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

Gear2D: um motor extensível de jogos baseado em componentes

Leonardo Guilherme de Freitas

Brasília
2013

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

Gear2D: um motor extensível de jogos baseado em componentes

Leonardo Guilherme de Freitas

Monografia apresentada como requisito parcial
para conclusão do Curso de Computação — Licenciatura

Orientador
Prof. Dr. Rodrigo Bonifácio

Coorientadora
Prof.ª Dr.ª Carla Denise Castanho

Brasília
2013

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Curso de Computação — Licenciatura

Coordenador: Prof. Dr. Flávio de Barros Vidal

Banca examinadora composta por:

Prof. Dr. Rodrigo Bonifácio (Orientador) — CIC/UnB
Prof.ª Dr.ª Fernanda Lima — CIC/UnB
Prof.ª Dr.ª Carla Denise Castanho — CIC/UnB

CIP — Catalogação Internacional na Publicação

de Freitas, Leonardo Guilherme.

Gear2D: um motor extensível de jogos baseado em componentes / Leonardo Guilherme de Freitas. Brasília : UnB, 2013.
54 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2013.

1. Motores de Jogos, 2. Reuso, 3. Gear2D

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

Gear2D: um motor extensível de jogos baseado em componentes

Leonardo Guilherme de Freitas

Monografia apresentada como requisito parcial
para conclusão do Curso de Computação — Licenciatura

Prof. Dr. Rodrigo Bonifácio (Orientador)
CIC/UnB

Prof.ª Dr.ª Fernanda Lima
CIC/UnB

Prof.ª Dr.ª Carla Denise Castanho
CIC/UnB

Prof. Dr. Flávio de Barros Vidal
Coordenador do Curso de Computação — Licenciatura

Brasília, 22 de Julho de 2013

Dedicatória

A minha mãe e ao meu amor, que sempre acreditaram em mim.
Aos verdadeiros amigos, vocês sabem quem são.
Para o totó/suco de pneu/sopa de piche/vitamina de carvão, amigo Sansão (eu sei que você só rasgaria esse monte de papel).

Agradecimentos

Aos professores Carla Castanho, Guilherme N. Ramos e Rodrigo Bonifácio pela paciência e orientação. Foi uma honra tê-los como orientadores. Agradeço ao professor José Carlos Loureiro Ralha, que, com seu jeito descontraído, me deu a oportunidade de compreender que a programação é interessante e motivadora. Talvez eu não me tornasse metade do programador que acho que sou se não fossem suas aulas. Agradeço ao meu amigo Thiago Coelho Vieira ("Thiago Piqueno"), por ter me dado a ideia genial e óbvia de usar a *Gear2D*, que era um *hobbie*, como projeto de graduação. Agradeço aos colegas e amigos Luiggi Monteiro Reffatti, Igor Rafael de Sousa *("Bicão")* e Anderson Campos Cardoso ("*"), por terem acreditado no meu trabalho e terem embarcado no navio, me ajudando a remar; me orgulho de saber que pude trabalhar junto de vocês. Guilherme Costa ("Shrek"), Matheus Pimenta, Tiago Galvão ("Spike"), Davi Silva ("Doom"), Antonio Martino e Vanessa Porto, obrigado por seu esforço e (muita) paciência no desenvolvimento do *Naval Warfare* (e desulpem os *bugs*, já estou corrigindo :). Agradeço à professora Suzete Venturelli e à equipe do Mídia-Lab pela aprendizagem e amizade. Minha amiga Juliana Hilário ("Jurema"), obrigado por não rir (muito) de mim quando fiquei sem cabelo. Érica Cristina Moreno ("Ericola") e Victor de Souza Bonfim ("Bicuds"), muito obrigado pelo incentivo e pela tremenda amizade durante todos esses anos.

Para aqueles que a memória, cansada por causa dessa tal de monografia, não permitiu lembrar, Meu grande obrigado por fazerem parte da minha vida e por me perdoarem ter esquecido de agradecer-lhes apropriadamente.

