Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Construção de um jogo eletrônico multiusuário em uma superfície de projeção multitoque

Pedro Guerra Brandão
Saulo Camarotti Rayol Braga

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientadora
Prof.ª Dr.ª Carla Denise Castanho

Coorientador
Prof. Dr. Ricardo Pezzuol Jacobi

Brasília
2009

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharelado em Ciência da Computação

Coordenadora: Prof.ª Dr.ª Carla Maria Chagas e Cavalcante Koike

Banca examinadora composta por:

Prof.ª Dr.ª Carla Denise Castanho (Orientadora) — CIC/UnB
Prof. Dr. Ricardo Pezzuol Jacobi (Coorientador) — IE/UnB
Prof. Dr. Díbio Leandro Borges — CIC/UnB

CIP — Catalogação Internacional na Publicação

Brandão, Pedro Guerra.

Construção de um jogo eletrônico multiusuário em uma superfície de projeção multitoque / Pedro Guerra Brandão, Saulo Camarotti Rayol Braga. Brasília : UnB, 2009.
71 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2009.

1. Multitoque, 2. Jogos eletrônicos, 3. Interfaces naturais

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Construção de um jogo eletrônico multiusuário em uma superfície de projeção multitoque

Pedro Guerra Brandão
Saulo Camarotti Rayol Braga

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof.ª Dr.ª Carla Denise Castanho (Orientadora)
CIC/UnB

Prof. Dr. Ricardo Pezzuol Jacobi
IE/UnB

Prof. Dr. Díbio Leandro Borges
CIC/UnB

Prof.ª Dr.ª Carla Maria Chagas e Cavalcante Koike
Coordenadora do Bacharelado em Ciência da Computação

Brasília, 15 de julho de 2009

Agradecimentos

Agradecemos principalmente a motivação da professora Carla Castanho, por participar desta empreitada em uma área de estudo tão pouco entendida pelo nosso departamento. Agradecemos também os professores Ricardo Jacobi e Pedro Berger, por nos auxiliarem nas discussões e nos trazer novas oportunidades. Outra professora fundamental na nossa jornada foi a professora Suzete Venturelli, pois nos ajudou a traçar os primeiros passos e nos trouxe a visão do artista no processo de desenvolvimento de jogos.

Agradecemos nosso querido amigo João, que trilhou conosco essa aventura sempre nos trazendo para o que realmente era fundamental e importante. Por fim, um agradecimento especial a todos os nossos amigos, familiares e esposa que se dispuseram nos últimos anos para nos ajudar e nos trazer novas idéias.

Resumo

A evolução da complexidade tecnológica e a demanda cultural dos jogos eletrônicos levam ao desenvolvimento de novas tecnologias que inovam no conceito de jogabilidade. A pesquisa de interfaces homem-máquina são essenciais nesse papel de inovação. Entretanto, devido à complexidade dos sistemas computacionais de jogos eletrônicos, é necessário o estudo e a aplicação dos conceitos de engenharia de *software* dentro deste cenário específico, fundamentando-se nos padrões já estabelecidos, tais como arquitetura de sistema, camadas de abstração, reusabilidade de código, padrões de projeto e eficiência de algoritmos.

Este trabalho tem como principal objetivo desenvolver um jogo eletrônico, denominado *Eco Defense*, para uma superfície horizontal sensível a múltiplos toques. Desta forma, o trabalho discute um paradigma de interação mais natural entre jogador e o sistema utilizando o toque das mãos. Devido à ausência de soluções multitoque acessíveis no mercado, o trabalho envolve, também, a construção de uma mesa multitoque com dispositivos de baixo custo para ser utilizada como *input* pelo jogo.

Palavras-chave: Multitoque, Jogos eletrônicos, Interfaces naturais

Abstract

The evolution of technological complexity and cultural demands of electronic games leads to the development of innovative new technologies with regard to gameplay. The research of human-machine interfaces is essential to bringing about this innovation. However, due to the complexity of such computational systems, it's necessary to study and apply software engineering concepts within this scenario, underpinned by established standards, such as system architectures, abstraction layers, code reusability, design patterns, and algorithmic efficiency.

This project's main goal is to develop an electronic game, entitled Eco Defense, for a horizontal surface sensitive to multiple touches. Thus, this paper discusses a paradigm of interaction between the player and the system by means of finger touches. Due to the lack of affordable multitouch solutions in the market, this project also involves the construction of a multitouch surface using low-cost materials to be used as input for the game.

Keywords: Multitouch, Electronic games, Natural interfaces

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A tecnologia de hoje viabiliza novas interfaces homem-máquina. O estágio de desenvolvimento dos componentes de *hardware* dos computadores e a evolução dos algoritmos de processamento de imagens possibilitam o desenvolvimento de interfaces homem-máquina mais naturais e menos custosas. Os gestos e a fala sempre foram componentes importantes para a comunicação do homem. Logo, o estudo de interfaces naturais visa tornar intuitivas as interações, por exemplo através da fala, gestos, toques, e movimentos corporais. [7]

Para o homem, interagir com objetos em uma mesa é considerado bastante natural. Fazendo uma analogia ao computador, uma interface gráfica com a possibilidade de arrastar e manipular documentos e programas com os próprios dedos torna-se mais natural. Esta abordagem de interface multitoque, que permite diversos dedos simultaneamente, é mais intuitiva se comparada a um dispositivo *mouse* para organizar seus arquivos em uma área de trabalho, por exemplo.

Ampliando a discussão deste paradigma, uma interface multitoque pode ser também aplicada a uma superfície de trabalho, como uma mesa de um escritório. Assim objetos reais como papéis, canetas e livros são combinados com objetos virtuais de um sistema computacional. Desta maneira, a manipulação com objetos reais e virtuais se torna semelhante e mais natural. Além disso, o sistema pode interpretar os objetos reais sobre a mesa ampliando a gama de interatividade.

A naturalidade do toque também pode ser empregada em jogos eletrônicos. Sabe-se que, a manipulação de elementos e objetos em um jogo tradicional de tabuleiro torna o mesmo significativamente interessante e envolvente. Da mesma forma, a possibilidade de interagir com um jogo eletrônico através do toque em uma superfície, traz um novo conceito de jogabilidade e interatividade eletrônica.

É possível encontrar diversos trabalhos, [1, 4, 8–10], cujo foco é o estudo de interfaces de toque em sistemas multimídia interativos. Nestes trabalhos, jogos tradicionais como *Tangram* e *Game of Life*, por exemplo, foram desenvolvidos para superfícies multitoque com o objetivo de promover uma interação mais natural do jogador com o jogo.

Neste contexto, o objetivo deste trabalho é desenvolver um jogo eletrônico, denominado *Eco Defense*, para uma mesa multitoque. Ou seja, o mecanismo de input do jogo proposto

é uma superfície horizontal sensível a múltiplos toques, onde diversos usuários podem simultaneamente manipular objetos virtuais.

Um dos desafios que envolvem o emprego de interfaces naturais, como o toque por exemplo, é a obtenção do equipamento que viabiliza o projeto. Devido a ausência de soluções prontas que atendam os requisitos técnicos e econômicos, este trabalho envolve, também, a construção da mesa multitoque a ser utilizada pelo jogo proposto. Por razões de limitação orçamentária, optou-se pela utilização de peças facilmente encontradas no mercado local. Além disso, esta proposta emprega técnicas de visão computacional para o reconhecimento dos toques realizados sobre a mesa.

O desenvolvimento de um jogo eletrônico requer o entendimento e a aplicação dos conceitos de Engenharia de *Software* dentro deste cenário específico. O alto grau de interação e comunicação das entidades do jogo entre si e com o *hardware* influenciam na complexidade destes projetos. Dessa forma, o desenvolvimento do jogo proposto neste trabalho fundamenta-se nos padrões e conceitos de engenharia de *software* já estabelecidos, tais como arquitetura de sistema, camadas de abstração, reusabilidade de código e padrões de projeto.

Este documento esta organizado da seguinte forma. No capítulo 2, são apresentados trabalhos correlatos na área de jogos eletrônicos em superfícies multitoque. O capítulo 3 tem como foco os aspectos téoricos necessários sobre visão computacional aplicada ao conceito de toques e gestos. Em seguida, o capítulo 4 trata as características e problemas encontrados no uso de mesas multitoque. Para um entendimento aprofundado de arquitetura de *software* aplicada a jogos eletrônicos, aborda-se este tema no capítulo 5.

A partir do capítulo 6, este trabalho descreve o processo de design e implementação do jogo *Eco Defense*, bem como a construção da mesa multitoque, da seguinte maneira: o capítulo 6 apresenta o documento de *Game Design*, o capítulo 7 ilustra a estrutura física e de hardware do dispositivo de entrada multitoque, e o capítulo 8 detalha a implementação e a arquitetura do sistema interativo multimídia. Por fim, no capítulo 9, são descritas algumas conclusões e considerações a cerca de trabalhos futuros.

Capítulo 2

Trabalhos correlatos

Ao longo do presente capítulo, pretende-se oferecer ao leitor a oportunidade de adquirir conhecimentos sobre outros projetos relacionados a este trabalho. Inicialmente, serão expostas algumas pesquisas em interfaces naturais multitoque. Em seguida, estas interfaces serão trazidas ao contexto dos jogos eletrônicos atuais.

Em [1], Corso *et. al.*, da Universidade Johns Hopkins, dos Estados Unidos, apresentam um modelo de tela sensível ao toque que utiliza o paradigma VIC–*Visual Interaction Cue*. Este paradigma usa a dinâmica do espaço entre o usuário e seu computador para criar uma interação homem-máquina mais natural. O modelo de interação proposto discute o modo atual de manipulação de janelas, ícones, menus e ponteiro, e critica o fato do *mouse* se restringir apenas a um usuário.

Além disso, os autores afirmam que, pela limitação do número de comandos possíveis com um *mouse*, o usuário deve executar sequências complexas de movimentos para atingir um objetivo, muitas vezes, simples. O modelo proposto trabalha de forma a crescer a gama de movimentos de toque sobre a tela. Desta maneira é possível formar comandos complexos com a união entre poucos movimentos. O sistema utiliza duas câmeras para calcular a posição da mão no espaço X e Y da tela e a distância entre os dedos e a superfície (figura 2.1).

Outro projeto, desenvolvido na *Lulea University of Technology*, na Suíça, propõe a construção de uma mesa multitoque. Os resultados, publicados em [8], indicam que com o avanço da performance dos computadores pessoais e a disponibilidade de câmeras eficientes, é possível realizar processamento de imagens em tempo real para a construção de interfaces multitoque, e ainda preservar poder computacional para aplicações gráficas intensas, como jogos e interfaces de usuário atrativas.

A mesa proposta no projeto utiliza uma superfície de acrílico de 6 mm para refletir uma luz infravermelha, através de um fenômeno físico denominado reflexão interna total frustrada (FTIR—*Frustrated Total Internal Reflection*). Quando o usuário pressiona seu dedo contra a superfície de acrílico, a reflexão total da luz dentro da superfície é quebrada naquele ponto, iluminando o dedo. Desta forma, uma câmera que filma a superfície é capaz de detectar a luz infravermelha naquele ponto e assim um *software* pode processar a imagem a procura destes pontos acesos.

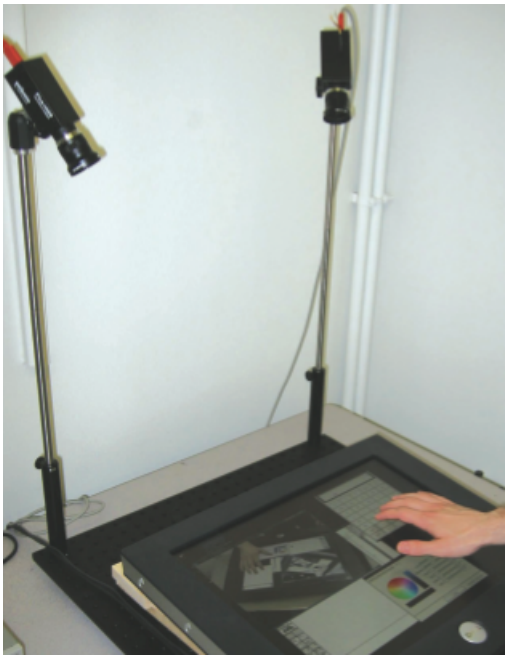


Figura 2.1: Trabalho com paradigma *Visual Interaction Cue*. (Imagem retirada de [1].)

Para permitir que a imagem projetada não influencie a imagem captada pela câmera, um filme fotográfico comum, que funciona como um filtro de luz visível, é colocado sobre a lente. Além disso, a câmera é configurada para captar imagens pequenas de 320x240 *pixels* a uma taxa de 60Hz, para que o tempo de resposta da mesa multitoque seja menor. O projeto utiliza um projetor comum de 1024x768 *pixels* de resolução para projetar a imagem gerada pelo computador na superfície de acrílico. E por fim, utiliza uma fonte ATX (*Advanced Technology Extended*) para acender duas trilhas de LEDs (*Light Emitting Diode*) infravermelhos dispostos em dois lados paralelos do acrílico. Estes LEDs são responsáveis por iluminar o acrílico através do fenômeno mencionado anteriormente, FTIR.

O *Music Technology Group*, na *Universitat Pompeu Fabra* em Barcelona, propôs uma aplicação de uma mesa multitoque que consiste em um instrumento musical inovador, denominado *reacTable* [11]. Seu principal componente é uma superfície tangível. Embora esta não seja uma interface movida a toque, a *reacTable* trabalha com marcadores fiduciais conhecidos pelo sistema (Figura 2.2).

Sua superfície redonda e translúcida é continuamente filmada por uma câmera que, por meio de um *software*, identifica a natureza, a posição e a orientação dos objetos que estejam em cima da mesa. Cada objeto possui uma impressão previamente conhecida, sendo denominados, portanto, marcadores fiduciais.

À medida que objetos são movimentados na superfície, ocorre um *feedback* sonoro e visual, de acordo com o tipo de fiducial. Um dado objeto pode ser responsável por gerar, modificar ou controlar um som. Certos tipos de objetos interagem entre si, sendo possível que a rotação ou a distância influencie seus efeitos.

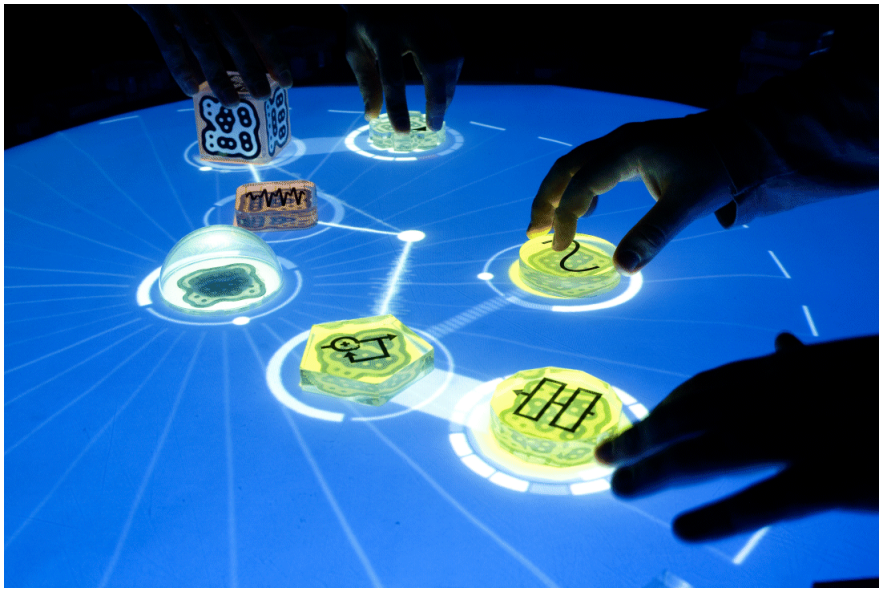


Figura 2.2: Marcadores fiduciais da *reacTable*. (Imagem retirada de [2].)

Dentre os objetivos da *reacTable* estão ser colaborativo, intuitivo, musicalmente desafiador, interessante, didático e possível de dominar, sendo adequado tanto para principiantes quanto para usuários profissionais, que podem utiliza-la em apresentações, shows ou concertos. Para fins de reconhecimento de imagem, foi criado para esse projeto o *software* *reacTIVision*, que se encontra licensiada sob a GPL (*General Public License*).

Estudantes do *California Institute of the Arts*, desenvolveram uma outra versão de mesa tangível e multitoque, que utilizou *reacTIVision* e foi denominada de *Brick* [12]. Esta mesa possui um funcionamento similar à *reacTable*, com superfície de acrílico iluminada por FTIR e reconhecimento de marcadores fiduciais. Embora a *Brick* também seja uma aplicação voltada para músicos, sua grande inovação foi denominada de "Previsão do Tempo". A aplicação recebe como *input* a previsão do tempo dos Estados Unidos e desenha na superfície de projeção um mapa com um gráfico da temperatura. Assim, os marcadores fiduciais que criam sons são influenciados pela temperatura do local onde o marcador se encontra. Os sons agudos são criados ao colocar os marcadores fiduciais sobre regiões do mapa que se encontram quentes, e os sons graves sobre regiões que a previsão do tempo marcou como frias.

Outro trabalho correlato, o *DiamondSpin* [9], é uma aplicação voltada para a interação multiusuário. Os autores, da Universidade de Stanford–EUA e da Universidade de Paris–França, mencionam que um dos desafios do estudo de superfícies multitoque e tangíveis é transformar a mesa em parte da interação homem a homem, de forma que as mesas não distoem do espaço em que habitamos ou trabalhamos. O *DiamondSpin* é um *toolkit* de desenvolvimento que trabalha com a rotação e escala de aplicações gráficas. Com esse *toolkit* pode-se desenvolver diversas formas de comportamento para as aplicações. Se vários usuários utilizam uma mesa multitoque simultaneamente, as aplicações se adequam à posição e orientação de cada usuário. No modo compartilhamento entre usuários, os objetos virtuais no centro da tela ficam maiores para que todos os usuários possam visualizar e interagir.

Além disso, é válido citar mesmo sem caracterização de trabalho científico, que diversos indivíduos e grupos em diferentes partes do mundo vêm construindo superfícies multitoque. Estes grupos trocam suas experiências em comunidades *online*, tal como *Natural User Interface Group* [13], voltadas especificamente para tratar desses assuntos. Esta colaboração *online* contribui, hoje, para encontrar formas de construir estes equipamentos a baxíssimos custos. É possível encontrar, inclusive, soluções multitoque construídas com porta-retratos, caixas de papelão e uma câmera USB de 10 dólares. Em grande parte destes trabalhos são incorporados jogos eletrônicos, tanto jogos já existentes, como novos jogos desenvolvidos especificamente para mesas multitoque.

