT. **Fallout New Vegas**. [S.l.]: Bethesda, 2010. http://fallout.bethsoft.com/eng/home/home.php?country=us. Último acesso em 25/03/2015. (document), 4.1.1, 4.5

FARO. **Faro Technologies**. http://www.faro.com/home. Último acesso em 25/03/2015. 3.2

FEZ. **FEZ**. [S.l.]: Microsoft Studios, 2012. http://fezgame.com/. Último acesso em 25/03/2015. (document), 4.1.1, 4.16

FLIGHTSIM. **Microsoft Flight Simulator X**. [S.l.]: Microsoft, 2006. www.microsoft.com/products/games/FSXInsider/fsxlauncher.aspx. (document), 4.6, 4.1.1, 4.7

FLOWER. **Flower**. [S.l.]: Thatgamecompany. = http://thatgamecompany.com/games/flower/. Último acesso em 25/03/2015. (document), 4.1.1, 4.22

FOCUS. **Faro Laser Scanner Focus**. http://www.faro.com/products/3d-surveying/laser-scanner-faro-focus-3d/overview. Último acesso em 25/03/2015. (document), 3.2, 3.3

FSCENE. **Faro Scene**. http://www.faro.com/products/faro-software/scene/overview. Último acesso em 25/03/2015. 3.2

GARTNER. **Forecast: Video Game Ecosystem, Worldwide, 4Q13.** [S.l.]: Gartner Says, 2013. www.gartner.com/newsroom/id/2614915. Último acesso em 25/03/2015. 1

GARTNER. **Worldwide PC, Tablet and Mobile Phone Combined Shipments to Reach 2.4 Billion Units in 2013**. [S.l.]: Gartner Says, 2013. www.gartner.com/newsroom/id/2408515. Último acesso em 25/03/2015. 1

GIT. **GNU Interactive Tools**. http://git-scm.com/. Último acesso em 25/03/2015. 3.7

GLOBO. **Rede Globo**. http://redeglobo.globo.com/. Último acesso em 25/03/2015. (document), 1, 3, 3.1, 3.1

GUITARHERO. **Guitar Hero Series**. [S.l.]: Activision, 2005. www.guitarhero.com. Último acesso em 25/03/2015. (document), 4.1.1, 4.19

GYGAX, G.; ARNESON, D. **Dungeons and Dragons**. [S.l.]: Wizards of the Coast. http://dnd.wizards.com/. 4.1.1

HANDFORD, M. **Where's Wally?** [S.l.]: Walker Books, 1987. ISBN 0316342939. 7.1.1

HARRYPOTTER. **Harry Potter and the Philosopher's Stone**. [S.l.]: Warner Bros. Pictures, 2001. (document), 1.2

HOLMES, S. **The Quiz Show (TV Genres)**. [S.l.]: Edinburgh University Press, 2008. ISBN 9780748627530. 4.2

HOUGH, P. V. C.; MICH, A. A. Method and means for recognizing complex patterns. 1962. 3, 2, 2, 13

IKARUGA. **Ikaruga**. [S.l.]: Infogrames Inc., 2003. (document), 4.1.1, 4.20

INGRESS. **Ingress**. [S.l.]: Niantic Labs, 2013. www.ingress.com. Último acesso em 25/03/2015. (document), 2.2, 2.2, 2.3

INTRASONICS. **Intrasonics SDK**. [S.l.]: Intrasonics, 2013. www.intrasonics.com/sdk-summary.html. Último acesso em 25/03/2015. 2.3

JONES, E. What makes strategy games exciting? In: **San Francisco IGDA's Pecha Kucha**. [S.l.: s.n.], 2011. 4.1.1

JOURNEY. **Journey**. [S.l.]: Thatgamecompany. = http://thatgamecompany.com/games/journey/. Último acesso em 25/03/2015. (document), 4.1.1, 4.23

JSON. **JavaScript Object Notation**. http://json.org/. Último acesso em 25/03/2015. 5.2, 6, A.1

LASTOFUS. **Last of Us, The**. [S.l.]: Naughty Dog and Sony Computer Entertainment, 2013. = www.thelastofus.playstation.com/. Último acesso em 25/03/2015. (document), 4.1.1, 4.17