Resumo

Motores de jogos elevam o nível de reuso na criação de jogos eletrônicos, centralizando em *APIs* coerentes com o domínio as funções comumente utilizadas. Entretanto, utilizando os motores atuais, a arquitetura de muitos jogos é construída modelando as entidades através de hierarquias de classes, o que pode ocasionar conflitos de nomes e/ou duplicação de código. Motores baseados em componentes minimizam esses problemas, contudo, devido ao forte acoplamento entre os componentes que pode ser introduzido quando é usado acesso direto para comunicação entre eles, ainda são pouco flexíveis no que tange à extensão e adaptação de entidades em tempo de execução. *Gear2D* é um motor de jogos que segue uma abordagem diferente, provendo adaptação dinâmica de entidades através de componentes desacoplados. Esse *design* resulta numa maior flexibilidade ao se criar jogos, característica discutida através do desenvolvimento de dois componentes não triviais, *Pathfinder2D* para busca de trajetória e *lua-proxy* para suporte a *scripting*, e através de um relato do processo de desenvolvimento do jogo *Naval Warfare* com um time de cinco desenvolvedores.

Palavras-chave: Motores de Jogos, Reuso, Gear2D

Abstract

Game engines boost software reuse during development activities by centralizing commonly used domain abstractions within a set of coherent application programming interfaces. Many games and game engines focus on class hierarchies, which is the intuitive way to model entity taxonomies but may lead to naming conflicts and code duplication. Component based architectures, conversely, minimize this problem by exposing features through components instead. However, due to strong coupling that can still be introduced when using direct access to provide component communication, dynamic entity adaptability may be hindered. Gear2D is a game engine that uses a different approach, providing dynamic entity adaptability through decoupled components. This design results in greater flexibility for creating games, a characteristic discussed through the development of two nontrivial components: an AI pathfinder and a Lua proxy, and through a report of the development process of a game called Naval Warfare with a team of five developers.

Keywords: Game Engine, Reuse, Gear2D

Sumário

Lista de Figuras

Listagens

Capítulo 1

Introdução

Refletindo sua grande popularidade entre crianças, jovens e adultos, os jogos eletrônicos impulsionam um ramo do entretenimento que já é mais lucrativo que o cinematográfico [6]. São caracterizados pela seu alto grau de imersão, criatividade, interatividade e capacidades multimídia.

Assim como todo software, o processo de desenvolvimento de jogos eletrônicos é permeado por diversas ferramentas, como *frameworks*, *SDKs* (*Software Development Kits*), plataformas, aplicativos de criação de mídias (som, imagem, vídeo), etc. Dentre elas, o motor de jogos representa a fundação de um jogo. Como o motor de um veículo, é artefato essencial não só no processo de construção, mas influencia também nas sensações do jogador.

Surgindo de maneira espontânea no processo de desenvolvimento de um jogo ou sendo claramente delineado previamente, o motor de jogos é a peça fundamental de software que qualquer jogo dependerá[1], pois é base do desenvolvimento do *gameplay*[2]: são responsabilidades do motor comunicar com a plataforma em que o jogo será executado, coletar entrada de dados, reproduzir recursos multimídia, etc. Outra responsabilidade do motor é traduzir essas funcionalidades em termos de uma *API*[3] para que possam ser utilizadas pelo desenvolvedor.

No contexto do desenvolvimento de software, motores de jogos introduziram um nível maior de reuso e flexibilidade nas arquiteturas dos jogos eletrônicos; o que antes era construído para cada jogo diferente, passou a ser centralizado e incorporado num conjunto de *APIs* e *frameworks* consistentes com o domínio, permitindo que desenvolvedores se concentrem na criação do jogo propriamente dito, sem se preocuparem com outros aspectos que não fazem parte dos requisitos funcionais ou do *game design*.

Em alto nível, um motor de jogos, quando isolado de um jogo, pode ser considerado um software de prateleira (ou em inglês *Commercial off the Shelf Software* - COTS), permitindo que possa ser reusado em diferentes produtos. Por outro lado, os motores de jogos, em geral, apresentam para o desenvolvedor uma arquitetura monolítica; entidades *in-game* são definidas através de relações de alto acoplamento e dependência, como he-

[1] Como veremos adiante, algumas arquiteturas de jogos não fazem clara separação entre o que é o motor e o que é o jogo propriamente dito. Nesses casos, o código do motor e código do jogo são entrelaçados e indivisíveis.