Uma das principais formas de interação homem-máquina da atualidade é a utilizada em jogos eletrônicos. Hoje, diversos componentes fazem parte do computador ou *console* a fim de maximizar a performance dos jogos eletrônicos e melhorar sua jogabilidade. Um elemento importante no avanço da jogabilidade destes dispositivos é a inovação nos mecanismos de input. Diversos trabalhos de pesquisa e produtos comerciais [14, 15] inovam neste conceito, com multitoque, acelerômetros e câmeras de vídeo. De acordo com pesquisadores da *Brown University* [14], muitos dos jogadores se surpreenderam com a facilidade de aprendizado e naturalidade destes novos mecanismos de interação.

No âmbito da abordagem multitoque, que é o foco desta monografia, o *Apple iPhone* e o *iPod Touch* [15], recém lançados no mercado, trazem a inovação de multitoque em dispositivos móveis. Estes dispositivos incluem como principais aplicações os jogos eletrônicos incorporando conceitos novos de jogabilidade, já que contam com interface multitoque e acelerômetros.

Um recente lançamento da *Illusion Labs* [3], empresa Sueca de desenvolvimento de jogos para dispositivos móveis, para *iPhone* é o *Touch Grind*. Este jogo simula um *skate* e o jogador deve manter dois dedos sobre a tela do *iPhone* como se fossem as pernas do skatista, e assim pode realizar truques e manobras sobre a prancha (figura 2.3). O jogo interpreta o movimento dos dedos sobre a tela e o incorpora no *gameplay* do jogo. Além deste, diversos outros jogos estão sendo lançados para estas plataformas.

Figura 2.3: Jogo para *iPhone*: *Touch Grind*, da *Illusion Labs*. (Imagem retirada de [3].)

Com relação a jogos para mesas multitoque, também é possivel encontrar diversos trabalhos na área. Estudantes do Centro de aplicações em Realidade Virtual da *Iowa State University*, nos Estados Unidos, desenvolveram uma mesa multitoque e um jogo multijogador interativo no âmbito do projeto denominado *Sparsh UI* [4]. Trata-se de uma adaptação de um outro jogo em que dois times, situados em lados opostos da mesa, devem simultaneamente, através de toques sobre a superfície, atacar as bases do time oposto ou defender suas próprias instalações virtuais (Figura 2.4). Este tipo de trabalho põe em prática a discussão sobre multiusuários em superfícies multitoque, trazendo um novo paradigma de interação com as tecnologias.

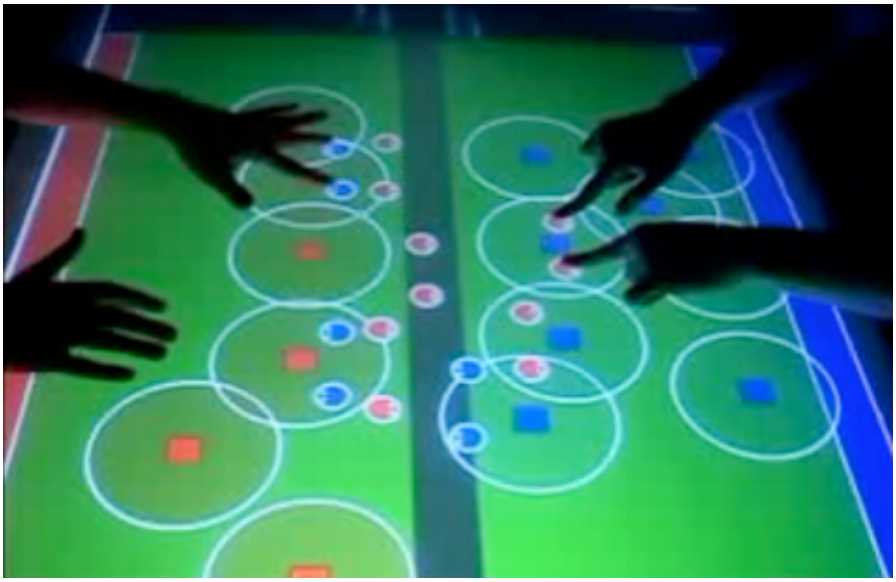


Figura 2.4: Jogo para uma mesa multitoque do projeto *Sparsh UI*. (Imagem retirada de [4].)

Um jogo eletrônico tradicional, denominado Jogo da Vida e criado em 1970 por John Conway [16], foi recriado para uma mesa multitoque incorporando a jogabilidade com diversos jogadores. A superfície é dividida em diversos organismos unicelulares que morrem e nascem dependendo de sua vizinhança. Quando um jogador toca a tela, um novo organismo é criado. Assim pode-se alterar, em tempo real, a vida dos organismos em volta daquela nova célula (Figura 2.5). O projeto que recriou este jogo, denominado *The Verve Project* [10], é sediado nos Estados Unidos e tem como principal objetivo estudar mecanismos de inteligência artificial com redes neurais.

Ainda na linha de jogos lúdicos e desafios matemáticos, um projeto indiano adaptou o jogo *Tangram* para uma mesa multitoque [17]. O *Tangran*, tradicionalmente, é um jogo chinês de quebra-cabeças com sete peças triangulares e retangulares que podem ser posicionadas de maneira a formar diversas figuras. No jogo adaptado pelos pesquisadores J. Divesh e A. Adithya, o usuário pode escolher umas das figuras que deseja construir e move as peças para as posições corretas. O jogo permite a utilização de gestos como arrastar e rotacionar.

Os trabalhos e projetos descritos neste capítulo demonstram o direcionamento de esforços no desenvolvimento de jogos eletrônicos que incluam uma interação mais natural

Figura 2.5: *The Verve Project: Game of Life* em uma superfície multitoque. (Imagem retirada de [5].)

e envolvente para o usuário. Centros de pesquisa e empresas do ramo tecnológico trabalham constantemente no aprimoramento destas interfaces e tecnologias, vislumbrando e atingindo novos patamares na área.

Capítulo 3

Visão computacional aplicada ao processamento de toques

Estudar e pesquisar o uso de jogos em mesas mutitoque demanda conhecimentos a respeito deste novo paradigma, em particular as técnicas voltadas para reconhecimento de toques através da captura de imagens. De modo a apresentar conceitos fundamentais para a proposta deste trabalho, este capítulo tem como objetivo específico abordar o tópico processamento de imagens através do campo de estudo denominado visão computacional, e apresentar as dificuldades encontradas por outros projetos semelhantes.

3.1 Visão computacional

Visão computacional é o ramo da ciência e da tecnologia que se refere a máquinas dotadas de visão artificial, buscando simular a visão biológica. Para tal, tem-se como objetivo principal a extração de informação de imagens. [18] Estas podem ser imagens estáticas (como fotografias), ou seqüências de imagens (como vídeos). As imagens podem ser obtidas de diversas fontes e estas processadas por meio de algoritmos. As informações obtidas podem ser utilizadas no intuito de dotar as máquinas de habilidades tais como a detecção de movimentos, controle de processos industriais, organização de informações, interação homem-máquina mais natural e reconhecimento de toques. o qual é foco deste trabalho.

Estudos e pesquisas nesta área vêm ocorrendo de forma intensa desde 1968, de acordo com [18], quando o poder computacional se tornou suficiente para o processamento de grandes quantidades de dados, tal como são representações digitais de imagens. A visão computacional surgiu indiretamente como necessidade de outras áreas, até que eventualmente consagrou-se como uma área de pesquisa distinta. Decorre deste fato que não existe uma formulação específica do problema de visão computacional, já que possui diferentes *backgrounds*, dependendo da área de aplicação em questão. Logo, não existe uma forma específica de resolver seus problemas, sendo que em cada problema emprega-se uma abordagem específica, frequentemente não generalizada a outros contextos.

A utilização das técnicas de visão computacional no trabalho apresentado nesta monografia, advém da necessidade de extrair informações de imagens que representam a superfície com a qual o usuário interage. A forma como essas imagens são obtidas serão explicadas adiante no capítulo 7. No momento, basta saber que são semelhantes à figura 3.1(a). Na seção que se segue, serão vistos os passos pelo qual a imagem passa até que esteja em condições de serem extraídos os pontos de referência da mesma, isto é, os pontos onde o usuário tocou a superfície.

3.1.1 Reconhecimento de toques em uma superfície

Uma das soluções existentes para reconhecimento de toques em uma superfície, utiliza um computador dotado de visão computacional com a habilidade de interpretar o posicionamento dos dedos. Uma câmera conectada ao computador captura imagens dos dedos dos usuários sobre a superfície multitoque. Através de algoritmos de processamento de imagens, o posicionamento de cada dedo e seus movimentos são calculados e transmitidos para a aplicação.

Processamento de imagem

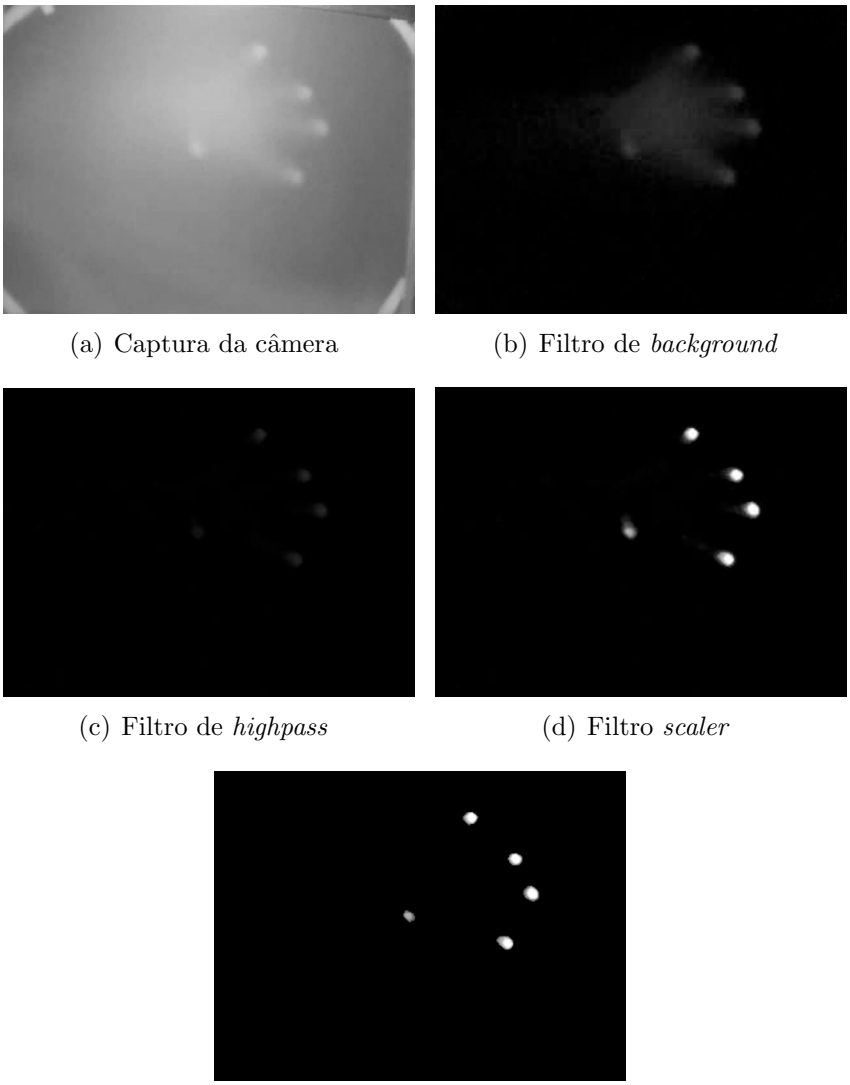
O processamento das imagens se dá em algumas etapas subsequêntes, dentre as quais, em sequência: captura da imagem através de um dispositivo de entrada, tratamento da imagem com filtros e finalização com a detecção de conjuntos de pixels que possuem uma intensidade de luz semelhante, denominados *blobs*.

Os filtros a serem aplicados variam de acordo com a técnica de iluminação utilizada. De acordo com Muller [6], imagens capturadas em contextos com iluminação FTIR (*Frustrated Total Internal Reflection*) tipicamente produzem imagens com menos chiado e maior contraste, conseqüentemente sua cadeia de processamento é mais simples. Já outra técnica de construção mais simplificada, denominada de DI (*Diffuse Illumination*), o processo de filtragem da imagem da câmera é mais complexo, como são descritos abaixo.

1. Capturar a imagem do dispositivo, como mostra a figura 3.1(a). Esta imagem deverá ser composta de apenas um canal de 8-bits, portanto caso necessário deverá ser convertida.
2. Remover *background* da imagem. *Background*, neste contexto, se refere àquilo que não seja o movimento das mãos do usuário (ou qualquer outro objeto posto na superfície). Sendo assim, é armazenado um *frame* de referência, que consiste na imagem da mesa sem qualquer interação. Este frame será subtraído dos subsequentes para filtrar o *background*. O resultado deste filtro pode ser visto na figura 3.1(b).
3. Aplicar um filtro *high-pass*, permitindo que *pixels* acima de um certo patamar (*threshold*) passem inalterados, enquanto *pixels* abaixo serão atenuados. Este valor é ajustado manualmente de acordo com as condições de iluminação da superfície. O objetivo é "selecionar" apenas os *pixels* que sejam ponto de contato dos dedos do usuário com a superfície, como exemplificado na figura 3.1(c).

4. Aumentar o brilho das manchas com baixa visibilidade, através de um filtro *scaler*. Figura 3.1(d).
5. Por fim, aplicar o filtro *rectify* para reduzir a quantidade de chiado e tornar os *blobs* mais definidos para que sejam identificados, como na figura 3.1(e).

A imagem 3.1(a) é a captura simples de uma câmera monocromática. Ao final do tratamento, temos um resultado semelhante à Figura 3.1(e).



(a) Captura da câmera

(b) Filtro de *background*

(c) Filtro de *highpass*

(d) Filtro *scaler*

(e) Filtro *rectify*

Figura 3.1: Processamento de imagem capturada da câmera. (Imagem retirada de [6].)

Blob-detection e *Blob-tracking*

Blob-detection é o processo de detectar toques em imagens resultantes dos procedimentos da seção anterior. Já *blob-tracking* é rastreamento de *blobs*, ou seja, é a análise destes elementos ao longo de uma seqüência imagens. Estas duas análises, conjuntamente, irão

permitir detectar e definir o movimento de cada *blob* ao longo de sua vida útil, bem como definir quando deverá o ocorrer a criação de um novo *blob*.

O processo de detecção de *blobs* é realizado a cada *frame*. Com base na imagem previamente processada, são extraídos os contornos dos pontos de contato do dedo com a superfície. Com base nesses contornos, deve se definir se esses correspondem a marcadores fiduciais ou toques de dedos, baseado em sua forma e tamanho. Se as suas extremidades formarem ângulos de aproximadamente 90 graus, assume-se que corresponde a um fiducial. Caso contrário, assume-se que é um toque, e uma elipse será inscrita no contorno. Com base nessa elipse, podem ser extraídas informações de posição, orientação, e tamanho do *blob*. Caso sua altura e largura estejam dentro de uma faixa pré-definida, o *blob* será adicionado a uma lista. Com isso conclui-se o procedimento de detecção de *blobs*.

Após detectados os *blobs*, o rastreamento destes deve ser feito em relação ao *frame* anterior para determinar se blobs devem ser criados, removidos, ou atualizados. Para fazer tal análise, são usados dois conjuntos. O primeiro, P, deverá conter a lista de pontos referentes a *blobs* do frame anterior. A título de exemplo (imagens 3.2 e 3.3):

$p_1 : (1, 1), p_2 : (4, 4) \Rightarrow |P| = 2$

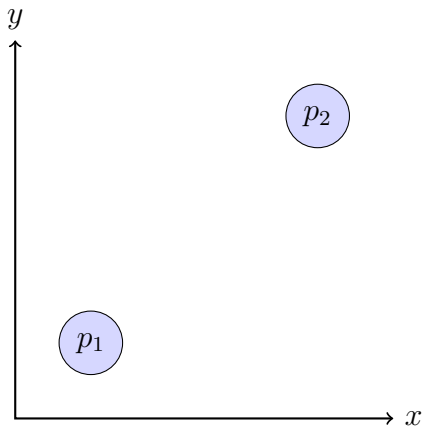


Figura 3.2: Gráfico de P.

O segundo conjunto, denominado Q, deverá conter a lista de *blobs* do frame atual:

$q_1 : (1, 2), q_2 : (1, 4), q_3 : (4, 3) \Rightarrow |Q| = 3$

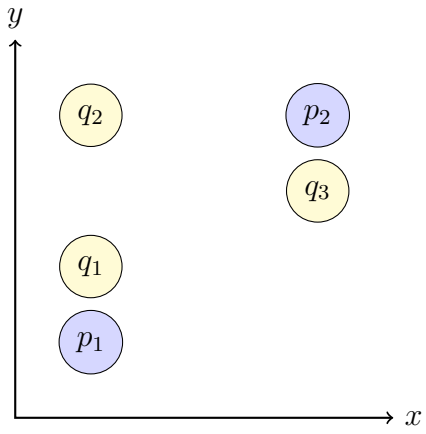


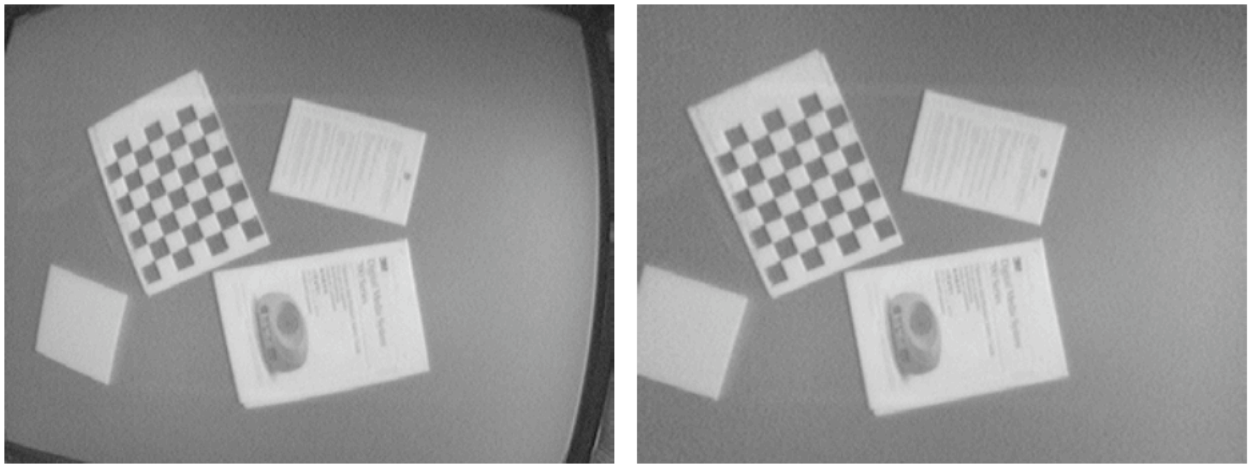
Figura 3.3: Gráfico de Q.