LATEX. **LaTeX Project**. http://www.latex-project.org/. Último acesso em 25/03/2015. 6

LIGHTCRAFT. **LightCraft Technology**. http://www.lightcrafttech.com/. Último acesso em 25/03/2015. (document), 2.1, 3.1, 3.2, 5, 7, 7.1.2

LOL. **League of Legends**. [S.l.]: Riot Games, 2009. leagueoflegends.com/. (document), 4.1.1, 4.12

MCQUAIL, D. **Mass communication theory: An introduction .** [S.l.]: Sage Publications, Inc, 1987. 4.2

MCQUAIL, D.; BLUMLER, J. G.; BROWN, J. R. The television audience: A revised perspective. **Media studies: A reader**, v. 271, p. 284, 1972. 4.2

MILLAN, M. P. B. Reality shows: uma abordagem psicossocial. **Psicologia: ciência e profissão**, Conselho Federal de Psicologia, v. 26, n. 2, p. 190–197, 2006. 4.2

NIELSEN. **Double Vision - Global Trends in Tablet and Smartphone Use While Watching TV**. 2012. [breaklines=true] http://www.nielsen.com/us/en/insights/news/2012/double-vision-global-trends-in-tablet-and-smartphone-use-while-watching-tv.html. Último acesso em 25/03/2015. 2.3

OCULUS. **Oculus Rift**. [S.l.]: Oculus. www.oculus.com. Último acesso em 25/03/2015. (document), 1, 7.1.2, 7.1

OMNI. **Omni Treadmill**. [S.l.]: Virtuix. www.virtuix.com. Último acesso em 25/03/2015. (document), 7.1.2, 7.1

OPENCV. **Open Source Computer Vision Library**. http://opencv.org/. Último acesso em 25/03/2015. 3, 3.6, 5.2

POKEDEX. **Pokedex 3D Pro**. [S.l.]: Nintendo. www.pokedex3d.com/en-us/get-pokedex3d-pro. Último acesso em 25/03/2015. (document), 2.2, 2.5

POKEMON. **Pokemon Company, The**. [S.l.]: Nintendo. www.pokemon.com. Último acesso em 25/03/2015. 2.2

PSVITA. **PlayStation Vita**. [S.l.]: Sony Computer Entertainment. www.playstation.com/en-us/explore/psvita/. Último acesso em 25/03/2015. 2.2

PYTHON. **Python**. [S.l.]: Python Software Foundation. www.python.org. Último acesso em 25/03/2015. 5.2, 6

ROCKBAND. **Rock Band Franchise**. [S.l.]: EA Games and MTV Games, 2007. http://www.harmonixmusic.com/games/rock-band/. Último acesso em 25/03/2015. 4.1.1

SCHELL, J. **The Art of Game Design. A Book of Lenses**. [S.l.]: Morgan Kaufmann, 2008. ISBN 9780123694966. (document), 4.1, 4.1

SHAZAM. **Shazam for TV**. [S.l.]: Shazam. http://www.shazam.com/. Último acesso em 25/03/2015. 2.3

SIMCITY. **SimCity**. http://www.simcity.com/. Último acesso em 25/03/2015. (document), 4.1.1, 4.8

SIMS. **Sims, The**. www.thesims.com/. Último acesso em 25/03/2015. (document), 4.1.1, 4.9

SKYRIM. **The Elder Scrolls V: Skyrim**. [S.l.]: Bethesda, 2011. http://www.elderscrolls.com/skyrim. Último acesso em 25/03/2015. (document), 4.1.1, 4.4

STARCRAFT. **StarCraft II: Wings of Liberty**. [S.l.]: Blizzard, 2010. http://us.battle.net/sc2/en/. Último acesso em 25/03/2015. (document), 4.1.1, 4.10

SUPERMEATBOY. **Super Meat Boy**. http://www.simcity.com/. Último acesso em 25/03/2015. (document), 4.1.1, 4.21

THATGAMECOMPANY. = thatgamecompany.com. Último acesso em 25/03/2015. 4.1.1

THUE, D. et al. Interactive storytelling: A player modelling approach. In: **AIIDE**. [S.l.: s.n.], 2007. p. 43–48. 4.1.1