[2] Jogos eletrônicos também podem ser considerados "simulações", pois agem de acordo com um conjunto de regras que modelam parte de uma realidade - mesmo que uma realidade imaginária [16].

[3] *Application Programming Interface.* - Interface para a programação de aplicações

rança[4] [4]. Atribuir características às entidades dos jogos com o uso de hierarquias de classes muitas vezes requer herança múltipla [42], o que está associada a problemas como colisões de nomes, combinação de métodos, e herança repetida [39].

Além disso, quando a taxonomia das entidades se torna longa, gerenciar a inserção de novos tipos de entidades que possuam características de duas famílias distintas se torna uma tarefa difícil mesmo com a adoção de recursos linguísticos que simulam herança múltipla através de *Mixins* [2] ou *Traits* [34], recursos presentes em linguagens pouco usadas na construção de motores de jogos, ou com a possibilidade de implementação de múltiplas interfaces (como suportado pela linguagem Java [15]).

Para contornar esse problema, alguns motores de jogos apresentam aos desenvolvedores uma arquitetura baseada em componentes, tais como Unity3D [40], Push Button Engine [29], entre outras. Nesse caso, entidades são definidas através da agregação de componentes ao invés de herança. Por exemplo, características associadas à dinâmica de corpo rígido, renderização, reprodução de áudio e outras comuns a todas as entidades ficam disponíveis através de componentes específicos e configuráveis.

Esta disposição permite uma grande flexibilidade na criação de novas entidades. Por outro lado, o uso de motores que permitem referência direta entre componentes (como a Unity3D) não oferecem bons mecanismos para adaptar e estender os jogos dinamicamente, uma característica relevante para diferentes tipos de jogos, como os jogos educacionais que devem se adaptar às ações dos jogadores e os jogos online em massa (*Massive Multiplayer Online Games* – MMOG).

Esse trabalho apresenta a *Gear2D*, um motor de jogos de propósito geral baseado em componentes, concebido para promover a flexibilidade e extensibilidade de jogos em tempo de execução. No Capítulo 2 é apresentado a arquitetura geral dos jogos eletrônicos e como motores de jogos emergiram naturalmente dessas arquiteturas além de uma análise de diversas arquiteturas de motores de jogos (e seus problemas) e como podem influenciar no processo de desenvolvimento e manutenção de um jogo. As decisões tomadas para a arquitetura e implementação do motor Gear2D são apresentados no Capítulo 3, bem como os primeiros jogos criados em caráter de teste da ferramenta. No Capítulo 4 será discutido a flexibilidade e extensibilidade promovida pelo modelo de componentes da Gear2D, comprovada através da criação de novos componentes não-triviais (*lua-proxy* e *Pathfinder2D*), juntamente com um relato do processo de criação do jogo *Naval Warfare*, com o propósito de averiguar a viabilidade da utilização do motor por pequenas equipes em um jogo com características realistas, indo além de simples protótipos ou *toy examples*. Por último, este trabalho apresenta os resultados e impressões obtidas através de todos os estudos de caso efetuados, extraindo dados de sua utilização que são relevantes para a implementação de novas funcionalidades e aperfeiçoamento do motor (Capítulo 5).

[4]Por exemplo: `http://irrlicht.sourceforge.net/forum/viewtopic.php?f=19&t=26852`

Capítulo 2

Arquitetura dos Jogos Eletrônicos

Jogos eletrônicos são aplicações multimídia interativas e imersivas. Seu fluxo de execução depende das decisões do usuário sobre quais botões são pressionados ou movimentos realizados com o *joystick*, teclado ou outro dispositivo de entrada. O que é mostrado ao jogador através de sons e imagens é resultado dessa entrada interpretada através das regras de *gameplay* [16].

Existem, então, três fases principais que podem ser definidas em um jogo enquanto aplicação interativa:

- entrada de dados (*input*);
- interpretação da entrada (*simulation*);
- apresentação de resultados (*feedback*).