O número de elementos de Q é maior que de P, indicando que houve o acréscimo de pelo menos um *blob*. Será necessário calcular as menores distâncias e determinar qual *blob* foi criado. A conclusão será que o *blob* em p_1 migrou para q_1, o *blob* em p_2 migrou para q_3, e q_2 é um novo *blob*. A cada frame, o conjunto Q do *frame* anterior passará a ser o P para que o processamento possa ser reiniciado.

Distorção da Câmera

Uma imagem obtida por uma câmera posicionada sob uma superfície plana possui uma distorção natural. Os pontos deste plano distantes à câmera são representados de forma reduzida e os pontos centrais do plano são ampliados, como mostrado na imagem (a) 3.4. Para um melhor resultado da detecção de *blobs*, a imagem é corrigida através de algoritmos. Diversas bibliotecas de processamento de imagem e detecção de multitoque realizam este tratamento, de acordo com [6].

Para a correção da distorção é necessário que seja feito o mapeamento dos pontos da imagem a um ponto no espaço, como proposto por Muller [6]. Para tal, é necessário considerar a distância focal da câmera, o centro da imagem, o tamanho de cada *pixel* e o coeficiente de distorção radial da lente, fatores estes intrínsicos à câmera. Além disso, há fatores extrínsecos à câmera, como vetor de translação e matriz de rotação que relacionam a câmera ao espaço.



(a) Imagem da câmera distorcida. (b) Imagem da câmera corrigida

Figura 3.4: Correção da distorção focal da câmera. (Imagem retirada de [6].)

Para uma câmera pontual, um ponto M no espaço corresponde à seguinte matriz de transformação m na imagem.

$m = A[Rt]M$

A matriz A denota os favores intrínsecos à câmera e é definida da seguinte forma:

$$A = \begin{vmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{vmatrix}$$

onde c_x e c_y correspondem ao centro da imagem, e f_x e f_y à distância focal da câmera. Por fim os parâmetros R e t equivalem respectivamente à matriz de rotação da câmera e ao vetor de translação.

3.1.2 Biblioteca *Touchlib* e Protocolo TUIO

Uma biblioteca que pode ser utilizada para o processamento das imagens da câmera é a *Touchlib* [19]. Foi desenvolvida especificamente para a criação de superfícies com interação multitoque. Seu funcionamento consiste na aplicação de filtros de forma customizável, da mesma forma que foi visto ao longo desta seção. Para a execução destes filtros, a *Touchlib* faz uso extensivo da *OpenCV* (*Open Source Computer Vision Library*), uma biblioteca multiplataforma para desenvolvimento de aplicativos de processamento de imagens.

A Touchlib atua de acordo com o paradigma servidor–cliente, sendo que esta faz o papel de servidor e a aplicação faz o papel do cliente. A comunicação com este *software* é realizada usando o protocolo TUIO (*Tangible User Interface System* [20]), que foi desenvolvido especificamente para a transmissão do estado de objetos tangíveis e eventos de toque em uma superfície. Ou seja, permite uma arquitetura distribuída, possibilitando que o aplicativo seja executado separadamente da aplicação.

Este protocolo permite que interfaces, tal como a *reacTable*, apresentada no capitulo 2, utilizem um protocolo comum, mesmo sendo desenvolvidas separadamente. O TUIO atualmente implementa a transmissão de marcadores fiduciais, mas pode também ser acrescido de novas funcionalidades através de componentes. A implementação do protocolo é feita por meio do protocolo *OpenSound Control*. Este opera sobre a camada UDP (*User Datagram Protocol*) de transporte, permitindo a divisão de aplicativos em mais de um computador.

O protocolo opera com três tipos de mensagem: *set*, *alive* e *fseq*. Mensagens *set* são usadas para informar sobre o estado atual de um objeto, tal como posição e orientação. Cada objeto recebe um ID único. Mensagens *alive* são usadas para indicar o conjunto de objetos presentes na superfície através de seus IDs. Não existem existem mensagens explícitas de adição e remoção de objetos, sendo estas inferidas das mensagens *set* e *alive*. Além disso, há as mensagens *fseq*, que precedem cada passo de atualização e incidam um *frame ID*, indicando que todas as mensagens que seguem se referem àquele *frame ID*.

Devido à natureza da camada UDP, existe a possibilidade que pacotes sejam perdidos em trânsito. Sendo assim, caso um pacote seja perdido, o TUIO inclui informações adicionais de redundância para corrigir eventuais erros na transmissão. Além disso, o estado de um objeto, mesmo que inalterado, será reenviado periodicamente em uma mensagem *set*.

O protocolo especifica diversos parâmetros para suportar vários paradigmas de *input*, e portanto, os parâmetros variam de aplicação para aplicação. No caso da Touchlib, são enviados os parâmetros apresentados na Tabela 3.1 abaixo.

`s`	sessionID, ID temporário do objeto, int32
`x`, `y`	posição, float32, range 0...1
`X`, `Y`	vetor de movimento (velocidade de movimento e direção), float32
`m`	aceleração de movimento, float32
`w`, `h`	largura e altura do *blob*, float32

Tabela 3.1: Parâmetros de mensagens TUIO em superfícies interativas 2D

Capítulo 4

Aspectos de mesas multitoque

As superfícies multitoque convidam os pesquisadores e a indústria a repensar na forma de interação com os computadores. É necessário rediscutir as funções disponíveis para os usuários, pois os computadores foram construídos para uma relação pessoal, entre o usuário e sua tela de trabalho.

Deve-se levar em conta que, historicamente, as mesas são locais de encontro e de trabalho, e hoje fazem parte do cotidiano de todos. Por isso, o uso destas mesas digitais se tornam mais intuitivas e distintas de outras interfaces. Entretento, ao mesmo tempo não se adequam às aplicações dos *software*s atuais.

A fim de que mesas multitoques se tornem objetos comuns, existem diversos problemas que devem ser tratados para que a eficiência se torne compatível com as necessidades atuais dos usuários. Dentre esses problemas, neste capítulo, serão discutidos os relacionados com hardware, tamanho das mesas, múltiplos toques e diversos usuários simultâneos.

De acordo com Scott *et al.* em [21], mesas digitais de toque são estudadas há mais de 20 anos, e o principal problema encontrado é o fato de que em cada novo projeto de uma mesa, era necessário reconstruir todo o aparato de *hardware* e *software*. Estas mesas variavam em suas tecnologias base, como aparência, compatibilidade, eficiência de *hardware*, tamanho, tipo de captura do *input* e projeção da imagem. Muitas vezes eram desenvolvidos sistemas caros e específicos de projeção e captura do *input*.

Atualmente, este cenário mudou, uma vez que diversos projetos abriram suas pesquisas publicamente, como é o caso do *DiamonSpin*, citado anteriormente neste trabalho. Ainda assim, existem diversos projetos na busca por combinações de *hardware* ou algoritmos avançados para encontrar uma mesa ideal. De acordo com Stacey [21], é seguro dizer que não existe uma configuração compatível com as necessidades atuais e que satisfaça requisitos comuns. Esforços conjuntos e pesquisas mais aprofundadas ainda se fazem essenciais para transpor as dificuldades existentes na viabilização de superfícies, ou mesas, multitoque.

4.1 Tamanho e Resolução

As superfícies multitoque trazem um problema para os usuários no que diz respeito ao seu tamanho. Uma mesa pode, por exemplo, ser grande o bastante para que dezenas de usuários interajam com suas aplicações, mas pode também ser pequena de modo que apenas duas ou três pessoas possam ter acesso a mesma. Tais fatores tem influência direta tanto no modo de interação dos usuários com a mesa, como nas aplicações a serem executadas, uma vez que estas devem estar adequadas às características de cada mesa a fim de prover diferentes recursos e ações aos usuários.

Em uma mesa com uma área extensa, é possível que uma aplicação não esteja, literalmente, ao alcance do usuário. Este problema pode ser solucionado por intermédio de outro tipo de *input*, como a técnica *TractorBeam* [21], que utiliza-se de uma caneta *stylus* para que o usuário possa apontar para uma aplicação do outro lado da mesa e trazê-la mais perto de si. Outra possibilidade é solucionar o alcance via *software*, onde um usuário pode, através de uma interface gráfica, trazer qualquer aplicação aberta para perto de si.

Estas mesas são, em sua maioria, maiores que os monitores de computadores pessoais. Logo possuem um problema relacionado à resolução da imagem projetada. É comum o uso de projetores XVGA (*Extended Video Graphics Array*) para projetar a imagem na superfície. No entanto, os projetores, em sua maioria, possuem resolução padrão de 1024 x 768 *pixels*, uma resolução considerada pequena para uma superfície extensa. Outras mesas foram construídas utilizando-se de dezenas de monitores LCDs com resolução alta, mas além do custo alto, possuem margens de cerca de 1,5 cm entre cada monitor [21].

4.2 Interfaces de usuário

De acordo com [22], interfaces sensíveis ao toque trazem ao usuário a possibilidade de interagir diretamente com os dados. Isto traz um grande apelo ao usuário, pois além de ser natural, é de fácil aprendizado. Por não possuir partes móveis, é interessante para ser usado, também, em equipamentos acessíveis ao público, tal como caixas automáticos e pontos de acesso.

O paradigma de interação com o computador usado atualmente usa elementos muito pequenos que devem ser clicados com o *mouse*. Devido à precisão inerente do dispositivo, a seleção com precisão de poucos *pixels* é relativamente fácil. Em uma interface baseada no toque, não se pode dizer o mesmo. Em diversas mesas multitoque, a câmera, que captura a imagem a ser processada, é o fator determinante da precisão da interface. Ou seja, diversos aspectos como luz, distância focal e resolução alteram a usabilidade para o usuário.

Um sensor de mais alta qualidade reduz a granularidade do *input*, reduzindo a taxa de erros, aumentando a resolução e permitindo uma taxa de amostragem mais alta. No entanto a falta de precisão não se refere somente ao equipamento, pois como o dedo possui uma grande área, relativamente grande em termo equivalentes a *pixels*, ocasionalmente haverá ambiguidade em relação à área que o usuário realmente pretende tocar. Esto é um problema que deve ser considerado durante a construção de tais interfaces gráficas.

Além dos problemas encontrados na construção de mesas multitoque citados acima, é relevante considerar uma discussão a respeito das interfaces de usuário. Uma superfície multitoque tem o propósito de prover interação de diversos usuários com um computador como se todos utilizassem uma mesma mesa de reuniões, por exemplo. Ou seja, normalmente cada usuário possui seus documentos e objetos organizados de forma que fiquem de frente para o próprio usuário e facilite sua visualização e leitura. Da mesma forma, aplicações e documentos virtuais em uma mesa multitoque devem trazer esta experiência ao usuário, bem como facilitar a leitura de documentos entre os mesmos.

Se diversos usuários utilizam uma mesa, a experiência de interação pode ser diferente para cada um e a interface pode mostrar informações contraditórias. Um botão tradicional, utilizado em sistemas operacionais atuais, pode parecer não pressionado se visualizado de um ângulo, e ao mesmo tempo pode parecer pressionado se visualizado de cabeça para baixo. Veja a diferença nas figuras 4.1(a) e 4.1(b).

(a) (b)

Figura 4.1: Um mesmo botão visto de ângulos diferentes.

Ao construir interfaces gráficas em superfície deve-se considerar, então, a relativização do ponto de vista do usuário, permitindo a rotação de aplicações e documentos abertos. Como mencionado anteriormente, a *DiamondSpin* já rotaciona automaticamente a aplicação de forma que fique de frente para a lateral da mesa mais próxima da janela da aplicação. Assim um usuário possui todas as aplicações organizadas corretamente para si mesmo, e se uma destas janelas for aproximada para a outra extremidade, a aplicação rotaciona automaticamente e outros usuários podem visualizá-la de forma correta. Assim, um usuário pode mostrar para um outro usuário uma aplicação, apenas arrastando sua janela para a extremidade próxima ao outro.

4.3 Multi-usuário

A grande dificuldade de promover o uso de uma mesa multitoque entre diversos usuários é manter o registro e identidade de cada toque relacionado ao usuário relativo.

Este problema é facilmente ilustrado pela seguinte situação: Dois usuários interagem com uma aplicação de desenho em uma superfície multitoque. Um dos usuários decide mudar a cor do seu pincel de amarelo para azul. Então, surge um problema. Como o *software* da mesa identificará que aquele toque recebido pelo *hardware* é do usuário com pincel amarelo? Neste caso, uma solução simples é tornar todos os pincéis azuis, sem levar em conta a opção dos outros usuários.

Esta ilustração mostra que, para certas aplicações, diferentes decisões deverão ser tomadas. Em alguns casos, a aplicação tornará comum a todos os usuários a última opção selecionada, ou em outros casos, diferentes áreas de trabalho poderiam ser delimitadas

para cada usuário, cada qual com sua respectiva opção de ferramenta. No caso de desenho exemplificado, a o emprego desta abordagem se daria através da divisão da área de desenho em partes separadas para cada usuário.

4.4 Gestos em multitoque

Tradicionalmente, interfaces com um único ponto de contato não utilizam gestos. Embora alguns *softwares* que trabalham com mouse incorporaram gestos em sua interface, nunca se viu uso generalizado deste paradigma, devido ao fato de tais gestos não serem intuitivos com apenas um ponto de contato.[1]

Por outro lado, interfaces multitoque, como citado em [23], abrem uma gama de possibilidades, pois possuem poucas restrições quanto ao número de pontos de contato simultâneos. Sendo assim, é necessário acoplar nos *softwares* mecanismos para o reconhecimento de tais gestos.

Esta abstração de gestos se torna necessária na medida em que é possível ampliar e acoplar ao sistema novas entidades. É importante, também, que seja possível adaptar este sistema a outras interfaces multitoque sem requerer grandes modificações.

Dentre os gestos reconhecidos e tradicionais, podemos citar os seguintes no contexto de mesas multitoque:

Click

Este é o mais simples dos gestos. Consiste em um único ponto de contato. O gesto é acionado ao inserir o dedo sobre a superfície e retirá-lo logo em seguida. Para o *software*, o dedo inserido e retirado no mesmo ponto é considerado um *click*.

Drag

Uma ampliação do gesto anterior, *Drag* é um gesto que interpreta a movimentação de um único ponto de *input*. É um simples processo de arrastar. O *software* interpreta o gesto arrastar até que o dedo seja retirado da superfície de projeção.

Rotate

O gesto *rotate* é executado a partir de dois pontos de contato. Isto é, são necessários dois dedos. O usuário deve inserir os dois dedos sobre a superfície e movimenta-los em direções opostas mantendo a distância entre os dedos constante. A cada passo de atualização, o *software* compara o ângulo entre dois dedos nos dois momentos distintos. A diferença entre os dois ângulos denota a rotação provocada pelo gesto.

[1] Um *software* que adotou este mecanismo nativamente é o navegador Opera. Os navegadores Firefox e Internet Explorer possuem *add-ons* para tal finalidade.

Scale

O *Scale* é similar ao *Rotate*, no entanto em vez de comparar as diferenças em ângulos, compara-se a distância entre os dois dedos. A aproximação indica uma redução e o afastamento entre os dedos indica uma ampliação.

Rotate e *Scale* podem ser considerados um mesmo gesto para o *software*, pois os dois podem ocorrer simultaneamente e possuem etapas de interpretação semelhantes. Enquanto o usuário executa um gesto de *Rotate*, pode executar, com os mesmos dois dedos, um gesto de *Scale*, e vice-versa.

Capítulo 5

Arquitetura de *software* aplicada a jogos eletrônicos

Um jogo eletrônico é um sistema computacional, cujo processo de construção é complexo, e portanto demanda a aplicação dos conceitos e técnicas de engenharia de *software*. Este capítulo visa contextualizar esta área de estudo aplicada à construção de jogos eletrônicos.

5.1 Arquitetura de jogos eletrônicos

Atualmente, os jogos estão atingindo um alto grau de sofisticação e, como consequência, os *softwares* necessários para sua implementação têm se tornado cada vez mais complexos. As equipes de programação de jogos estão se tornando maiores e seus membros realizam tarefas cada vez mais especializadas. Além disso, de acordo com Doherty [24], a área de desenvolvimento de jogos está crescendo e amadurecendo ao ponto que as expectativas são similares às outras áreas de desenvolvimento de software. Assim, é essencial que jogos tenham uma arquitetura bem-definida para auxiliar, ou até mesmo viabilizar, o processo de desenvolvimento, manutenção e eventual evolução.

Inicialmente é necessário fazer a distinção entre código da *engine* e código específico do jogo. O primeiro é a fundação sobre a qual o jogo é construído, sem se ater ao conceito específico do jogo sendo desenvolvido, com a utilização de conceitos genéricos. Ou seja, são criadas camadas de abstração para o hardware e interfaces para controladores genéricos de recursos e eventos. Já o código específico do jogo é construído com lógica de comportamento específico das entidades e das regras do jogo. Para a produção de outros jogos, na maioria dos casos, este código específico deve ser descartado, enquanto a *engine* pode ser reutilizada.

Um aspecto importante no que se refere a arquiteturas de *software* é o acoplamento. Esta é uma medida qualitativa das dependências entre duas partes do código. Um acoplamento forte significa que duas partes de código dependem fortemente uma da outra. Isto reduz a possibilidade de reuso, além de tornar as modificações neste código um processo custoso. Forte acoplamento entre entidades de domínios diferentes é um fator a ser evitado por qualquer sistema.

A arquitetura de um jogo deve contemplar a execução de diversas tarefas, tais como receber *input* do jogador, verificar colisão entre objetos, atualizar entidades do jogo, renderizar texturas e executar simulações. Embora tratem de aspectos diferentes do jogo, muitas destas tarefas devem ser executadas a cada *loop* iterativo. Assim, pelas razões mencionadas acima, a arquitetura de *software* do jogo deve ser projetada de modo a evitar o forte acoplamento entre estas tarefas distintas. Na seção a seguir serão apresentados alguns padrões que viabilizam a elaboração deste tipo de arquitetura.