TODE, C. **ABC makes show shoppable via mobile**. 2013. = www.mobilecommercedaily.com/scandal-makes-show-shoppable-via-mobile. Último acesso em 25/03/2015. 2.3

TODE, C. **National Geographic Channel targets tablet users to drive co-viewing**. 2013. = www.mobilemarketer.com/cms/news/advertising/14806.html. Último acesso em 25/03/2015. 2.3

TWITCHTV. **twitchtv**. www.twitch.tv/. Último acesso em 25/03/2015. 5.1

WHANNEL, G. et al. The television game show simposium keynote. In: . [S.l.]: University of South Wales, 2008. 4.2

WHARTON, S. **3...2...1...PulzAR! PS Vita Gets Augmented Reality Puzzle Game Today**. blog.us.playstation.com/2012/06/12/3-2-1-pulzar-ps-vita-gets-augmented-reality-puzzle-game-today/. Último acesso em 25/03/2015. (document), 2.4, 2.2

WOW. **World of Warcraft**. [S.l.]: Blizzard, 2004. http://us.battle.net/wow/en/. (document), 4.1.1, 4.3

ZELDA. **Legend of Zelda: SkyWard Sword, The**. [S.l.]: Nintendo, 2011. http://www.zelda.com/skywardsword/. Último acesso em 25/03/2015. (document), 4.14

# A
# Anexos

## A.1
## Descrição detalhada das classes do Framework

Este anexo detalha mais minuciosamente as classes descritas na seção 5.3.

Os atributos estão em azul e os métodos e seus argumentos em vermelho. É importante ressaltar que certos atributos são funções, então também estarão em azul (ex: atributos onCreate, onUpdate, onCollision e onInteraction da classe Script). Tudo que for estático, seja atributo ou método, estará em verde. No caso de herança, a superclasse estará entre parenteses.

–**GameSpace**

Define o espaço tridimensional do jogo. Via de regra é o espaço do mapeado do estúdio.

**xmin** é a coordenada x mínima que pertence ao espaço definido.
**xmax** é a coordenada x máxima.
**ymin** é a coordenada y mínima.
**ymax** é a coordenada y máxima.
**zmin** é a coordenada z mínima.
**zmax** é a coordenada z máxima.
**origin** é a origem do sistema de coordenadas. A princípio, é o ponto (0, 0, 0).
**grid** é o grid que divide o chão do espaço. Pode ser nulo, no caso do desenvolvedor não desejar discretzar o espaço do jogo.
**__init__(xmin, xmax, ymin, ymax, zmin, zmax, origin, cellsize)** é o método contrutor. **xmin**, **xmax**, **ymin**, **ymax**, **zmin**, **zmax** e **origin** correspodem aos atributos de mesmo nome. **cellsize** é o tamanho de cada célula que se deseja ter no **grid**.

–**Grid**

Divide o chão do GameSpace -plano onde a coordenada Z é zero- em um Grid. Dependendo do jogo desenvolvido, pode-se poupar tempo de

processamento considerando o chão como um tabuleiro em vez de um espaço contínuo.

**cellsize** é o tamanho de cada célula(Cell) do Grid.
**xmin** é a coordenada x mínima que pertence ao grid.
**xmax** é a coordenada x máxima que pertence ao grid.
**ymin** é a coordenada y mínima que pertence ao grid.
**ymax** é a coordenada y máxima que pertence ao grid.
**g** é a matriz de células(Cell). g possui $(xmax - xmin)/cellsize \times (ymax - ymin)/cellsize$ células.
**__init__(xmin, xmax, ymin, ymax, cellsize)** é o método contrutor. Os argumentos correspondem aos atributos de mesmo nome.
**getGridPosition(pos)** retorna as coordenadas da célula(Cell) que contenha a posição **pos**.
**getCellObjects(x, y)** retorna uma lista de todos os GameObjects que estão dentro da célula de coordenadas **x**, **y**.

–**Cell**

É uma célula, ou casa, do Grid.