Quando são executadas repetidamente, criam a ilusão de fluidez e movimento. Essa repetição é chamada de *game loop* e cada iteração recebe o nome de quadro (em inglês, *frame*).

2.1 O *Game Loop*

Embora a interatividade de um jogo aconteça dentro da fase de simulação, existem medidas a serem tomadas para preparar e garantir a execução correta do *game loop*. Isso é, recursos devem ser inicializados, como vídeo e audio, *assets*[1] devem ser carregados e identificados para referência posterior, memória suficiente deve ser alocada, entre outros. Esse código é executado apenas uma vez, como indicado no segmento de código C++ usando a biblioteca SDL (Listagem 2.1, linhas 1–6).

Na entrada de dados, é executada a captura de todos os eventos dos dispositivos de entrada e a representação de seus estados atuais, isso é, quais botões estão pressionados, quais não estão, em que posição está o cursor do mouse, entre outros. Nessa fase do *frame* também pode ocorrer a captura de dados através de outros canais de entrada, como rede ou eventos do sistema. Geralmente essa fase é expandida para compreender toda a captura de todos os eventos a serem interpretados.

[1] Recursos auxiliares que residem fora do código fonte como definições de mapa, de entidade, recursos multimídia, etc

```
1   void run(list<Entidade *> & entidades) {
2         bool playing = true;
3         SDL_Event ev; /* evento a ser processado */
4         list<Entidade *>::iterator e;
5
6         /* game loop */
7         while (playing) {
8
9           /* input */
10          while (SDL_PollEvent(&ev) != 0)
11            for (e = entidades.begin(); e != entidades.end(); e++)
12              *e->input(&ev);
13
14          /* simulation */
15          for (e = entidades.begin(); e != entidades.end(); e++)
16            *e->update();
17
18          /* output/feedback */
19          for (e = entidades.begin(); e != entidades.end(); e++)
20            *e->render();
21        }
22  }
```

Listagem 2.1: Código em alto nível de um game loop básico utilizando a biblioteca multimídia SDL para a captura de eventos exibindo as três fases principais de um game loop.

Quanto ao tratamento de eventos, pode acontecer através de notificações ou através de *queries*. Na forma apresentada nas linhas 10-13 da Listagem 2.1 cada entidade é avisada de um novo evento. Entretanto, a forma atual deixa espaço para otimizações, considerando que todas as entidades são informadas de todos os eventos que forem capturados, mesmo que não estejam interessadas. Técnicas mais elaboradas envolvem o uso do padrão *Observer* [14].

Para o tratamento através de *queries*, entidades não são explicitamente avisadas de eventos ocorridos, mas indagam ao sistema de eventos se houve algum evento que lhes interessa ou se algum dispositivo de entrada está em algum estado desejado (por exemplo, se a tecla `ESC` foi pressionada). Nesse cenário, deve haver um centralizador de eventos que mantém o estado dos dispositivos de entrada e os eventos ocorridos.

Na fase de simulação, os eventos de entrada recebem um significado dentro do contexto do jogo, sendo utilizados para influenciar o estado das entidades e objetos presentes no mundo virtual (Listagem 2.1, linhas 15–18). Cada entidade registrada é mantida numa lista ou em alguma estrutura de dados que permita iteração, e, uma a uma, tem sua chance de executar os procedimentos para interpretar o significado de cada evento ocorrido.

Para entidades controladas pelo computador, essa fase é a ideal para executar algoritmos de *path-finding*, resoluções de qual ação tomar, mudança de comportamentos baseados em máquinas de estados, etc.

Uma vez computado o estado de cada entidade, o jogo pode produzir *feedback*, posicionando recursos de vídeo, reproduzindo sons ou disparando outros dispositivos de retorno [30] para devolver ao jogador as consequências de suas escolhas. No código da Listagem 2.1, as linhas 20-21 são destinadas a produzir retorno visual, deixando que cada entidade execute o código necessário para se fazer visível.