5.2 *Design patterns*

Design patterns (padrões de projeto) é o termo usado para designar padrões de projeto recorrentes em sistemas orientado a objetos.[1] O objetivo destes padrões é aumentar a flexibilidade e proporcionar o reuso de código se aplicados de forma adequada. Podem, também, facilitar a evolução do sistema ao criar independência entre os módulos do *software*. Devido à sua complexa natureza, decisões feitas na arquitetura de um jogo podem ser cruciais para maximizar a eficiência do desenvolvimento.

Em 1995, Gamma *et al.*[26] criaram um catálogo de diversos padrões. Este estudo permanece como referência no assunto até hoje. Dentre estes padrões, alguns possuem maior relevância para o contexto de desenvolvimento de jogos eletrônicos. É oportuno mencionar que, como mencionado em [27], *design patterns* se aplicam tanto ao ramo de programação de *softwares* interativos quanto à concepção dos requisitos, área de estudo denominada de *game design*.

Para cada padrão, são estabelecidos os seguintes aspectos:

- O **nome** usado para descrever, de maneira sucinta e precisa em no máximo duas palavras, o problema, a solução e as conseqüências inerentes do padrão.
- O **problema** e o contexto adequados para a aplicação do padrão.
- A **solução** descreve os elementos que compõem o padrão. Cada solução inclui relacionamentos entre classes e a colaboração entre estas caso existam.
- As **consequências** de se aplicar o padrão, ou seja, os resultados e as desvantagens.

Devido à natureza modular da estrutura de um jogo, estes padrões podem ser empregados com extensivamente com êxito. O entendimento dos padrões é essencial para formular uma arquitetura adequada. Portanto, a seguir serão descritos alguns padrões que se aplicam ao domínio de desenvolvimento de jogos.

[1] O termo *design patterns*, bem como a noção de padrões, não se restringe ao domínio da computação, sendo anteriormente identificado por Christopher Alexander quando propôs soluções para problemas arquiteturais. [25]

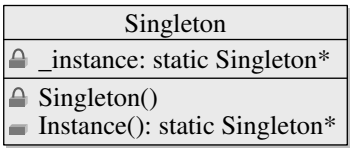


Figura 5.1: Padrão *Singleton*

5.2.1 *Singleton*

"Assegurar que uma classe possui apenas uma instância, e fornecer um ponto de acesso global a ela." [26]

Em certas ocasiões, é importante que exista apenas uma instância de determinada classe. Se, além disso, a classe for usada de maneira global, têm se um cenário possível de uso do padrão *Singleton*.

O objetivo deste padrão é tornar a própria classe responsável por sua existência, assegurando que exista uma única instância e interceptando a criação de novas instâncias.

Uma implementação ingênua deste padrão envolve armazenar a única instância em uma variável global. Esta solução, no entanto, não é adequada pois permite a instanciação de vários objetos, além de desnecessariamente poluir o *namespace* global com uma variável de instância.

Uma correta implementação deste padrão (figura 5.1) é definir uma operação de classe (por exemplo `Instance()`) que retorne a única instância, a ser salvo como ponteiro em um membro estático da própria classe. Além disso, o construtor deve ser privado de forma a impedir a instanciação externa à classe.

Em jogos eletrônicos, é comum o uso do padrão *Singleton* em controladoras de eventos, que gerenciam mensagens por todo o sistema, e controladoras de recursos, que gerenciam o carregamento de recursos audiovisuais por todo o jogo. Em ambos os casos, apenas uma instância de cada classe é necessária e devem permitir acesso global.

5.2.2 *Observer*

"Definir uma dependência um-para-vários entre objetos de tal forma que a alteração do estado de um objeto automaticamente notifica seus dependentes." [26]

Neste padrão (figura 5.2), um objeto, denominado `subject`, mantém uma lista de observadores e os notifica de mudanças de estado. Cada `subject` pode ter diversos observadores dependentes e todos são notificados quanto a mudança de estado. Ao recebimento da notificação, cada observador irá consultar o `subject` para sincronizar seu dados. O `subject` não necessita ter conhecimentos específicos dos observadores.

A classe `subject` é responsável por, além de ter conhecimento de todos os observadores, fornecer uma interface para a adição e remoção de observadores.

O observador, por sua vez, define uma interface para objetos que devem ser notificados de alterações no `subject`. Esta interface é a única característica que todos os observa-

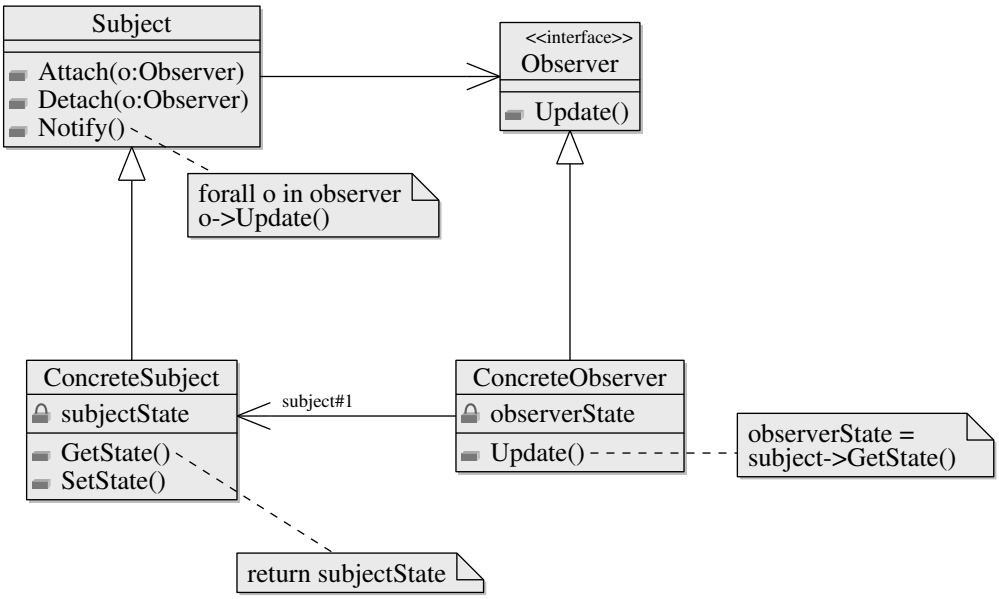


Figura 5.2: Padrão *Observer*

dores devem ter em comum, mantendo a relação entre `subject` e observadores mínima e abstrata.

Este padrão de projeto é amplamente utilizado no desenvolvimento de jogos. Como a arquitetura deste tipo de *software* é caracterizada pela grande interação entre diferentes entidades, faz-se necessário o uso de uma solução padronizada como *Observer*. Assim, entidades se cadastram para receber atualizações de outras entidades que compõem o ambiente virtual.

5.2.3 *Façade*

"Oferecer uma interface unificada para um conjunto de interfaces de um subsistema. Define a interface de alto-nível que torna o subsistema mais fácil de manusear." [26]

Sistemas podem ser estruturados em subsistemas para reduzir a complexidade. Um dado subsistema, no entanto, é composto de diversas classes, o que torna a sua utilização complexa. O princípio deste padrão está em minimizar dependências entre subsistemas.

O padrão *façade* institui uma classe (que leva o mesmo nome do padrão) a fornecer uma simples interface para a utilização do subsistema, evitando, para o caso geral, que seja necessário se comunicar com outras classes do subsistema. Sendo assim, esta classe delega solicitações aos objetos apropriados do subsistema, ocasionalmente sendo necessário realizar operações para atingir tal fim.

O uso deste padrão não impede que as classes do subsistema sejam acessadas diretamente, sendo a sua adoção voltada para facilitar o uso do subsistema no caso geral.

Em uma arquitetura de jogo eletrônico, pode-se utilizar este padrão *façade* para diminuir as dependências entre os subsistemas gráfico, de *input* e de som, com os subsistemas de lógica de jogo, criando assim uma interface única, denominada *Engine*.

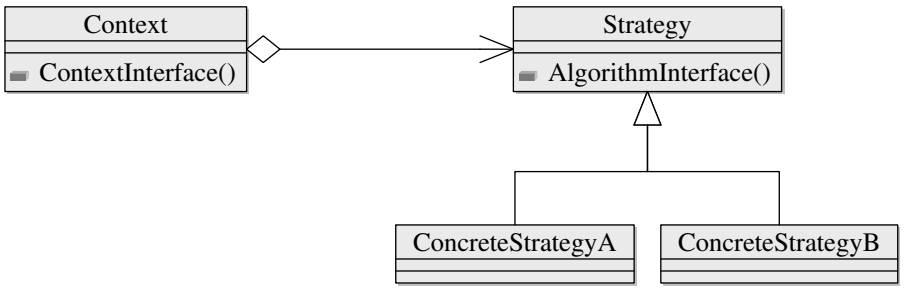


Figura 5.3: Padrão *Strategy*

5.2.4 *Strategy*

"Define uma família de algoritmos, encapsula cada um, e os torna intercambiáveis. Permite que o algoritmo varie independentemente dos clientes que fazem uso deste." [26]

O uso deste padrão é indicado quando se têm uma família de algoritmos relacionados e estes são usados de forma intercambiável. A solução é definir classes que encapsulam estes algoritmos, como apresentado na figura 5.3.

A classe `Strategy` é responsável por definir a interface comum aos algoritmos. Esta é usada por `Context` para usar os algoritmos definidos em uma `ConcreteStrategy`.

A classe `ConcreteStrategy` implementa o algoritmo obedecendo à interface definida em `Strategy`.

A classe `Context` é configurada para usar determinada `ConcreteStrategy`, através de uma referência mantida ao objeto da estratégia selecionada. É possível permitir uma interface que permita à classe `Strategy` o acesso aos seus dados.

5.3 Áreas de reuso

Conforme sugerido por Folmer, em [28], uma entidade de um jogo eletrônico é formada por diversos componentes distintos, cada qual com um comportamento específico. Estes componentes podem ser reutilizados por diferentes entidades, evitando assim o replicamento do código de forma desnecessária. O reuso diminui a complexidade de manutenção e ampliação deste código. Alguns exemplos de componentes são:

- **Rede**: Trata da comunicação entre clientes de jogo e servidores.
- **Gráficos**: Responsável pela representação gráfica.
- **GUI** (*Graphical User Interface*): Funcionalidade necessária para a construção de interfaces de jogo.
- **Inteligência artificial**: Permite que entidades do jogo adotem comportamentos autônomos, simulando inteligência.
- **Física**: Funcionalidades e simulações físicas, tais como detecção de colisão e aderência às leis de Newton.

- **Som**: *Playback* e outras funcionalidades relacionadas ao áudio.

Foge do escopo deste trabalho uma discussão detalhada acerca de cada área de reuso. A lista acima é um ponto inicial para que o leitor possa identificar algumas das diversas áreas de reuso existentes na área de desenvolvimento de jogos. Além destas áreas, devemos considerar aspectos específicos do jogo em questão, pois para cada jogo existem também diversas peculiaridades. Aos aspectos intrínsecos do jogo é dado o nome de **lógica de jogo**.

Esta distinção entre componentes permite que a arquitetura do *software* seja construída de uma forma mais dinâmica, como exposto na seção a seguir.

5.4 Sistema de componentes

O sistema orientado a objetos não nos obriga a utilizar hierarquia de classes para estruturar as entidades do jogo. É possível utilizar uma alternativa que permite uma maior dinâmica da estrutura ainda em tempo de execução. Uma arquitetura orientada a componentes é uma solução que flexibiliza os comportamentos das entidades do jogo através de uma agregação de componentes independentes em uma classe única.

A seguir será apresentado um contraste conceitual entre a hierarquia de classes e a composição de classes.

5.4.1 Desvantagens da hierarquia de classes em jogos

Uma maneira tradicional de representar entidades do jogo é através de uma decomposição hierárquica das entidades que se deseja representar. Um exemplo de tal hierarquia pode ser visto na figura 5.4.

De acordo com Rabin *et al.* [29], uma arquitetura de jogo baseada inteiramente em hierarquia de classes possui algumas limitações. O primeiro problema é o forte acoplamento entre classes. Um *software* com um acoplamento forte é indicativo de ser pouco modularizado e complexo de ser modificado. A herança cria o maior acoplamento possível entre duas classes, pois a classe base possui conhecimento sobre as estruturas públicas e protegidas da classe pai. Ou seja, mudanças na classe pai, frequentemente representam mudanças na classe base.

Outra limitação exposta em [29] é a falta de flexibilidade da hierarquia de classes. Em um sistema complexo, como um jogo eletrônico, podem existir situações difícies de modelar através de hierarquia. Existem características comuns entre entidades de jogo que não possuem herança comum, logo isto leva a uma replicação de código e comportamento semelhantes, prejudicando a manutenção desta estrutura.

Por fim, a estrutura imposta por hierarquias é estática em linguagens comuns como C++ e JAVA. Durante a execução não é possível alterar a herança de classes. Em um *software* comum isto não costuma ser um limitante, mas em um jogo eletrônico muitas

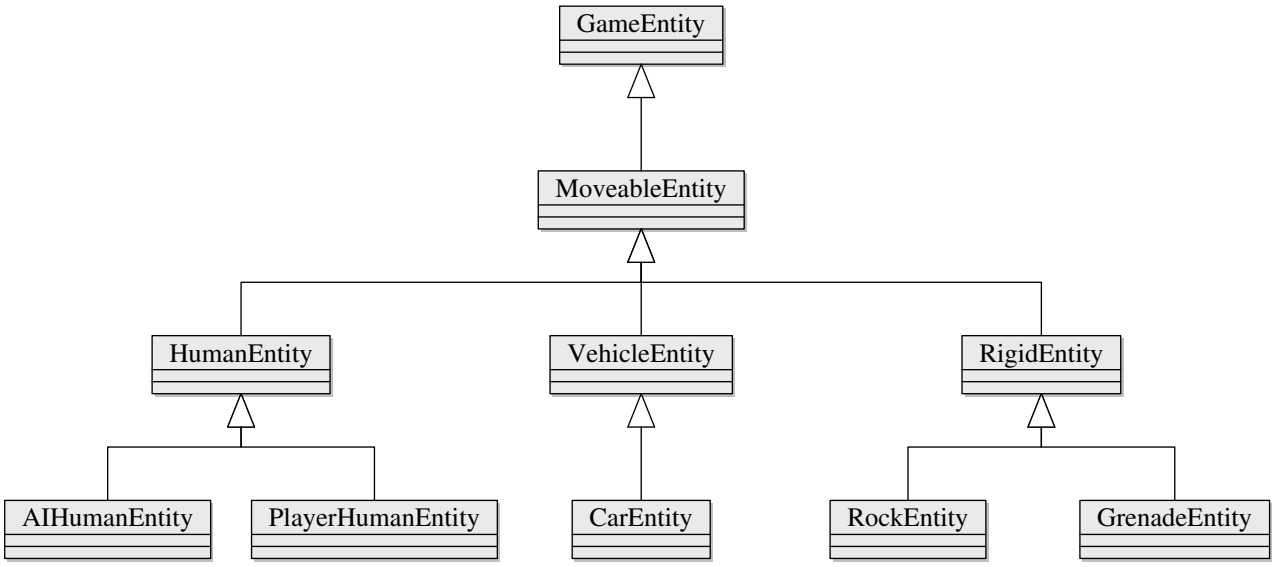


Figura 5.4: Exemplo do uso de hierarquia em entidades de jogo.

vezes entidades alteram drasticamente seu comportamento em tempo de execução. Contextualizando em um jogo, uma entidade `Inimigo`, por exemplo, após sofrer uma derrota, se torna um `Veiculo` para o personagem principal do jogo e altera completamente seu comportamento. Em um sistema estruturado por hierarquias as soluções para tal mudança levam o replicamento do código. Esta situação exemplifica a falta de flexibilização da hierarquia de classes.

5.4.2 Solução por composição de classes

Um sistema de componentes utiliza composição para estruturar as entidades do jogo. Ao invés de cada entidade do jogo possuir uma classe distinta, apenas uma classe é criada com o objetivo de simbolizar todas as entidades do jogo. Esta classe pode, por exemplo, ser denominada de `GameEntity`. Esta classe única contém diversos componentes que juntos compõem a estrutura de um entidade, como exemplificado na figura 5.5.

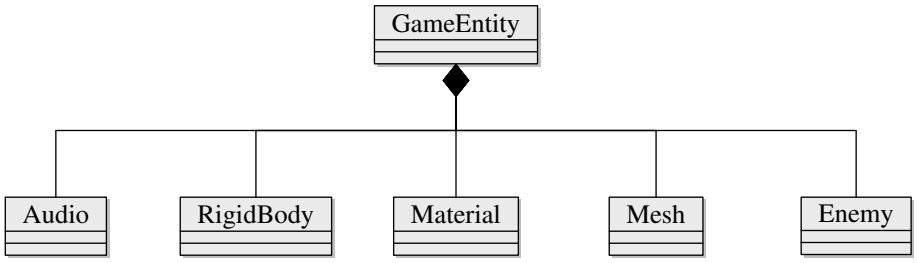


Figura 5.5: Classe `GameEntity` composta por diversos componentes.

Como mostrado na seção 5.3, cada componente é independente dos demais, possibilitando que a entidade do jogo seja composta por qualquer combinação entre som, física,

IA, GUI, gráfico, rede ou componentes de lógica de jogo. A classe `Inimigo`, no exemplo citado acima, pode ser substituída por um `GameEntity` com componentes de `Audio`, `RigidBody`, `Material`, `Mesh`, `Enemy`, dentre outros (figura 5.5).

Assim, para modificar a entidade `Inimigo` modifica-se sua estrutura de componentes. Tirar o componente de lógica `Enemy` e acrescentar o componente `Vehicle`, ainda em execução, modifica inteiramente o comportamento de lógica de jogo, entretanto mantém outros aspectos que ainda são comuns, como aparência, corpo físico e emissão de sons.

Nesta arquitetura de sistema, a entidade `GameEntity` não precisa conhecer a estrutura específica de cada componente, pois ela é apenas uma estrutura de agregação entre estas classes. Assim, o código mantém um acoplamento baixo, permitindo sua manutenção e modificação com mais facilidade.