**xb** significa "x begin". É a coordenada x mínima que pertence á célula.
**xe** significa "x end". É a coordenada x maxima que pertence á célula.
**yb** significa "y begin". É a coordenada y mínima que pertence á célula.
**ye** significa "y end". É a coordenada y maxima que pertence á célula.
**gameobjects** é uma lista de GameObjects que estiverem na célula em um dado momento.
**__init__(xb, xe, yb, ye)** é o método construtor. Os argumentos correspondem aos atributos de mesmo nome
**addObject(obj)** insere o GameObject **obj** na célula, ou seja, insere **obj** em **gameobjects**.
**removeObject(obj)** remove o GameObject **obj** da célula, ou seja, remove **obj** de **gameobjects**.
**getObjects()** retorna a lista **gameobjects**.
**center()** retorna as coordenadas X e Y do ponto central da célula.

–**GameObject**

Representa todos os objetos dentro de um jogo, desde corpos inanimados a personagens e jogadores. Classe para ser herdada por outras mais específicas.

**objid** é o número identificador do GameObject
**parent** é uma referência ao GameObject pai, se houver um.
**layer** é a camada(Layer) à qual o GameObject pertence.
**tags** é uma lista de tags, um subconjunto da lista estática **tagdomain**.
**visibility** é uma lista de outros GameObjects que conseguem enxergar o objeto.
**interactability** é uma lista de outros GameObjects que conseguem interagir com o objeto.
**position** é a posição tridimensional do GameObject.
**gridposition** é um vetor bidimensional com as coordenadas da célula(Cell) do Grid a que a posição **position** corresponde.
**lastgridposition**é um vetor bidimensional com as coordenadas da célula(Cell) do Grid que correspondiam a posição do GameObject no último turno antes do turno mais atual.
**rotation** é o vetor tridimensional que indica a rotação do objeto em relação a **parent**. Se **parent** for nulo, a rotação é relativa a origem do GameSpace(**origin**).
**scale** é um vetor tridmensional que indica a escala do GameObject
**lookat** é um vetor tridimensional normalizado que aponta na direção para a qual o objeto está olhando.
**isrigid** é uma variável booleana que indica se o GameObject é rígido, ou seja, sujeito a colisões.
**interactiondomain** é uma lista de interações possíveis(Interaction) que o objeto pode sofrer.
**interactiontimelimit** é o limite de tempo que os jogadores(Player) tem, por turno, para interagir com o objeto.
**interactionQ** é a fila de interações(Interaction) que o objeto sofreu e ainda não foram tratadas.
**script** é um objeto da classe Script que dita o comportamento do GameObject.
**team** é o time(Team) ao qual o GameObject pertence.
**latency** é o número de próximos turnos durante os quais o GameObject permanecerá inativo, ou latente.
**score** é a pontuação do GameObject.
**__init__(objid, position, space, rotation, scale, lookat, isrigid, parent, layer, tags, visibility, interactability, interactiondomain, interactiontimelimit, script, team, latency)** é o método construtor. Os argumentos correspondem aos atributos de mesmo nome. O atributo **space** é uma referência a um objeto da classe GameSpace.

**getSuperClassName()** retorna o nome da super classe mais alta na hierarquia de herança do objeto.
**addTags(tags)** adiciona a lista de tags recebida como argumento a **tags**.
**removeTags(tags)** remove cada elemento da lista de tags recebida como argumento de **tags**.
**addInteraction(interaction, interactor, lcond, lcons)** inclui uma interação em **interactionQ** desde que o tipo dela esteja em **interactiondomain**. **interactor** é o GameObject autor da interação. Tipicamente um membro da classe Player. **lcond** e **lcons** são as listas de parâmetros para as funções **fconditions** e **fconsequences**, respectivamente. Ambas são atributos da classe interaction
**gameobjects** é uma lista estática de todos os GameObjects do jogo.
**tagdomain** é um lista estática das tags do jogo. Um GameObject pode ter múltiplas tags, desde que todas pertençam a **tagdomain**.
**initTagDomain(tagdomain)** inicializa o atributo estático **tagdomain** com o conjunto de tags recebido como argumento.
**objFromId(objid** retorna o GameObject dentre os que estão na lista estática **gameobjects** que possui o atributo **objid** igual ao argumento.

–**Character(GameObject)**
Um GameObject que pode se mover pelo espaço e ter um comportamento. Uma especificação de GameObject que exclui objetos inanimados.