Através das três fases principais em um *game-loop* se configura a experiência de *gameplay*. Este ciclo de entrada–simulação–*feedback* dita qual a sensação que o jogador vai ter ao fazer curvas em um veículo a 250 Km/h, por exemplo.

É possível concluir que a experiência de um jogador é determinada principalmente na fase de simulação, que deve ser então o conjunto de regras de interatividade, definindo de maneira distinta um jogo eletrônico, dado que é a fase em se interpreta os eventos e se configura o *feedback*.

As outras fases, embora cruciais, são executadas de maneira comum a todos os jogos. Por exemplo, um jogo de corrida e um jogo de luta podem utilizar os mesmos dispositivos de entrada (ex: *joystick*) e os mesmos dispositivos de saída, mas são totalmente diferentes em sua experiência de jogo.

Como são comuns, entrada de dados e a apresentação/*feedback* podem ser isoladas e abstraídas em uma biblioteca reusável, permitindo que a criação de um jogo possa se resumir no uso dessas abstrações para a criação das regras que definem a experiência do jogador (*gameplay*) e na produção dos recursos multimídia para serem utilizados na representação do mundo virtual.

2.2 Como emergem motores de jogos

Usualmente a arquitetura de um jogo é decomposta entre em módulos específicos da plataforma[2] e módulos específicos para o jogo, incluindo submódulos de cada parte. Essa distinção muitas vezes não é clara e dependendo da arquitetura adotada na criação do jogo, ocorre o entrelaçamento das preocupações, fazendo que não haja uma clara separação entre os componentes e impedindo o reuso do código da plataforma em outro jogo ou a adoção de uma plataforma diferente com mínimas mudanças no código fonte. Entretanto, a medida que a complexidade dos jogos aumenta, arquiteturas que reduzam o custo de manutenção e adaptação se tornam mais atraentes, como a modular ou camadas.

Idealmente, a arquitetura adotada por um jogo deve ser tal que alterações em um ponto do código não provoquem alterações em outros pontos. Em outras palavras, deve-se buscar o desacoplamento e a separação entre código que gera *gameplay* e o código que lida com a plataforma e dispositivos de E/S. Mesmo esse último deve estar organizado em tal arquitetura que o torne fácil de manter [16].

O termo "motor de jogos" emergiu em meados dos anos 90 [16], denotando a tecnologia que permitia que jogos pudessem ser modificados com conteúdo criado por usuários; tinham portanto uma clara separação entre o código do jogo e o código da plataforma. Para Jason Gregory [17], o nome "motor de jogos" (em inglês *game engine*) denota a porção do software usado em um jogo que pode ser reusado, servindo de fundação para a criação de outros jogos sem grandes alterações.

[2] Assume-se plataforma como o conjunto Hardware + Sistema Operacional onde o jogo será executado

Em geral, motores de jogos servem um gênero específico (por exemplo, motores de jogos de luta, motores de FPS[3]), expondo as funcionalidades mais usadas em suas criações; entretanto existem motores de propósito mais geral. Podem ser ainda classificados por sua representação do espaço virtual (2D ou 3D) e/ou plataforma alvo (PC, Mac, Mobile, Web, etc), entre outros.

Podem ser considerados tanto *middleware* quanto *framework* [25], tratando peculiaridades da plataforma em que o jogo está sendo desenvolvido, provendo funcionalidades como carga e manipulação de recursos (*assets*) de audio e vídeo, expondo interfaces para as tarefas rotineiras e outras utilidades. Sendo *framework*, um motor deve ser, por natureza, incompleto, no sentido que teve ser instanciado com o código e funcionalidades necessárias para criar o jogo desejado [38], ou seja, um motor enquanto *framework* deve ter espaço para ser especializado com as futuras regras de *gameplay*.

A interface do motor de jogos apresentada ao desenvolvedor para que se seja especificado o comportamento de unidades tipicamente acontece através de uma classe abstrata, para que seja implementada pelo programador.