5.4.3 Comunicação entre componentes

É comum a necessidade da troca de mensagens entre componentes da mesma entidade. Logo, a `GameEntity` pode ser responsável por transmitir estas mensagens entre os componentes próprios a ela. O componente de lógica `Enemy` pode requerer ao componente de `Material` a troca de uma imagem de textura.

Entretanto, existe a possibilidade da `GameEntity` não possuir um determinado componente destino de uma mensagem. Neste caso a `GameEntity` pode ignorar a mensagem ou retornar um erro para o componente remetente. Este tipo de tratamento é necessário, pois a classe da entidade pode existir sem possuir determinados componentes.

Existe ainda uma outra solução muitas vezes necessária para o funcionamento de componentes. Um componente pode requerer a existência de outro componente na mesma entidade. A entidade `Inimigo` possui, por exemplo, um componente `RigidBody` que armazena informações sobre sua resposta a efeitos dinâmicos da física. Este componente pode requerer a atribuição de um componente `Collider` para delimitar as colisões com outras entidades no espaço virtual do jogo.

Em um sistema mais simples orientado a componentes, a troca de mensagens entre componentes pode ser uma simples chamada de função. Assim, na inicialização dos componentes, ponteiros são passados entre eles para garantir a comunicação.

5.4.4 Sistema orientado a dados

Rabin *et al.* vai além da estrutura de componentes. Ele exemplifica a construção de um sistema orientado a dados (*data-driven system*) que, ao invés das entidades serem descritas no código do jogo, são construídas dinamicamente através de informações coletadas a partir de arquivos de dados, como um arquivo no formato XML (*Extensible Markup Language*).

Estas descrições em arquivos mostram a estrutura de componentes de cada entidade. Assim, durante a execução do jogo, é possível carregar estes arquivos e instanciar todas as entidades com seus respectivos componentes. Esta solução orientada a dados facilita

o desenvolvimento de ferramentas de edição que permite a criação de novas entidades de jogo por aqueles não-programadores, como artistas e designers que fazem parte da equipe de produção.

Uma abordagem orientada a dados permite a utilização de ferramentas para auxiliar no desenvolvimento do jogo. Pode-se citar como exemplo um editor de fases do jogo, que permite editar todos os aspectos da fase através de um editor integrado, sem haver a necessidade de alterações no código. As vantagens disso são inúmeras, dentre as quais podemos mencionar as seguintes. Não é necessário que o código seja compilado a cada alteração, já que os dados são carregados dinamicamente. Acelera o processo de desenvolvimento ao tirar das mãos do programador tarefas de codificação de entidades do jogo. Permite que não-programadores desempenhem tarefas de *level design*[2]. Em um jogo suficientemente grande, o desenvolvimento de ferramentas desta natureza pode ser decisivo para o sucesso da equipe de desenvolvimento.

[2]Nome atribuído ao responsável pela elaboração de fases em jogos.

Capítulo 6

Game design do *Eco Defense*

Este capítulo tem como objetivo apresentar uma breve descrição do jogo *Eco Defense*. O método de descrição utilizado é denominado *game design*, que advém de uma necessidade de padronizar projetos de jogos eletrônicos. Assim, este método propõe a padronização de documentos que incluem requisitos de alto nível, conceito do jogo, mecânica de interação, visual dos gráficos, dentre outros [29].

6.1 Requisitos de Alto nível

O jogo *Eco Defense* deve atender uma série de requisitos para possibilitar testes de interação com a mesa multitoque construída. Estes requisitos foram levantados de forma subjetiva pela equipe de desenvolvimento deste trabalho.

1. Sua mecânica de jogo deve permitir a participação de diversos jogadores simultaneamente.
2. A interface do jogo deve prever a distribuição dos jogadores ao redor da mesa.
3. O jogo deve interpretar diversos toques simultaneamente.
4. Deve ser feito o uso de gestos na mecânica de jogo.
5. Um tema ou enredo do jogo deve ser voltado a um contexto sério, como sustentabilidade e reciclagem.

6.2 Proposta do Eco Defense

6.2.1 Conceito geral

O *Eco Defense* tem como objetivo utilizar-se da mesa multitoque e trazer como conteúdo inovações tecnológicas que combatem os problemas do meio-ambiente. A identidade visual do jogo está representada na figura 6.1. Durante o jogo, seus participantes deverão discutir

Figura 6.1: *Banner* do jogo.

colaborativamente como investir seu dinheiro para controlar a poluição gerada por uma fábrica. Diversos poluentes saem de uma fábrica no centro da tela e os jogadores devem pressionar seus dedos contra estes poluentes para que sejam destruídos. Os poluentes também são destruídos por torres que os jogadores optam por construir. Os usuários tomam decisões em conjunto de como investir o dinheiro arrecadado na construção de novas torres. À medida que o jogo avança, mais poluentes aparecem da fábrica.

O *Eco Defense* possui um contexto atual sobre as questões enfrentadas a favor da sustentabilidade e da qualidade de vida. As torres que os jogadores podem escolher são baseadas em soluções encontradas na atualidade, mas não se comportam como na realidade. Seu comportamento é construído de forma lúdica que figurativa um combate das tecnologias com os poluentes. Os jogadores possuem um papel de tomada de decisão a respeito dos investimentos, e possuem como objetivo reverter o processo de aquecimento global no ambiente do jogo.

6.2.2 Regras de jogo

As regras da mecânica e fluxo de progressão do Eco Defense são as seguintes:

- Os jogadores começam sem dinheiro.
- Para cada poluente destruído os jogadores ganham dinheiro.
- A fábrica, no centro da tela, produz uma quantidade de poluentes relativo ao nível de dificuldade. Ou seja, nos níveis iniciais, a quantidade de poluente é pequena. Nos níveis finais do jogo a quantidade de poluente é grande.
- Estes poluentes se movem do centro da tela para a floresta.
- Se um poluente atinge o limite da tela, os jogadores perdem uma árvore da floresta.
- Se todas as árvores acabarem, os jogadores perdem o jogo.
- Para destruir os poluentes os jogadores podem:

– tocar os poluentes com os dedos.
– investir seu dinheiro em torres que destroem os poluentes.

6.2.3 Objetos essenciais

No Eco Defense, existem dois objetos essenciais, os Poluentes e as Torres, que são detalhados abaixo.

Poluentes

Poluentes são gerados pela fábrica em diferentes quantidades, velocidades e resistência, e podem ser dos seguintes tipos:

- Nuvem cinza - Este poluente é destruído com apenas um toque do usuário. Possui uma velocidade de deslocamento inferior às outras nuvens. Se atingir a floresta, o jogador perde uma árvore.
- Nuvem preta - Ao ser destruída, esta nuvem se divide em duas outras nuvens cinzas. Se atingir a floresta, o jogador perde três árvores.
- Nuvem laranja - Esta nuvem possui uma velocidade maior e, por isso, se torna mais difícil de ser destruída. Se atingir a floresta, o jogador perde cinco árvores.

Torres

São entidades do jogo que *atacam* os poluentes de uma forma direta. Possuem diferentes tipos de ataque em diferentes velocidade.

- Painel solar
- Misseis químicos

6.2.4 Conflitos e soluções

Os jogadores em cada turno devem avaliar suas necessidades com relação aos seus investimentos. Existem investimentos que custam pouco, mas trazem menos benefícios. Existem investimentos caros, entretanto resolvem os problemas dos poluentes por muito mais tempo. Assim, os jogadores devem avaliar a necessidade dos investimentos naquele momento do jogo.

Quanto mais jogadores entram no jogo, maiores são as chances de se atingir um melhor resultado. Com mais dedos disponíveis, é possível destruir mais poluentes ao mesmo tempo e obter mais dinheiro para investimento. Assim o jogo promove a participação de vários usuários colaborativamente.

Todos os jogadores compartilham o dinheiro do jogo, assim, se um dos jogadores investir na construção de torres, o dinheiro de todos é gasto. Por isso é importante a comunicação entre os jogadores para a tomada de decisão.

6.2.5 Fluxo do jogo

O jogo é baseado em um principio lúdico onde é fácil aprender mas se torna difícil ao longo do tempo. Os níveis iniciais do jogo são simples e não demandam a construção de torres. Apenas os toques com as mãos podem resolver os problemas dos poluentes. Mas à medida que o jogo avança, a quantidade de poluentes que sai da fábrica cresce exponencialmente, dificultando a vida dos jogadores.

Assim em poucos minutos os jogadores podem ter uma sensação de vitória e não vivenciam frustração logo no início. Mas se os jogadores desejam investir seu tempo no jogo, o nível de dificuldade rapidamente se tornará um obstáculo, pois a quantidade de poluentes será cada vez mais desafiadora.

6.2.6 Interface do jogo

A interface do jogo Eco Defense deve atender aos requisitos da mesa multitoque, sendo distribuída em torno da superfície de projeção. Desta maneira todos os usuários podem vivenciar a mesma experiência. Na figura 6.2, pode-se visualizar a distribuição dos botões e textos. A interface é replicada em lados opostos da projeção, e contém os valores referentes a quantidade de árvores, dinheiro e nível de dificuldade atingido pelos jogadores. Além disso, contém os botões Painel Solar e Reciclagem, que ao serem clicados criam uma nova torre.

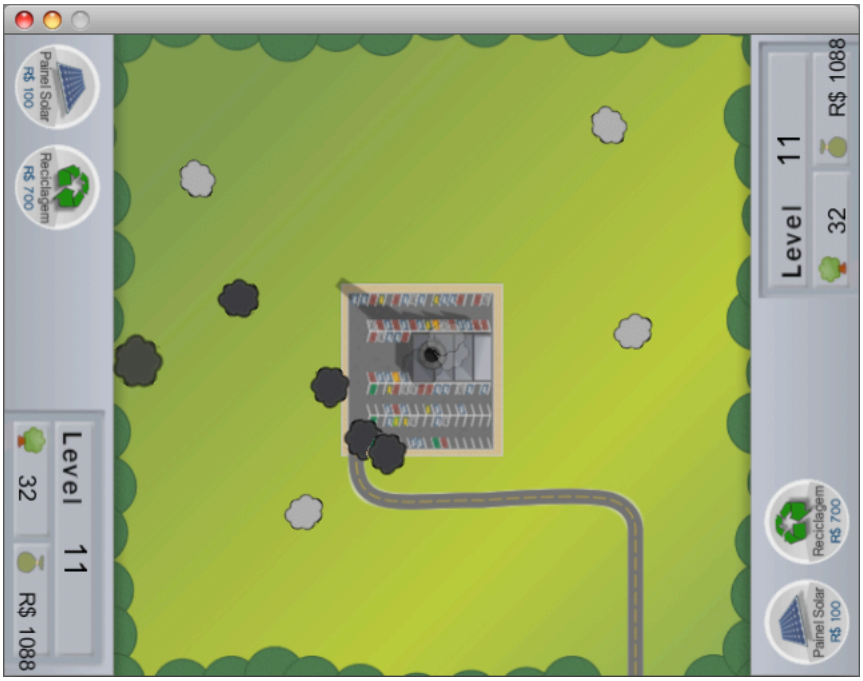


Figura 6.2: Esquema da interface *in-game*.

Capítulo 7

Construção da mesa multitoque

Através do estudo dos diversos temas apresentados nesta monografia e utilizando-se do aprenzidado dado pelas soluções previamente testadas, é possível construir uma mesa multitoque a baixo custo com a utilização de componentes e equipamentos facilmente encontrados no mercado. O foco deste trabalho não é a construção da mesa, ou seja, esta estrutura física será utilizada como solução alternativa à aquisição de uma superfície multitoque.

Neste capítulo é descrita a construção desta interface como ilustrado na figura 7.1.

7.1 Mesa multitoque

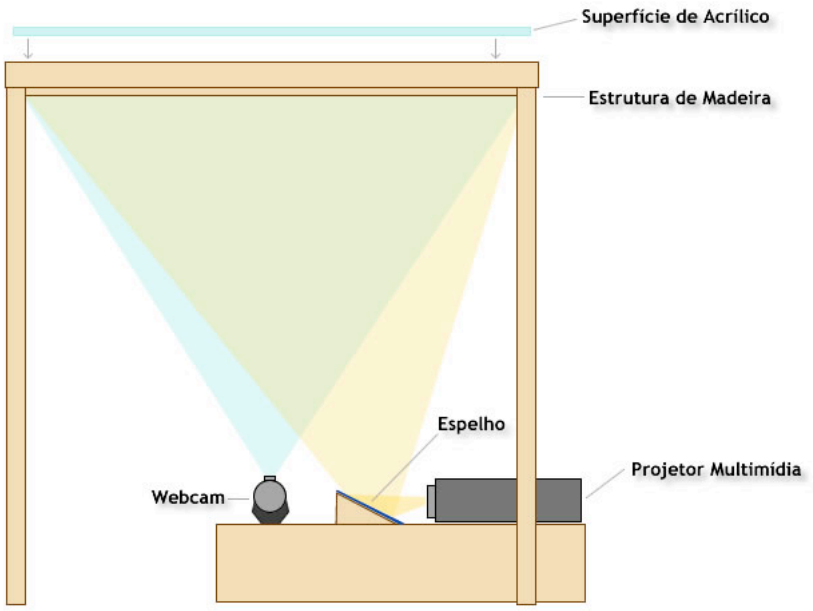


Figura 7.1: Esquema base da proposta de mesa multitoque.

7.1.1 Estrutura

A estrutura é constituída de madeira que dá suporte à superfície de acrílico e se posiciona a uma altura aproximada de 1 metro, para que os usuários possam interagir ainda em pé. A madeira possui encaixes para a iluminação e a estrutura é montada através de parafusos e porcas, que permitem um desmonte da mesa. As dimensões são mostradas na figura 7.2.

A estrutura é essencial para o tipo de iluminação utilizada. Como mencionado anteriormente nesta monografia, existem dois padrões básicos de iluminação: FTIR e DI. O FTIR exige a distribuição dos LEDs ao longo da lateral do acrílico, dispostos de maneira a iluminar a toda a superfície. No caso do DI, utilizado por este trabalho, a estrutura serve de apoio para posicionar o acrílico a uma distância adequada de duas fontes de luz infravermelhas posicionados abaixo junto ao projetor.

7.1.2 Superfície

A superfície é constituída de acrílico de espessura de $5mm$. A escolha do material foi baseada no coeficiente de refração que permite que a iluminação percorra toda a superfície. O acrílico tem dimensões aproximadas de $65 \times 45cm$. A espessura foi determinada para garantir a sustentação da superfície mesmo quando pressionada por vários usuários.

Abaixo da superfície, utiliza-se uma película de projeção para receber a imagem gerada pelo computador. Esta superfície é essencial, também, para a iluminação correta dos dedos sobre a superfície. Sem a película, toda a mão do usuário seria iluminada igualmente. Com a película, os dedos ficam melhor iluminados se comparadas à palma da mão. Para a película, utiliza-se um papel vegetal comum esticado sob a superfície de acrílico.

7.1.3 Iluminação infravermelha

A iluminação infravermelha é oriunda de 96 LEDs montados sobre duas placas de circuito. Estes LEDs são ligados a uma fonte ATX de 12V e organizados em 12 circuitos paralelos de 8 LEDs. Como mostrado na figura 7.3.

Os LEDs infravermelhos utilizados possuem uma voltagem (*forward voltage*) de $1,4V$ e necessitam de uma corrente aproximada de $20mA$. Para tal, utilizou-se um resistor de 39 ohms em cada circuito paralelo.

7.1.4 Câmera

Com intuito de prover a imagem adequada para captura dos toques, utiliza-se uma câmera que filtre a luz do espectro visível e capture apenas a luz infravermelha. Existem no mercado câmeras que possuem tal funcionalidade, mas com o objetivo de reduzir os gastos, utilizou-se uma câmera comum, *Microsoft VX-1000*.

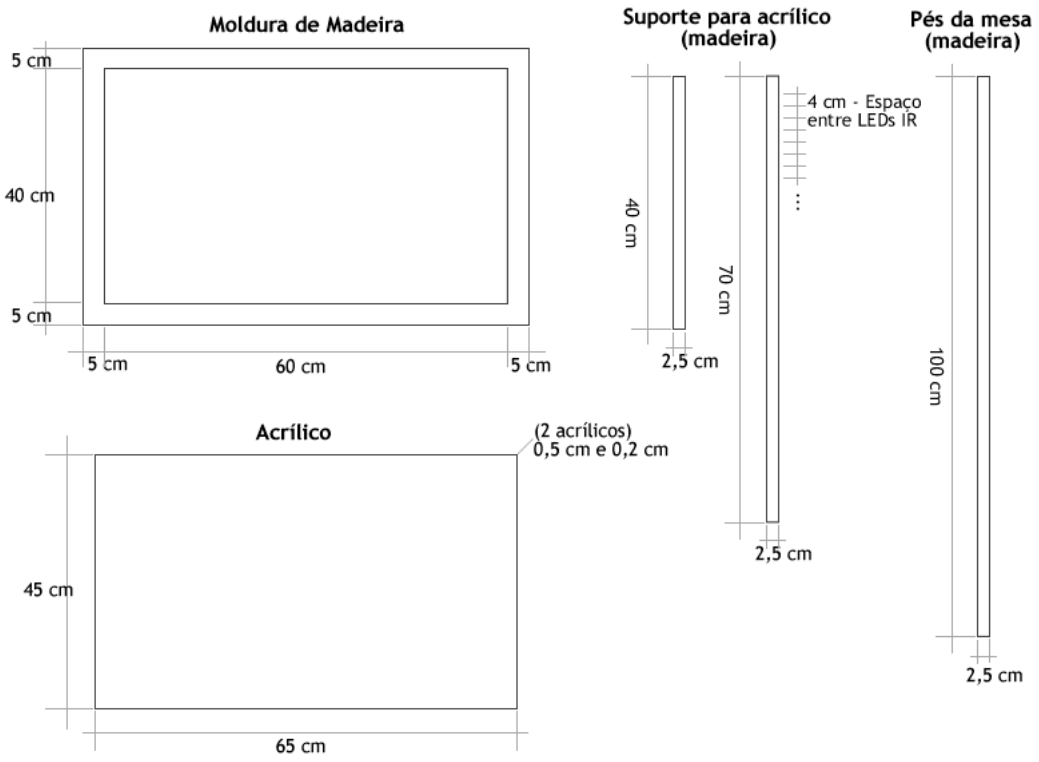


Figura 7.2: Dimensões da estrutura em madeira e da superfície de acrílico.