**velocity** é o vetor velocidade, tridimensional, de um Character.
**acceleration** é o vetor aceleração, tridimensional, de um Character.
**__init__(objid, layer, tag, visibility, interactability, position, rotation, scale, lookat, isrigid, interactiondomain, fprocessinteraction, interactiontimelimit, supdate, team = None, latency = 0, v0, a0)** é o método construtor. **v0** e **a0** são os vetores velocidade inicial e aceleração inicial, respectivamente. Todos os outros argumentos são para inicialização da super classe(GameObject).

–**Player(Character)**
Um Jogador. No momento, ou jogador de estúdio ou doméstico. A classe player existe somente como classe intermediária entre Character e os tipos mais específicos de jogadores.

–**HomePlayer(Player)**
Jogador doméstico.

–**TVPlayer(Player)**
Jogador de estúdio.

–**Layer**
Um conjunto de GameObjects. Agrupar objetos em layers pode melhorar o tempo de processamento sempre que se tem que iterar por eles.

**layername** é o nome da Layer
**objects** é uma lista com todos os GameObjects que pertencem à Layer.
**__init__(layername, initialobjects)** é o método construtor. **layername** é o nome a ser dado à Layer e **initialobjects** é uma lista com todos os GameObjects a serem inseridos em **objects** no momento da instanciação da Layer.
**addObject(obj)** insere o GameObject **obj** na Layer.
**removeObject(obj)** remove o GameObject **obj** da Layer.
**layers** é uma lista estática com todas as Layers do jogo
**getLayerByName(layername)** retorna a Layer cujo nome é igual ao argumento.

–**Collision**
Classe que contem informações sobre uma colisão entre dois GameObjects.

**culprit** é um dos objetos envolvidos na colisão.
**victim** é o outro objeto envolvido na colisão.
**turn** é o número do turno em que a colisão ocorreu.
**__init__(culprit, victim)** é o método construtor. Os argumentos correspondem aos atributos de mesmo nome.

–**Script**
Um script para ditar o comportamento de um GameObject. O desenvolvedor escreve o script e o anexa a um objeto. Quase todos os atributos de um Script são funções que serão chamadas em momentos específicos do jogo. O desenvolvedor fica livre para modelar as funções da maneira mais conveniente.

**owner** é o GameObject que estará sujeito ao comportamento ditado pelo Script.

**onCreate** é a função a ser chamada logo apos a instanciação de **owner**.
**onUpdate** é uma função a ser chamada a cada turno, responsável por atualizar o estado de **owner**.
**onCollision** é um afunção a ser chamada a cada colisão de **owner** com outro GameObject.
**onInteraction** é um afunção a ser chamada sempre que **owner** for alvo da interação de um jogador(Player).
**__init__(owner, onCreate, onUpdate, onCollision, onInteraction)** é o método construtor. Os argumentos correspondem aos atributos de mesmo nome.

– **Interaction**

Trata a interação entre jogadores(Player) e outros GameObjects. Basicamente um conjunto de condições e consequências. Quando o objeto sofre um tipo de interação específica (condições), as consequências ocorrem.

**interactiontype** é o tipo da interação dentre os possíveis tipos da lista estática **possibletypes**.
**fconditions** é uma função que booleana que retorna True quando as condições que constituem a interação são cumpridas e False caso contrário.
**fconsequences** é a função a ser chamada quando **fconditions** retornar True, ou seja, é a função que executa as consequências da interação.
**__init__(interactiontype, interactor, fconditions, fconsequeces)** é o método construtor. Os argumentos correspondem aos atributos de mesmo nome.
**interact(lcond, lcons)** é a função que executa a interação. Ela chama **fconditions** passando a lista **lcond** como parâmetro e, se o resultado for True, chama **fconsequences** com a lista **lcons** como parâmetro.
**interactions** é uma lista estática que contém todos as interações existentes no jogo.

– **Team**

Um time de GameObjects. Inicialmente, membros de um time deveriam ser da classe Player. Essa restrição foi abolida para dar mais liberdade ao desenvolvedor.