Tomando como base a Listagem 2.1, é possível sugerir que o código apresentado pode fazer parte de um motor de jogos - representa uma porção de alto-nível e geral dos jogos, pois não há código específico de nenhum jogo. Vemos que por si só não define nenhum comportamento, mas sim dá espaço para que cada entidade se comporte conforme descrito pelo game design. Ainda, levando em consideração o paradigma orientado a objetos, há de se perceber que qualquer jogo criado usando tal código vai girar em torno da classe *Entidade*, mas não é claro a interface a qual deve se sujeitar o desenvolvedor que utilizará esse motor.

```
1 class Entidade {
2   public:
3     virtual void input(const SDL_Event * ev) = 0;
4     virtual void update() = 0;
5     virtual void render() = 0;
6 }
```

Listagem 2.2: Possível código para uma interface abstrata *Entidade* para definição de entidades de *gameplay*.

A Listagem 2.2 representa uma abordagem para a interface de um *framework* para a construção de um jogo utilizando o código apresentado na Listagem 2.1. Entidades que desejam fazer parte do *game loop* devem se adequar à classe *Entidade*, usando o mecanismo de herança, como por exemplo na Listagem 2.3. Se estiver separado do código de game loop definido na Listagem 2.1 e da interface *Entidade* definida na Listagem 2.2, temos, distintamente, código do jogo e código do motor de jogos.

Na Listagem 2.4 é apresentada uma possível utilização das classes *Inimigo* e *Chefe* para definir um cenário inicial do jogo e da função *run()* para dar início ao *game-loop*. Perceba que as partes específicas do jogo foram mantidas em separado das partes do código do motor, permitindo a sua reutilização em outros jogos. Por outro lado, se as classes *Inimigo* e *Chefe* fossem utilizadas dentro da fase de inicialização na função *run()*, seriam necessárias alterações no motor sempre que necessário introduzir uma nova entidade no jogo.

[3]First Person Shooter - jogos de tiro em primeira pessoa

```
class Inimigo : public Entidade {
        public:
          /* Sobrescrever update() e render() */
}

class Chefe : public Inimigo {
        public:
          /* Sobrescrever update() e render() */
}
```

Listagem 2.3: Classes *Inimigo* e *Chefe* valendo-se de herança para definir uma hierarquia de inimigos de um jogo.

```
#include "engine/entidade.h" /* interface Entidade */
#include "engine/engine.h"   /* run() */
#include "jogo/inimigos.h"   /* classes Inimigo e Chefe */
int main(int argc, char ** argv) {
        /* lista de entidades para ser usada pela engine */
        list<Entidade *> entidades;

        /* adiciona inimigo e boss */
        entidades.push_back(new Inimigo());
        entidades.push_back(new Chefe());

        run(entidades);

        return 0;
}
```

Listagem 2.4: Possível definição de um jogo com uma entidade *Inimigo* e uma entidade *Chefe*.

2.3 Arquiteturas baseadas em herança

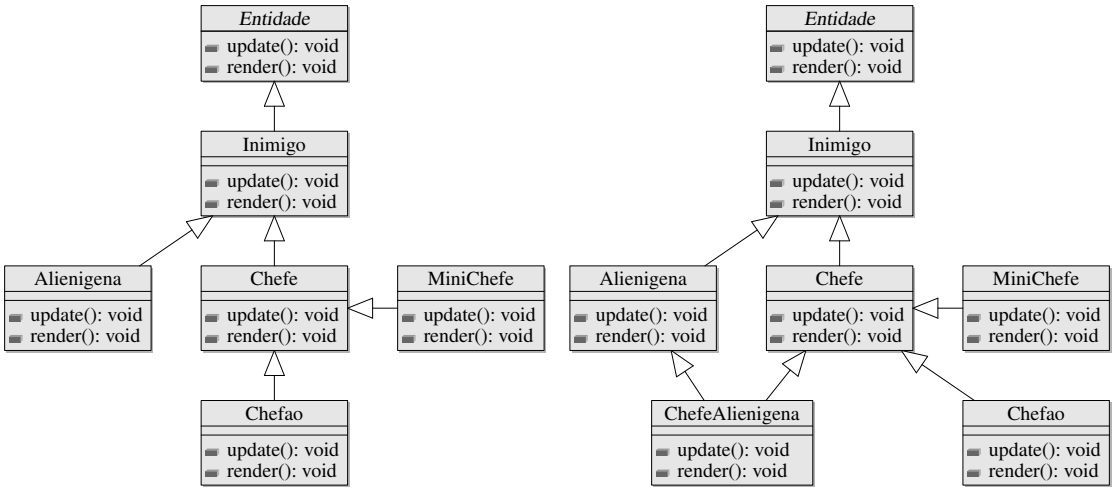
Motores de jogos baseados em herança esperam que as entidades de um jogo sejam definidas através de subtipos, exportando classes de entidades pré-prontas para serem usadas pelo desenvolvedor. Por exemplo uma classe *NPC*[4] para personagens não-jogáveis, *Veículos* para objetos que se comportam como veículos, etc, fariam parte da *API* pública do motor, representando uma classificação em hierarquia dos possíveis comportamentos.