Esta câmera possui características adequadas de resolução (640x480 *pixels* e 320x240 *pixels*) e *frame rate* ($30Hz$ e $60Hz$), e permite a remoção do filtro de luz infravermelho de fábrica. Foram incluídos dois filtros de luz visível dentro da câmera, que melhoram o contraste da imagem com relação a luz infravermelha. Como filtro, é utilizado filme fotográfico revelado exposto à luz visível que possuem esta característica de filtragem.

7.1.5 Projeção

A projeção é obtida através de um projetor XVGA de resolução padrão 1024 x 768 *pixels*, a uma distância aproximada de $80cm$ da superfície. Coloca-se um papel vegetal sobre a superfície para que a imagem seja nitidamente projetada e permita uma boa qualidade de imagem.

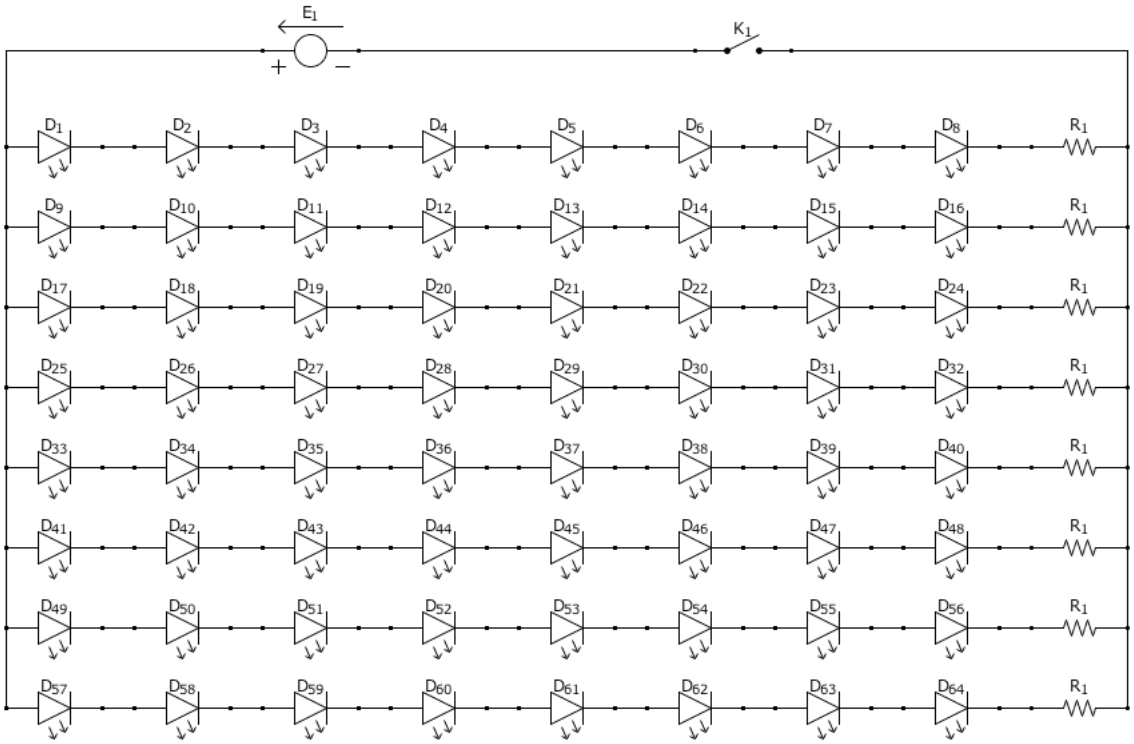


Figura 7.3: Circuito dos LEDs infravermelhos, onde $R_1 = 39\Omega$ e $E_1 = 12V$

Capítulo 8

Implementação do *software* interativo

Este capítulo esclarece a estrutura implementada do *software* do jogo. Primeiramente, apresenta uma visão geral das ferramentas utilizadas para o desenvolvimento. Em seguida, é explicado a arquitetura de alto-nível e a arquitetura dos componentes individuais, tais como a estrutura dos objetos do jogo e o sistema desenvolvido para reconhecimento de gestos.

8.1 Linguagens e bibliotecas

As linguagens e bibliotecas disponíveis para o desenvolvimento são essenciais para facilitar o acesso ao *hardware* e prover o *feedback* esperado por um jogo eletrônico.

Neste trabalho toda a implementação do jogo foi feita usando a linguagem de programação C++. De acordo com Rabin *et al.*, em [29], esta linguagem é amplamente usada nos jogos comerciais, sendo a mais adotada para tal finalidade. Por incorporar orientação a objetos, se torna mais adequada do que puramente a linguagem C, já que a arquitetura de um jogo potencialmente tira muito proveito da orientação a objetos. Sua biblioteca padrão, STL (*Standard Template Library*), oferece algumas estruturas complexas de dados já implementadas. Por fim, C++ é uma linguagem muito eficiente, um dos requisitos para se desenvolver jogos com boa qualidade.

Além de C++, foram usadas e integradas diversas ferramentas e bibliotecas para compor o produto final.

Para a construção de um jogo, é necessário o uso de alguma ferramenta que gere o *feedback* visual e sonoro. Assim os usuários podem interagir diretamente com o que vêem e receber um retorno em tempo real de suas ações. É necessário que a ferramenta funcione com a biblioteca escolhida para realizar a tarefa de visão computacional. Foram encontradas diversas soluções que atendem a estes requisitos, tanto em nível de *hardware* como *software* disponíveis, mas a SDL (*Simple DirectMedia Layer* [30]) foi escolhida pois atende as necessidades do projeto.

Esta biblioteca é uma abstração multiplataforma para multimídia. Uma de suas vantagens é oferecer uma fina camada entre o *hardware* e *software* para prover esses serviços multimídia. Possui *bindings* para uma variedade de linguagens, incluindo C++, Python e Java. Sendo assim, é respeitado entre desenvolvedores de jogos e utilizado em alguns títulos comerciais.

Com relação à parte gráfica, o SDL foi utilizado apenas para disponibilizar o acesso direto ao ambiente *OpenGL* (*Open Graphics Library*). Para carregar imagens em diversos formatos foi utilizada a pequena biblioteca *SOIL* (*Simple OpenGL Image Library* [31]), que carrega imagens diretamente em texturas *OpenGL*. Para carregar fontes *TrueType* e gerar texturas foi usada a FTGL (*Free TrueType OpenGL Library* [32]), que por sua vez usa a biblioteca *FreeType*. Estas foram usadas no principal intuito de facilitar o desenvolvimento, permitindo que componentes gráficos possam ser facilmente expandidos.

Para a parte de áudio, a abstração do SDL fornece acesso direto ao *stream* de áudio. Por ser uma abstração simples, não trata da mixagem de sons, o que se torna necessário quando se deseja mais de um som em execução ao mesmo tempo. Para este fim, foi utilizado o SDL_mixer, um *plugin* para o SDL que permite a execução de diversos sons simultaneamente.

Por fim, para o *input* foi utilizada a Touchlib [19]. Do ponto de vista do *software*, este opera de acordo com o protocolo TUIO [33], que por sua vez utiliza o *OpenSound Control*. Consoante com a decisão de utilizar a biblioteca TUIO para transmissão de dados do *input* está o fato de este acompanhar *bindings* para a linguagem C++.

8.2 Visão geral da arquitetura

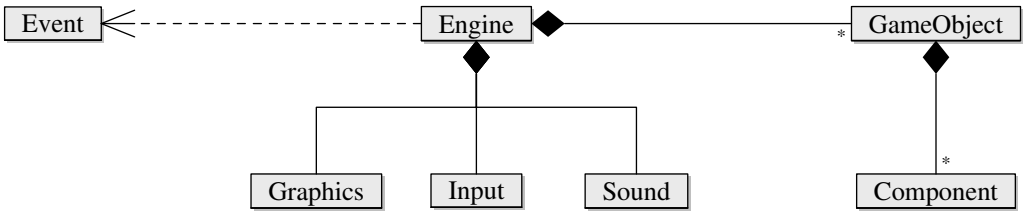


Figura 8.1: Visão geral das classes do jogo

Como visto na figura 8.1, a arquitetura do jogo é composta por uma controladora principal `Engine` que gerencia os subsistemas e as entidades de jogo. A classe `Engine` é intrinsecamente bastante simples. Gerencia as fases de inicialização, execução e encerramento do sistema.

Durante a inicialização, instancia todos os subsistemas e, durante a rotina de encerramento, finaliza cada um na ordem inversa. A fase de execução consiste em um loop infinito que atualiza cada subsistema e todos os `GameObject`s. Este loop somente é encerrado quando um dos subsistemas solicita término de execução.

No passo de inicialização, esta controladora adiciona um `GameObject` que será responsável por iniciar o jogo.

8.3 Subsistemas

Conforme o nome sugere, subsistemas são parte de algum sistema maior. Neste caso, cada subsistema trata de um aspecto específico do jogo. A arquitetura implementada foi dividida em quatro subsistemas: Gráfico, *Input*, Som e Eventos. O funcionamento de cada um consiste em três etapas:

- **Inicialização**: Inicializa a classe principal e suas dependências. Se for necessário, é nesta etapa em que recursos devem ser obtidos e o ambiente correspondente ao domínio do subsistema é inicializado. Alguns subsistemas devem se registrar no subsistema *Events* para receber alguns tipos de eventos.
- **Atualização**: Esta etapa é executada uma vez a cada iteração do *looping* da `Engine`, e nela são executadas rotinas que devem ser atualizadas a cada frame.
- **Encerramento**: Libera os recursos obtidos durante a inicialização.

Os subsistemas acessam uma controladora de mensagens, `EventManager` do subsistema *Events*, para se comunicar e receber eventos de outros subsistemas. Cada entidade do jogo e subsistema pode registrar-se para receber um determinado tipo de evento. A discussão deste tópico será ampliada na seção 8.4.

A arquitetura do jogo implementado seguiu o padrão de sistema de componentes, que foi explicada em 5.4. Assim, além de uma controladora principal, todo subsistema possui famílias de componentes que agregam comportamento às entidades do jogo. Os componentes possuem nomes com prefixo `GOC`, como em `GOCRenderTexture`, componente do subsistema gráfico.

8.3.1 Gráfico

O subsistema gráfico possui uma controladora principal denominada `Graphics`. Esta é responsável pela renderização de todos os elementos visíveis na tela (por exemplo texto, texturas ou animações). Durante a inicialização, são invocados comandos para instanciar a tela através do SDL e obter o contexto *OpenGL*. Em seguida, o ambiente *OpenGL* é inicializado para renderizar gráficos em 2D de forma simples. Isto é feito ajustando-se as matrizes de projeção (`GL_PROJECTION`) e model-view (`GL_MODELVIEW`) do *OpenGL*.

Além disso, esta classe se registra para o recebimento de eventos do tipo `GraphicsEvent`. Portanto cada componente da família `GOCRender` se registra e desregistra, sendo mantida dentro da classe uma lista de todos os componentes registrados.

A atualização do `Graphics` consiste em limpar o *buffer* de exibição e executar a atualização de todos os componentes registrados para este subsistema. Cada entidade no jogo que possui um componente `GOCRender` registrado é desenhada neste buffer. Para a

exibição na tela, o *buffer* em questão é trocado com o *buffer* da placa de vídeo. Para este processo, o OpenGL foi inicializado com o modo *double-buffer*[1].

8.3.2 *Input*

O subsistema de *input* é controlado primariamente por uma classe denominada `Input`. A entrada do jogo tem duas fontes. Uma é oriunda do SDL e trata de convencionais eventos de teclado e mouse. É usada apenas para testes e encerramento da aplicação. A outra fonte é oriunda da Touchlib. Conforme já foi explicado, estes eventos são transmitidos por meio do protocolo TUIO. Foi utilizado um cliente fornecido em C++ para recebimento de eventos.

O cliente fornecido, `TuioClient`, se associa a uma porta UDP pré-estabelecida, escutando os eventos que nela chegarem. Esta classe é executada em outra *thread*, permitindo a escuta dos eventos de forma paralela. `TuioClient` não trata os eventos de nenhuma forma, apenas fornecendo os eventos a classes *listener*, previamente registradas no `TuioClient`. Cada objeto *listener* deve aderir à interface (através de herança) especificada em `TuioListener`, que especifica os métodos referente às alterações no estado da mesa. (Esta situação corresponde ao padrão de projeto *Observer*.)

Sendo assim, `Input` herda de `TuioListener`. Esta especifíca um método para cada alteração no estado da mesa. No caso desta aplicação, são de interesse apenas os eventos envolvendo toque: `addTuioCursor(TuioCursor *tuioCursor)`, `updateTuioCursor(TuioCursor* tuioCursor)` e `removeTuioCursor(TuioCursor *tuioCursor)`.

Na inicialização do `Input`, é instanciado um objeto `TuioClient` que registra o próprio `Input` como receptor dos eventos. Estes eventos são enviados através do sistema de eventos, com o tipo `FingerInputEvent`. Sendo assim qualquer entidade interessada em receber eventos de input deverá registrar-se como receptora deste tipo de evento. Cada evento possui posição e um ID, referente à "instância do dedo" que provocou aquele evento. Desta forma, é possível acompanhar o movimento ao longo da superfície de um único dedo com facilidade.

Do lado do servidor, a Touchlib foi configurada para utilizar os filtros que foram descritos na seção 3.1. Esta configuração é feita através de um arquivo XML, cujos detalhes podem ser encontrados no apêndice A.

8.3.3 Áudio

O subsistema responsável pelo áudio é controlado pela classe `Sound`. Durante a inicialização, é aberta uma instância do *SDL_mixer*. Como a execução do som é delegada ao *SDL_mixer*, não é necessário nenhum passo de atualização. Durante o encerramento basta fechar o canal de áudio.

[1]Em *double-buffering* são usados dois buffers em paralelo, onde um buffer é lido enquando o outro é escrito. Ao final de cada *frame*, os buffers de leitura e escrito são trocados. Esta técnica é adotada no intuito de evitar artefatos visíveis durante o processo de renderização do *frame*, sendo este apenas exibido uma vez concluído.

8.4 Eventos

Essencial ao desenvolvimento de qualquer jogo é a comunicação entre objetos. Ingenuamente, pode se pensar que a comunicação pode ocorrer por chamadas de funções, e isto sem dúvida não é errado. Mas existem casos em que o fluxo de informação é mais complexo e um mecanismo mais flexível deve ser adotado.

Para o caso em que várias entidades estão interessadas na notificação da ocorrência de determinado tipo de evento, ou quando se deseja enviar um evento mas não se sabe previamente quem deve recebê-lo, utiliza-se este sistema.

Existem diversos tipos de evento. Para cada tipo, cria-se uma classe, que pode conter todas as estruturas de dados necessárias. Uma classe central é responsável pelo gerenciamento dos eventos. Objetos que desejam receber certo tipo de evento devem se registrar com esta classe, informando também o tipo de evento pelo qual se interessa. Objetos que desejam enviar um evento bastam especificar o evento como parâmetro.

O subsistema é composto pelos seguintes elementos:

- **Object**: qualquer objeto pode enviar ou receber eventos, acessando o *singleton* **EventManager**. Caso o objeto opte por receber eventos, deverá possuir um método responsável por tratá-los.
- **EventManager**: responsável por registrar e desregistrar objetos, além de enviar eventos quando solicitado.
- **Event**: classe da qual eventos concretos herdam. Especifica apenas o tipo de mensagem.

Em seu funcionamento interno, a classe **EventManager** utiliza a estrutura de dados `std::map`, juntamente com `boost::signals`

Tem-se, aqui, a aplicação de dois *design patterns*. A classe **EventManager** é, devido à sua natureza, acessível de maneira global, através do padrão *Singleton*. Isto traz uma maior legibilidade ao código, evitando a necessidade de se passar referências ao **EventManager** ao longo da hierarquia. Além disso, o padrão *Observer* pode ser observado, onde o **EventManager** faz o papel do **ConcreteSubject**, e os **Object**s representam os **ConcreteObserver**s. A funcionalidade é semelhante, com apenas a diferença nos métodos **GetState()** e **SetState()**, que não existem nesta implementação, sendo substituído pelos **Event**s.

O envio de eventos pode ser instantâneo ou pode ser atrasado (*delayed*). Neste segundo caso, o evento é enviado apenas no início do próximo *update*.

8.4.1 Classes de eventos

Existem os seguintes tipos de eventos:

- `GOEvent` é enviado quando se deseja adicionar ou remover *game objects*. É, portanto, registrado pelo `Engine`, e no seu recebimento o mapa de *game objects* será alterado.

Este é o único evento tratado fora de componentes e subsistemas. Especifica a operação (`mOperation`, `ADD` ou `REMOVE`) e o *game object* (`mGO`).

- `AttackEvent` é emitido por componentes que realizam ataques nas nuvens; no caso `GOCAttackMissile` e `GOCTowerMissile`. Êste evento contém como parâmetros o tipo de ataque (`mOperation`), a intensidade do ataque (`mValue`), e a entidade alvo (`mTargetID`). Sendo assim, quem trata do ataque é o próprio componente receptor do *game object* com ID `mTargetID`. O único componente registrado é `GOCPollution`, que destrói a poluição e toma as ações de acordo com o seu tipo.
- `GameEvent` é enviado quando ocorre uma mudança no estado do jogo. Quando o jogo termina, o evento é enviado com o valor *game over*.
- `MoneyEvent` efetua alterações na quantia disponível pelos jogadores. Contém, além da operação (`mOperation`) e do valor (`mValue`), um nome dado ao que provocou o evento (`mName`).
- `PollutionEvent` é enviado no momento que uma poluição é destruída. Informa o tipo de evento (`mOperation`, pode ser destruída por uma árvore, pelo jogador, diretamente por uma torre, ou pelo ataque de uma torre), o valor atribuído ao evento (`mValue`, tipicamente 1) e um ponteiro para a poluição (`mGO`).
- `GraphicsEvent` é enviado por todo componente da família `GOCRender` durante o construtor e destrutor, especificando o tipo de operação (`mOperation`, sendo registrar ou desregistrar) e um ponteiro para o componente (`mGOCRender`).
- `ButtonEvent` envia para a aplicação informações de quando um botão é pressionado ou solto. Informa sua localização (`mX` e `mY`), ponteiro para o botão (`mButtonGO`), `sessionID` que provocou o evento (`mSessionID`) e tipo de operação (`mOperation`, `PRESS` ou `RELEASE`).
- `DragEvent` é enviado por *game objects* que possuem o componente `GOCDraggable` ao serem arrastados. Assim, é possível que outras entidades interessadas no movimento de tais *game objects* possam atualizar seus estados de acordo. Especifica o tipo de operação (`mOperation`), o nome da entidade ao qual se refere (`mEntityID`), e sua nova localização (`mX` e `mY`).
- `FingerInputEvent` é enviado pelo subsistema de eventos baseado em dados recebidos pelo TUIO. Consiste no tipo de evento (`mOperation`), na localização do evento (`mX` e `mY`, convertido para coordenadas da tela), e no ID de sessão fornecido pelo TUIO (`mSessionID`).