**score** é a pontuação do time.
**teamid** é o numero identificador do time.

**name** é o nome do time.
**members** é a lista de GameObjects membros.
**fscore** é uma função que calcula a pontuação do time. Tipicamente a pontuação do time é calculada somando as pontuações dos membros.
**__init__(teamid, name, initialmembers, fscore)** é o método construtor. Todos os argumentos correspondem aos atributos de mesmo nome, com exceção de **initialmembers**, que corresponde a lista inicial dos membros do time a ser inserida em **members**.
**updateScore()** atualiza a pontuação do time.
**getScore()** retorna a pontuação do time.
**addMember(member)** inclui o GameObject **member** ao time.
**addMembers(members)** inclui todos os GameObjects da lista **members** ao time.
**removeMember(member)** remove o GameObject **member** do time.
**removeMembers(members)** remove todos os GameObjects da lista **members** do time.
**memberIn(member)** responde True se o GameObject **member** estiver no time e False se não.
**teams** é uma lista estática que contém todos os times.
**getTeamById(teamid)** retorna o time cujo atributo **teamid** seja igual ao argumento.
**getTeamByName(name)**retorna o time cujo atributo **name** seja igual ao argumento.

### –Poll

Uma votação, ou enquete. Pergunta múltipla escolha dirigida aos jogadores.

**question** é a pergunta que se quer responder através da votação.
**data** é a lista das possíveis respostas, cada uma com um possível objeto atrelado.
**turns** é o número de turnos até a votação expirar.
**isactive** é uma variável booleana para indicar se a Poll está ativa.
**fformat** é uma função para formatar as strings tanto em **question** quanto em cada uma das respostas em **data**. Esta função pode ser usada, por exemplo, para inserir o nome de um objeto atrelado a uma resposta ao texto da resposta.
**__init__(question, data, turns, isactive, fformat)** é o mtodo construtor. Os argumentos correspondem aos atributos de mesmo nome.
**computeVote(answer, votes)** computa o número de votos indicados

em **vote** para a resposta **answer**.
**getRank()** retorna a o resultado da votação ordenado decrescentemente por número de votos.
**getWinner()** retorna a resposta que recebeu mais votos.
**newTurn(turns)** computa **turns** mudanças de turno.
**formatedstr()** retorna uma lista das strings da votação, sendo a primeira a pergunta e as outras as respostas, formatadas de acordo com **fformat**.

–**Game**

Classe que contém todos os atributos e funções necessárias para rodar o jogo, ou seja, estabelecer um estado e suas transições.

**name** é o nome do jogo.
**description** é uma descrição breve para o jogo.
**space** é o espaço físico(GameSpace) onde o jogo será executado.
**gameobjects** é uma lista de todos os objetos(GameObject) do jogo.
**rigidobjects** é um subconjunto de **gameobjects** que contém somente os GameObjects que são rígidos.
**teams** é uma lista com todos os times(Team) do jogo.
**isdiscrete** é uma variável booleana que indica se o espaço do jogo será discretizado ou não. Basicamente, indica se o Grid de **space** será utilizado.
**polls** é a lista de votações(Polls).
**activepolls** é um subconjunto de **polls** que contem somente as votações ativas.
**finit** é a função que será chamada para incializar o jogo, ou seja, ela cria o estado inicial.
**fturn** é uma função que será chamada ao início de cada turno.
**fend** é uma função que será chamada ao final do jogo.
**lastturn** é uma variável booleana que indica se o turno atual será o último.
**turncount** é o número do turno atual.
**collisions** é uma lista de colisões(Collisions) a serem tratadas.
**statebuff** é uma lista contendo o estado atual do jogo e alguns estados anteriores.
**__init__(name, description, space, gameobjects, teams, isdiscrete, polls, finit, fturn, fend, statebuffsize)** é o método construtor. Os argumentos correspondem aos atributos de mesmo nome.
**statebuffsize** é o tamanho a ser dado ao atributo **statebuff**

**run()** é a função que inicia a execução do jogo.
**addPoll(poll)** adiciona a votação **poll** à lista **polls**. Por se tratar de um programa de TV ao vivo, é interessante que se possa inserir perguntas no meio da execução do jogo.
**switchOffPoll(poll)** desativa a votação **poll**.
**switchOnPoll(poll)** ativa a votação **poll**.
**linitialstate** é o estado inicial do jogo, descrito em um dicionário com formato compreensível por JSON(JSON).
**laststate** é o estado do último turno do jogo.
**currentstate** é o estado atual do jogo.