Através do código apresentado nas listagens 2.1, 2.2, 2.3 e 2.4, pode-se observar bases para um motor de jogos e um possível jogo usando esse motor. Inicialmente, a *API* definida nos força a implementar a classe *Entidade* se quisermos inserir alguma entidade no *game-loop*. Criamos então a classe *Inimigo*, responsável por definir o código que seria executado por um inimigo do herói do jogo. Por exemplo, detectada a proximidade do herói, o código em *update()* decidiria que deve atacar. Em seguida, foi definida a classe

[4]do Inglês *Non-Playable Character* ou Personagem Não Jogável

Chefe, que seria uma versão mais forte do inimigo, que, por exemplo, seria mais rápido e teria uma aparência mais assustadora.

Definir as entidades através de herança é uma alternativa natural, pois, claramente, uma entidade *Chefe* é um *Inimigo* que, por sua vez, é uma *Entidade* [30, 31]. Podemos nos valer da herança para definir uma hierarquia maior de inimigos, como na Figura 2.1(a).



(a) Classes de vários inimigos

(b) A classe *ChefeAlien* representa um caso de herança múltipla

Figura 2.1: Hierarquia de classes para inimigos (a) e herança múltipla (b)

Poderíamos, inclusive, querer criar um novo tipo de inimigo, um chefe alienígena que o jogador deve derrotar. Gostaríamos de dar a ele o comportamento de um chefe (implementado pela classe *Chefe*) e de um alienígena (classe *Alien*). Podemos então criar uma classe *ChefeAlien* como na Figura 2.1(b), configurando uma relação de herança múltipla das classes *Alien* e *Chefe*. Embora o uso de herança possa parecer interessante para modelar entidades, alguns problemas podem emergir ao longo do caminho, para hierarquias maiores e complexas. Das relações possíveis entre classes, a herança é a de maior acoplamento, produzindo dependência de assinatura de métodos, nomes de classes, campos de dados, etc [30]. Classes filhas ficam atreladas à assinatura da classe pai, e uma mudança na classe pai provoca mudanças nas classes filhas. Por exemplo, se houver mudanças na assinatura do método *input()* na classe *Entidade*, no modelo apresentado na Figura 2.1(b), seis classes devem ser alteradas para se adequarem. O exemplo dado talvez não consiga demonstrar a real gravidade do problema: alterar seis pequenas classes não parece uma tarefa difícil, mas imagine uma hierarquia complexa com centenas de classes. O processo de refatoração de grandes hierarquias acopladas e com entidades dependentes certamente não é desejável no processo de desenvolvimento de um jogo, pois demanda tempo.