8.5 *Game Objects* e *Game Object Components*

Um *Game Object* representa uma entidade que pode existir no mundo do jogo. Consiste, em um nível elementar, de uma identificação, um *Transform* (representando a posição, orientação e escala do objeto), e um conjunto de GOCs (*Game Object Components*).[34] Essencialmente qualquer entidade do jogo é um game object, inclusive entidades não

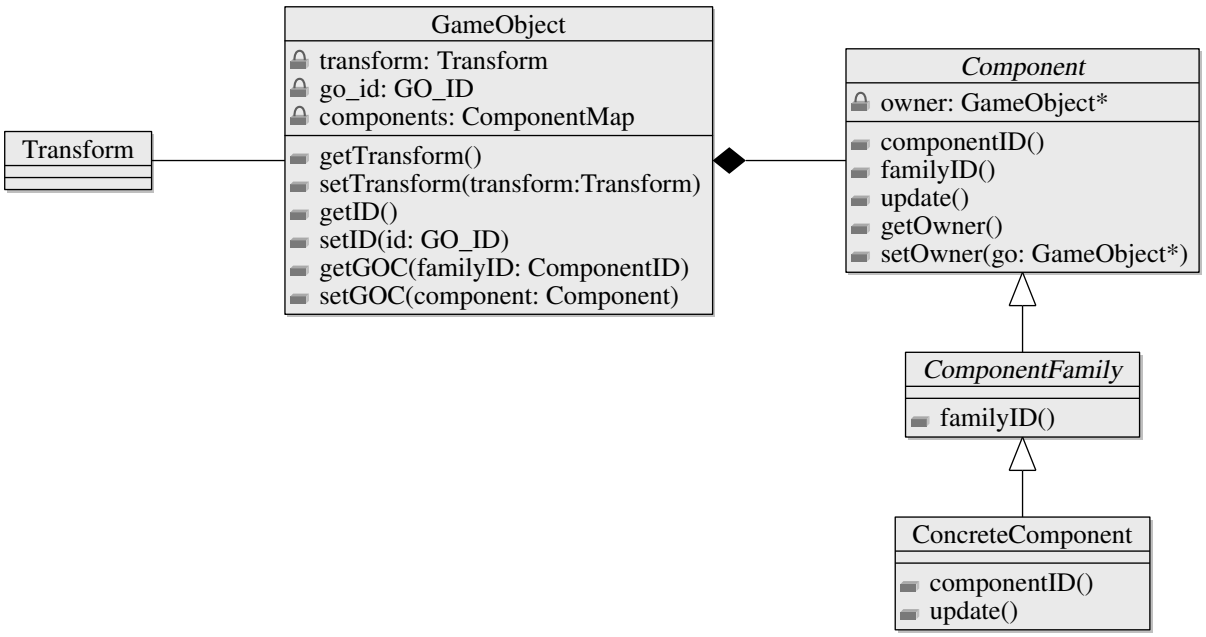


Figura 8.2: Relação entre *Game Objects* e *Game Object Components*

visíveis. Em geral, deseja-se sempre ter componentes associados a um *game object*, caso contrário este não terá comportamento algum.

8.5.1 Componentes

O objetivo de cada GOC é oferecer interfaces simples e funcionalidades muito específicas, permitindo que os *Game Objects* possam fazer uso de diversos GOCs de forma flexível.

Um GOC abstrato (*Component* na figura 8.2) descreve uma simples interface comum que permite com que cada *Game Object* possa gerenciar seus próprios GOCs.

Componentes são organizados de acordo com suas funcionalides. Estes grupos são denominados famílias. A família é obtida por meio do método `familyID()` (puro e virtual em *Component*), e o valor retornado deve ser diferente entre todas as famílias. Cada componente obrigatoriamente pertence a uma família, que define a interface comum àquela família de componentes. Em uma família responsável pela representação gráfica, por exemplo, a interface poderia especificar um método `render()` para todos os componentes pertencentes àquela família.

Ao definir o componente base de uma família, a classe continua abstrata, pois os métodos `componentID()` e `update()` permanecem abstratos. Sendo assim, é necessário que a implementação concreta especifique estes métodos. O método `componentID()` é usado para distinguir os componentes, assim como `familyID()` é usado para distinguir famílias de componentes. A presença do método `update()` também é obrigatória em todo componente, e este método será executado a cada *game loop*.

8.5.2 Gerenciamento de componentes

O gerenciamento de componentes em *game objects* consiste em adicionar e obter um componente. Cada *game object* pode conter no máximo um componente de cada família. Sendo assim, a estrutura de dados adotada é o `std::map` da STL (*Standard Template Library*), onde é feita a relação entre `ComponentID`s e `Component*`.

Para obter um componente de determinada família é usado o método `getGOC` especificando o ID da família desejada. Para adicionar um GOC, basta passar o ponteiro para o GOC que este será adicionado ao mapa de componentes. Uma consulta ao seu `familyID()` irá adicioná-lo naquela categoria no mapa.

Embora componentes devam primar por reduzir dependências entre componentes de outras famílias, ocasionalmente haverá ligação forte com outros tipos de componentes. Através do ponteiro `owner` do próprio componente, é possível acessar o método `getGOC` do *game object*. Existem dois motivos pelos quais este recurso deve ser usado com cautela. O primeiro é que isto insere dependências entre componentes. Isto dificulta a reutilização de determinado componente em outros contextos. O segundo motivo é que em termos de processamento, uma consulta ao mapa pode potencialmente ser lenta se for executada com frequência. Caso seja necessário acessar outro componente a cada atualização, por exemplo, isto deve ser feito durante a inicialização do componente e salvo em uma variável de objeto.

Uma discussão válida que poderia ser levantada é o fato de `Transform` não ser um componente e sim uma variável do próprio *game object*. Primeiro, vale dizer que nem todo *game object* faz uso do `Transform`, mesmo uma instância deste existindo para todo *game object*. Algumas exceções, dentro do jogo, são alguns gerenciadores implementados por meio de componentes. Mas essas são exceções, e não regras. A regra geral é que todo *game object* necessite ao menos uma localização na tela. Além disso, como a dependência de todos tipos de componentes pelo `Transform` é bastante freqüênte, prefere-se manter uma referência direta em vez de adicioná-lo ao mapa de componentes.

8.5.3 Categorias de componentes

Para a produção deste jogo, foram definidas cinco famílias de componentes. Cada uma será explicada, bem como seus componentes individuais.

GameLogic

Nesta família estão contidos diversos componentes com todo o código referente à lógica do jogo. Todos os componentes abaixo referidos abaixo são classes que herdam de `GOCGameLogic`.

- `GOCMenuScreen` e `GOCGameScreen` correspondem a componentes controladores das telas, uma abstração para cada estado do jogo. (Esta relação tela–estado não é reforçada, é apenas de convenção.) Cada um desses componentes se torna "pai" de

todos os *game objects* que cria, te tal forma que a destruição do estado implica na destruição dos seus filhos.

- `GOCMoneyBank` é responsável pelo gerenciamento do dinheiro do jogador, escutando mensagens do tipo `MoneyEvent` e alterando a quantia disponível.
- `GOCForest` é responsável pelo gerenciamento da floresta: quando o jogador permite que um certo número de árvores sejam atingidas, o jogo deverá ser encerrado ("game over"). Neste momento, é enviado um evento `GameOver`.
- `GOCPointer` é um componente anexado a um novo *game object* a cada vez que um evento `FingerInputEvent` é recebido com a operação `ADD`. Este componente, que recebe o `sessionID` do toque, o acompanha durante sua vida útil, sendo o *game object* destruído quando o `sessionID` for removido.
- `GOCPollution` é responsável pelo controle de cada poluição ao longo de sua vida útil. Como todas as poluições possuem comportamentos semelhantes, diferindo apenas em alguns parâmetros, todas são implementadas nessa mesma classe e os parâmetros inicializados de acordo com o tipo. Recebe eventos de tipo `FingerInputEvent` e `AttackEvent`, ambos os quais fazem com que a nuvem se destrua. Emite os eventos `MoneyEvent`, `PollutionEvent` e `GOEvent` ao ser destruída.
- `GOCPollutionFactory` possui a principal tarefa de instanciar *game objects* de poluição periodicamente enquanto o jogo estiver ativo. Para tal armazena, dentre outras informações, o *level* atual e informações correspondentes à forma como as poluições devem ser emitidas de acordo com o level.
- `GOCTowerMissile` especifica o comportamento de uma das torres. Esta torre é instanciada a partir do `GOCGameScreen` no momento em que o usuário a seleciona. Periodicamente realiza ataques com mísseis, instanciando *game objects* com o componente `GOCAttackMissile`, que funciona como um míssil teleguiado, seguindo determinado alvo. Este é selecionado durante a inicialização do *game object*. Por fim, complementando o funcionamento das torres, a `GOCTowerMissile` pode ser rotacionada. Enquanto esta funcionalidade estiver acionada, um *game object* com ponteiro para rotacionar será exibido. Seu comportamento está contido em `GOCRotateGizmo`, que realiza a rotação.
- A `GOCTowerSolar`, o segundo tipo de torre presente no jogo, periodicamente faz a varredura de uma área em seu redor destruindo todas as poluições que lá estiverem por meio da `GOCAttackArea`.

Ocorre comunicação entre esses componentes em diversos momentos, por meio dos eventos `AttackEvent` (ataques em área e mísseis), `GameEvent` (quando o jogador perde), `MoneyEvent` (quando ocorre adição ou remoção de moeda virtual), e `PollutionEvent` (quando ocorre uma poluição é destruída).

Por não exigir nenhuma forma de controle central, não foi necessário criar um subsistema residente na `Engine`, já que cada estado de jogo é auto-gerenciável.

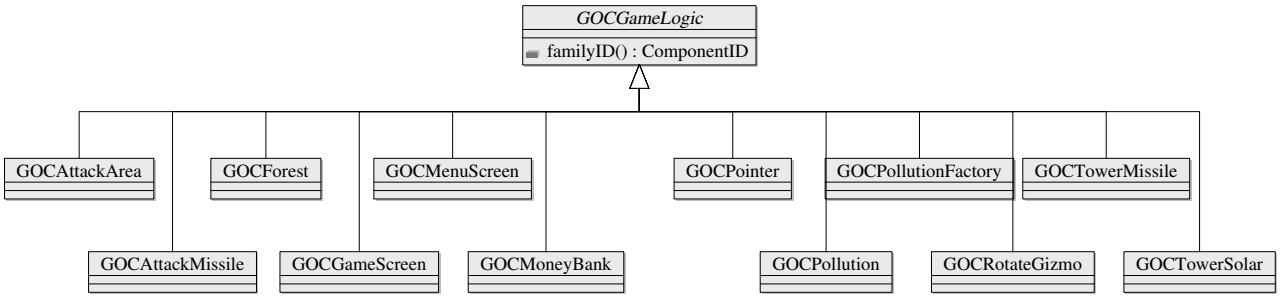


Figura 8.3: Componentes da família `GOCGameLogic`

Graphics

Dentro desta família estão componentes que, quando presentes em um *game object*, fazem com que o mesmo possua representação gráfica. A interface de `GOCRender` define o comportamento que os componentes concretos devem definir. O método `Update()` deverá ser responsável por atualizar a parte lógica, enquanto o método `Render()` efetivamente desenha o objeto na tela. Ambos são chamados a cada atualização, mas em momentos diferentes. O primeiro é invocado pelo próprio *game object* durante a atualização de todos os seus componentes, enquanto o `Render()` é chamado pela atualização do subssistema `Graphics`.

A interface disponibiliza um método `DoesCollide(int x, int y)` que deve retornar um booleano correspondente à colisão de um ponto (x,y) com a representação gráfica em si, e disponibiliza `SetEnabled(bool)` que pode ser usado para ativar e desativar o componente.

O construtor de `GOCRender` registra a instância perante o subsistema `Graphics` emitindo `GraphicsEvent` com operação `REGISTER` em seu construtor, e emitindo `GraphicsEvent` com operação `UNREGISTER` em seu destrutor.

- `GOCRenderTexture` é usado na renderização de texturas, podendo conter mais de um frame. O carregamento das texturas pode ser feito pelo método `setTextures(string filename,` ou pelo construtor, que recebe o nome do arquivo, ou dos arquivos, no caso de uma animação. Caso seja composto por vários frames, o método `Update(int dt)` irá atualizar os frames sendo exibidos. `Render()` exibe o frame na posição definida pelo `Transform` do *game object*.
- `GOCRenderTTF`, similarmente, exibe a textura de texto na tela, no entanto esta é gerada com base no texto a ser exibido. A geração desta textura é feita através da biblioteca FTGL, mencionada anteriormente.

Outra família, `GOCAnimation`, se relaciona com a `GOCGraphics` realizando transições no `Transform` do *game object*. Contém um único componente concreto, `GOCKeyFrameAnimation`. Este contém um `std::vector` de `KeyFrame`s e realiza transições entre estes keyframes. Cada `KeyFrame` contém uma posição, uma escala, uma rotação, um *alpha*, e o tempo de transição daquele `KeyFrame` para o próximo. Opcionalmente, a transição pode ser executada em *loop* e o *game object* pode ser encerrado ao término das transições.

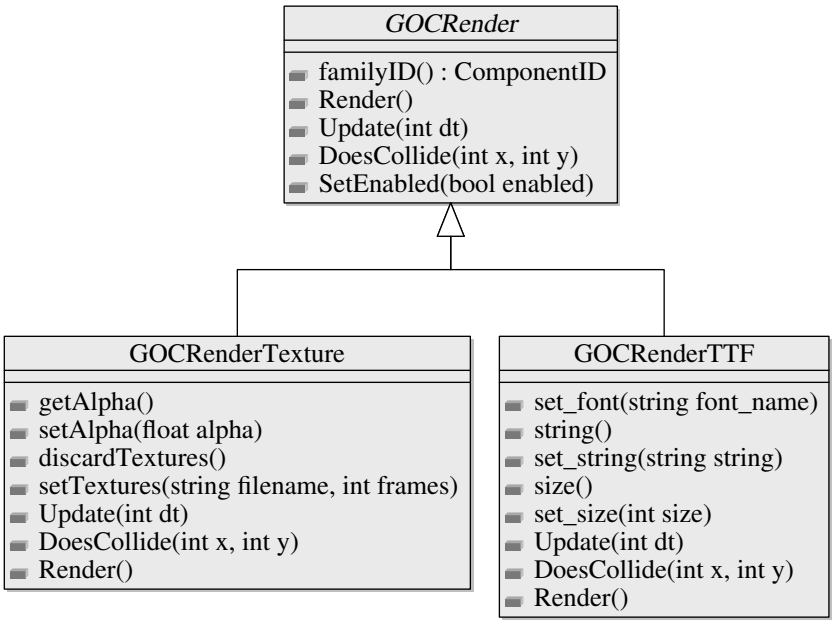


Figura 8.4: Componentes da família `GOCRender`

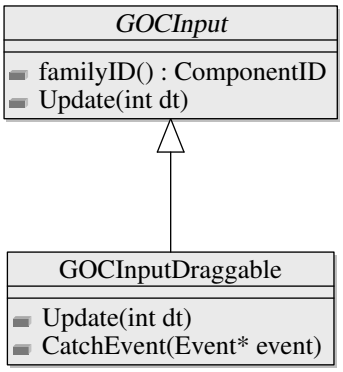


Figura 8.5: Componentes da família `GOCInput`

GUI

Família (`GOCGUI`) referente a objetos que tenham relação com interfaces gráficas (*Graphical User Interfaces*). Contém como subclasse o componente `GOCGUIButton`. Sua funcionalidade é controlar o estado do botão, alterando sua representação gráfica através do componente `GOCRender` e realizando chamadas a um *callback*.

Input

O componente `GOCDraggable` é usado quando se deseja que determinado *game object* acompanhe o movimento do dedo. Para tal, se registra para receber eventos do tipo `FingerInputEvent`. É necessário que o *game object* possua representação gráfica; ou seja, necessita de um componente `GOCRender` para que seja verificada colisão com o mesmo.

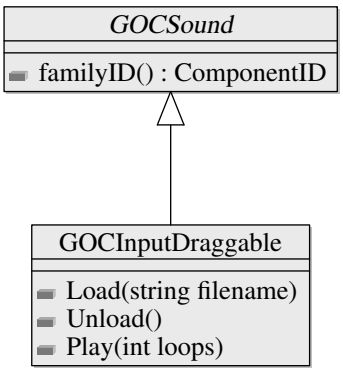


Figura 8.6: Componentes da família `GOCSound`

Sound

Consiste em um único componente, `GOCSoundSample`. Neste, uma amostra de som pode ser carregada pelo método `Load(string filename)`, e esta pode ser executada pelo método `Play(int loops)`, executa `loops` vezes a amostra carregada.

Capítulo 9

Conclusão

O desenvolvimento de jogos eletrônicos ainda não é visto como uma área de pesquisa acadêmica entre os mais puristas da engenharia de *software*. Mas a evolução destes sistemas interativos leva a um estudo aprofundado de diversas técnicas e conceitos de arquitetura, padrões e processos. Sem este estudo, os jogos complexos de hoje se tornariam inviáveis, pois alguns ultrapassam milhões de linhas de código, demandam equipes de mais de 50 profissionais e algoritmos muito eficientes.

Além disso, existem aspectos que são exclusivos aos jogos eletrônicos. São considerados tecnologia, cultura e entretenimento simultaneamente. Assim, precisam evoluir e se atualizar constantemente pois acompanham a evolução tecnológica. Precisam adequar-se às novas gerações para acompanhar a evolução da cultura. Por fim, precisam adequar-se a um público altamente crítico, pois fazem parte da sua própria escolha de lazer e entretenimento. Portanto, a solução chave encontrada por diversos países, é envolver a academia diretamente no processo de pesquisa e inovação, adequando-se a todas esses aspectos próprios dos jogos eletrônicos.

Para acompanhar tal evolução, a pesquisa e o estudo de novas interações homem-máquina são essenciais. Através de estudo e uso de novas tecnologias é possível rediscutir os paradigmas de interação entre jogadores e jogos eletrônicos. Assim, a imersão, o envolvimento e o entretenimento atingem outros patamares.

Estas interfaces naturais podem, também, auxiliar outros aspectos importantes dos jogos na sociedade. O processo lúdico e o desenvolvimento cognitivo podem ser melhor trabalhados em jogos construídos para tais fins, pois os gestos e a fala são partes importantes do desenvolvimento do homem. Os jogos eletrônicos trabalham especificamente na construção do raciocínio lógico e podem, com estas interações mais naturais, também trabalhar o desenvolvimento motor.

A partir da constatação da importância destas áreas de estudo, as próximas seções dissertam sobre melhorias possíveis para o jogo e a mesa multitoque construídos e dissertam sobre trabalhos futuros para pesquisa na área de inovação e melhoria dos processos de desenvolvimento de jogos eletrônicos.

9.1 *Postmortem*

A análise *postmortem* é um método que objetiva o compartilhamento da experiência da equipe a outros grupos [35]. Este processo é frequentemente utilizado em projetos de desenvolvimento de jogos, logo, nesta seção utiliza-se deste método para discutir os problemas e soluções encontrados por este trabalho.

O desenvolvimento de um jogo é uma tarefa complexa. Desenvolver um jogo em uma interface ainda instável pode ser ainda mais trabalhoso. Foram encontradas dificuldades no processo de construção da própria estrutura da mesa, na escolha das bibliotecas e na falta de tempo para desenvolver novas funcionalidades para o jogo.

A mesa multitoque construída para este projeto possui problemas que poderiam ser resolvidos se fossem utilizados equipamentos de melhor qualidade. Devido à dificuldade encontrada aqui na região, os LEDs infravermelhos comprados poderiam possuir melhores especificações quanta à potência e frequência da onda de luz. Devido ao preço do acrílico, não foi possível utilizar uma superfície maior e mais espessa. Para melhorar ainda a detecção de toques, poderia ter-se utilizado iluminação por FTIR. Entretanto isto levaria a uma complexidade maior da estrutura da mesa, pois os LEDs precisam estar perfeitamente alinhados, angulados e distribuídos ao redor da superfície.

Durante o decorrer deste projeto, utilizou-se para detecção dos toques uma biblioteca denominada *ReacTIVision*, descrita no capítulo de trabalho correlatos 2. Entretanto, esta biblioteca não detectava os toques de forma correta, pois nossas estrutura de câmera, mesa e iluminação DI não fornecia uma imagem com qualidade. Para adequar-se às nossas necessidades, foi necessário utilizar outra biblioteca mais específica e menos portável chamada *Touchlib*. Esta, por sua vez, possui filtros que foram capazes de melhorar a resposta dos toques sobre a superfície.

A construção da mesa foi um fator determinante no tempo investido no desenvolvimento do *software* do jogo eletrônico, por isso, algumas funcionalidades não foram implementadas. Para um jogo com muitos recursos, é essencial a elaboração de uma controladora e gestora de texturas, sons e animações. De tal forma que estes sejam carregadas no tempo devido e sejam reaproveitadas por toda a aplicação. Apesar do jogo desenvolvido possuir poucos recursos, uma gerenciadora de recursos é sempre uma solução elegante.

O desenvolvimento de ferramentas que auxiliem a construção do conteúdo do jogo poderia trazer inúmeras vantagens. Isto inclui ferramentas que auxiliam a construção de novas entidades a partir de uma combinação de componentes, que permitem o controle das variáveis do *gameplay* de forma dinâmica e que permitem diversificar a construção de novos cenários para os *levels*.

Um ponto específico do estudo deste trabalho é o sistema orientado a componentes. Este padrão traz uma série de benefícios que, por causa do tamanho pequeno do jogo, não foram aproveitados como poderiam. Primeiramente, este sistema nos permite diversificar mais o *gameplay*, pois rapidamente é possível criar novas entidades, nuvens de poluição, torres e botões. Segundo, seria de grande valia uma adequação do sistema orientado a componentes a um padrão de criação de entidades através de templates em arquivos

(paradigma *data-driven*). Assim poderia-se incluir mais facilmente novas entidades e carregar todas dinamicamente, sem a necessidade de recompilação.

Um fator limitante da escolha da arquitetura foi a necessidade de utilizar um sistema dividido em famílias. Esta escolha nos permitiu realizar algumas tarefas de forma mais fácil. Por exemplo, um componente de lógica pode fazer uma requisição do componente gráfico a partir do nome da família. Assim é possível acessar outros componentes pelo polimorfismo da própria família.

Entretanto, nossa arquitetura só permite um componente de cada família em uma mesma entidade. Logo, algumas combinações não podem ser feitas. Por exemplo, em uma entidade botão, não é possível acoplar um componente `GOCRenderTexture` para desenhar o próprio botão e um `GOCRenderTTF` para desenhar o texto, pois ambos componentes são da mesma família `GOCRender`.

Uma solução simples para este limitante é a iteração através de todos os componentes de uma mesma família. No entanto, outros aspectos do jogo foram tomados como prioridade no processo de desenvolvimento.

Este trabalho permitiu um estudo aprofundado de diversos aspectos da construção de um jogo e de interfaces homem-máquinas. Assim, apesar das dificuldades encontradas, atingiu seu objetivo de discutir inovações neste contexto. Este trabalho pode servir, também, como inspiração e ponto de partida para outras diversas empreitadas.

9.2 Trabalhos futuros

Devido à natureza da indústria do desenvolvimento de jogos, a área de *game design* se sobrepõe a área de programação de jogos. É devido a esse fato que o *game design* naturalmente levou a indústria de jogos a favorecer arquiteturas *data-driven*, pois permite o desenvolvimento de protótipos mais rapidamente e, consequentemente, testes com os novos conceitos de jogos. Uma expansão cabível neste jogo desenvolvido por este projeto é adequar-se a este tipo de arquitetura.

Assim, os dados de jogo são mantidos separados do código e o projeto não cria a necessidade de compilação constante, pois são usadas linguagens interpretadas. Ou seja, permitem que o *game designer* e o programador trabalhem de forma paralela, mas independente. O uso deste tipo de arquitetura favorece a construção de jogos, seja qual for o tipo de sua interface.

Durante a produção deste trabalho, *frameworks* para a construção de aplicativos multimídia para multitoque foram liberados à comunidade desenvolvedora. A utilização de um destes *frameworks* permitiria ao programador concentrar na lógica da aplicação, facilitando a experimentação com tecnologias emergentes, como são as superfícies multitoque. Como um exemplo, vale mencionar o PyMT ([36]), para a linguagem Python. Este *framework* fornece infraestrutura para renderização de interfaces gráficas, recebimento de *input* e processamento de gestos.

Apêndice A

Configuração do Touchlib

A configuração do Touchlib é toda armazenada em um arquivo XML, `config.xml`. Devido à carência de documentação oficial referente ao Touchlib, recomenda-se a consulta ao apêndice B de [6] para compreender o código que se segue.

```
<?xml version="1.0" ?>
<blobconfig distanceThreshold="250" minDimension="2" maxDimension="250"
    ghostFrames="0" minDisplacementThreshold="2.000000" />
<bbox ulX="0.000000" ulY="0.000000" lrX="1.000000" lrY="1.000000" />
<screen>
    <!-- Coordenadas de calibração da superfície -->
</screen>
<filtergraph>
    <dsvlcapture label="dsvlcapture0" />
    <mono label="mono1" />
    <backgroundremove label="backgroundremove2">
        <threshold value="0" />
    </backgroundremove>
    <highpass label="highpass3">
        <filter value="12" />
        <scale value="11" />
    </highpass>
    <scaler label="scaler4">
        <level value="3" />
    </scaler>
    <brightnesscontrast label="brightnesscontrast5">
        <brightness value="0.0980392" />
        <contrast value="0.788235" />
    </brightnesscontrast>
    <rectify label="rectify6">
        <level value="25" />
    </rectify>
</filtergraph>
```

Apêndice B

Fotos

Segue abaixo fotos tiradas da mesa multitoque e do jogo Eco Defense.

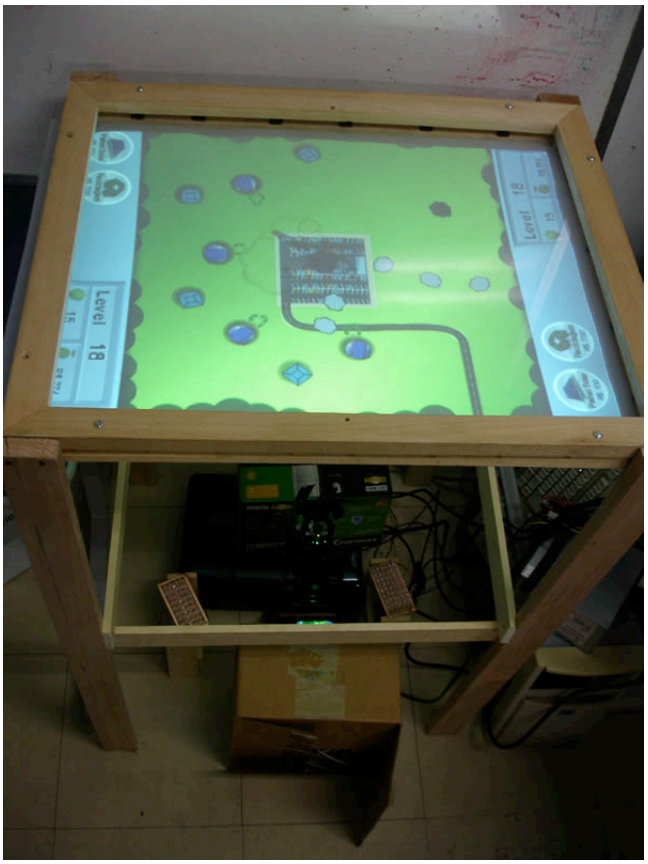


Figura B.1: Mesa em funcionamento.

Figura B.2: Tela inicial do *Eco Defense*.

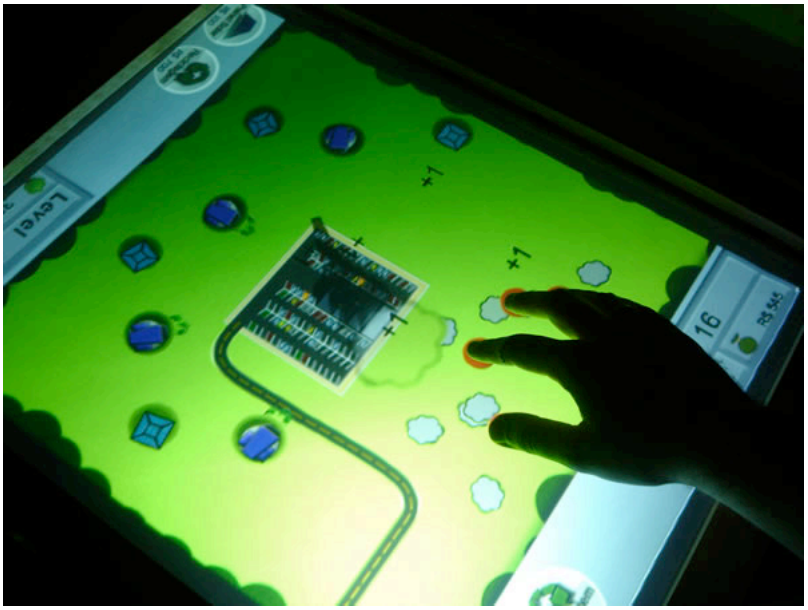


Figura B.3: *Eco Defense* no Level 16.

Referências

[1] J.J. Corso, G. Ye, D. Burschka, and G.D. Hager. A practical paradigm and platform for video-based human-computer interaction. *Computer, IEEE Computer Society*, 41(5):48–55, 2008. x, 1, 3, 4

[2] reactable. `http://mtg.upf.es/reactable/`. x, 5

[3] Illusion labs. `http://www.illusionlabs.com`. Acessado em junho/2009. x, 6

[4] Sparsh UI: A multitouch api for any multitouch hardware / software system. `http://code.google.com/p/sparsh-ui/`. Acessado em junho/2009. x, 1, 7

[5] The Verve project. `http://verveproject.blogspot.com/`. x, 8

[6] L.Y.L. Muller. Multi-touch displays: design, applications and performance evaluation. Master's thesis, Universiteit van Amsterdam, 2008. x, 10, 11, 13, 14, 54

[7] Matthias Rauterberg and Patrick Steiger. Pattern recognition as a key technology for the next generation of user interfaces. *IEEE International Conference on Systems, Man, and Cybernetics*, pages 2805–2810, 1996. 1

[8] Peter Bomark. Visualization and prototyping of a multitouch display device. Master's thesis, Luleå tekniska universitet, 2007. 1, 3

[9] C. Shen, F.D. Vernier, C. Forlines, and M. Ringel. Diamondspin: An extensible toolkit for around-the-table interaction. In *CHI '04: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 167–174, 2004. 1, 5

[10] Verve: Veneral purpose agents. `http://verve-agents.sourceforge.net/`. Acessado em junho/2009. 1, 7

[11] Sergi Jordà, Martin Kaltenbrunner, Gûnter Geiger, and Ross Bencina. The reactable*. In *Proceedings of the International Computer Music Conference (ICMC 2005)*, Barcelona, Spain, 2005. 4

[12] Bricktable. `http://flipmu.com/work/bricktable/`. Acessado em julho/2009. 5

[13] NUI Group – natural user interface group. `http://www.nuigroup.com/`. Acessado em junho/2009. 6

[14] M. Katzourin, D. Ignatoff, L. Quirk, J.J. LaViola Jr., and O.C. Jenkins. Swordplay: Innovating game development through vr. *Computer Graphics and Applications, IEEE*, 26(6):15–19, 2006. 6

[15] Apple inc. iPhone. `http://www.apple.com/iphone/`. Acessado em junho/2009. 6

[16] M. Gardner. Mathematical games - the fantastic combinations of johns conway's new solitarie game life. *Scientific American*, (223):120–123, 1970. 7

[17] C.H. Santos and M. Imenes. Tangram: Um antigo jogo chinês nas aulas de matemática. *Revista de Ensino de Ciências*, (18):42–49, 1967. 7

[18] Walter G. Kropatsch. History of computer vision: A personal perspective. `http://www.icg.tu-graz.ac.at/Members/hgrabner/historyOfCV/kropatsch.pdf`, 05 2008. 9

[19] Touchlib: a library for creating multi-touch interaction surfaces. `http://nuigroup.com/touchlib/`. Acessado em junho/2009. 14, 40

[20] M Kaltenbrunner, T Bovermann, R Bencina, and E Constanza. Tuio: A protocol for table-top tangible user interfaces. In *Proc. of the 6th International Workshop on Gesture in Human-Computer Interaction and Simulation*, 2005. 14

[21] Stacey D. Scott and Sheelagh Carpendale. Interacting with digital tabletops. *Computer Graphics and Applications*, 26(5):24–27, 2006. 16, 17

[22] H. Benko, A. Wilson, and P. Baudisch. Precise selection techniques for multi-touch screens. *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 1263–1272, 2006. 17

[23] Michael Thörnlund. Gesture analyzing for multi-touch screen interfaces. Master's thesis, Luleå University of Technology, 2007. 19

[24] Michael Doherty. A software architecture for games. *University of the Pacific Department of Computer Science Research and Project Journal*, 1(1), 2003. 21

[25] Apostolos Ampatzoglou and Alexander Chatzigeorgiou. Evaluation of object-oriented design patterns in game development. *Information and Software Technology*, 49:445–454, 2007. 22

[26] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, 1995. 22, 23, 24, 25

[27] Paul V. Gestwicki. Computer games as motivation for design patterns. *SIGCSE Bull.*, 39(1):233–237, 2007. 22

[28] Eelke Folmer. Component-based game development. *Lecture Notes in Computer Science*, 2007:66–73, 2007. 25

[29] Steve Rabin, editor. *Introduction to Game Development*. Charles River Media, 2005. 26, 30, 39

[30] SDL: Simple directmedia layer. `http://www.libsdl.org`. Acessado em junho/2009. 39

[31] SOIL: Simple OpenGL Image Library. `http://www.lonesock.net/soil.html`. Acessado em junho/2009. 40

[32] FTGL: A font rendering library for OpenGL. `http://homepages.paradise.net.nz/henryj/code/`. Acessado em junho/2009. 40

[33] TUIO. `http://www.tuio.org/`. Acessado em junho/2009. 40

[34] Chris Stoy. *Game Programming Gems 6*, chapter Game Object Component System. Charles River Media, 2006. 44

[35] A Birk, T Dingsoyr, and T Stalhane. Postmortem: never leave a project without it. *Volume 19, Issue 3, Software, IEEE*, 2002. 52

[36] PyMT: multi touch UI toolkit for pyglet. `http://code.google.com/p/pymt/`. Acessado em junho/2009. 53

[37] A. Agarwal, S. Izadi, M. Chandraker, and A. Blake. High precision multi-touch sensing on surfaces using overhead cameras. *Proceedings of the Second Annual IEEE Workshop on Horizontal Interactive Human-Computer System*, pages 197–200, 2007.

[38] C. Sousa and M. Matsumoto. Study on fluent interaction with multi-touch in traditional gui environments. *Proceedings of the TENCON 2007 - 2007 IEEE Region 10 Conference*, pages 1–4, 2007.

[39] J. Kim, J. Park, and H. Lee. Hci (human computer interaction) using multi-touch tabletop display. *Proceedings of the IEEE Pacific Rim Conference*, pages 391–394, 2007.

[40] J. Han. Low-cost multi-touch sensing through frustrated total internal reflection. *Proceedings of the 18th annual ACM symposium on User interface software and technology*, pages 115–118, 2005.

[41] K.C. Dohse, Thomas Dohse, Jeremiah D. Still, and Derrick J. Parkhurst. Enhancing multi-user interaction with multi-touch tabletop displays using hand tracking. *First International Conference on Advances in Computer-Human Interaction*, pages 297–302, 2008.

[42] D. Voth. Evolutions in gaming. *Pervasive Computing, IEEE Computer Society*, 6(2):7–10, 2007.

[43] OpenCV: Open Source Computer Vision. `http://opencvlibrary.sourceforge.net/`. Acessado em junho/2009.

[44] reacTIVision: a tangible multi-touch interaction framework. `http://reactivision.sourceforge.net`. Acessado em junho/2009.

[45] Scott Bilas. A data-driven game object system. In *Game Developers Conference Proceedings*, 2002.