Além de alto acoplamento, hierarquias de classes são estáticas. A hierarquia definida no código fonte se torna imutável[5] assim que se torna código executável. Quando a implementação de um jogo depende da hierarquia de classes, adaptar heranças em tempo

[5]Existem algumas linguagens que permitem adaptar heranças em tempo de execução, mas são pouco utilizadas no desenvolvimento de jogos

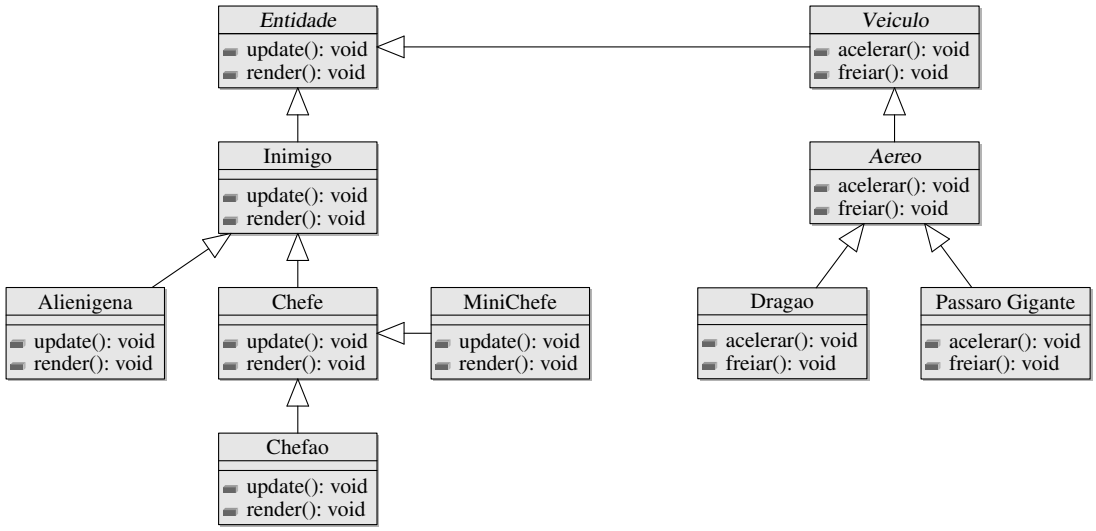


Figura 2.2: Diagrama de classe representando um possível mapeamento de tipos de veículos

de execução é impossível. Por outro lado, adaptabilidade é uma característica desejável em jogos eletrônicos, por que envolvem ações que não são totalmente previsíveis. Uma arquitetura que permite a adaptação de entidades certamente evitará o uso de herança.

Por exemplo, imagine que estendemos a hierarquia da Figura 2.1(a), adicionando um inimigo de classe *Dragon*, em seguida, mapeamos as entidades que servem como veículo para o jogador, como na figura como na Figura 2.2. O que acontece se quisermos que, depois que um inimigo dragão, instância da classe *Dragon*, seja derrotado, se torne um veículo e permita ser montado? Em outras palavras, queremos que uma instância em particular saia da família de entidades (herdeiras da classe *Entidade*), em especial da hierarquia de inimigos e se transfira para a hierarquia de veículos.

Usando uma linguagem como C++ ou Java essa mudança é impossível, tanto no nível do objeto (como o exemplo dado) quanto no nível da classe. Isto é, não é possível em tempo de execução refatorar hierarquias de classes, deixando como solução a duplicação de código (criar uma nova classe *Dragon* na hierarquia de *Veiculo* e copiar código da classe homônima na hierarquia de *Entidade*) ou a herança múltipla (a classe *Dragon* herda, ao mesmo tempo, as classes *Inimigo* e *Veiculo*).

Embora dê mais flexibilidade ao desenvolvedor, a herança múltipla pode causar problemas; particularmente um conhecido como o Problema do Diamante [24]. Esse caso pode ser observado na Figura 2.1(b), e causa ambiguidades. Se ambas as classes *Alienígena* e *Chefe* sobrescreverem o método *input()*, mas a classe *ChefeAlien*, que herda tanto de *Alienígena* quanto de *Chefe*, não o faça, no *game-loop* apresentado no código da Figura 2.1, qual método realmente será chamado na fase de entrada de dados? O método *input()* de *Alienígena* ou *input()* de *Chefe* [31]?

No caso de ambiguidade, alguns compiladores se recusarão a seguir adiante deste ponto[6]. A solução imediata desse problema, em C++, é indicar explicitamente de qual classe o método *input()* deve ser chamado, resultando em código específico do jogo permeando o código da *game engine*.

[6]Esse é o comportamento padrão usando o compilador GCC versão 4.4.5

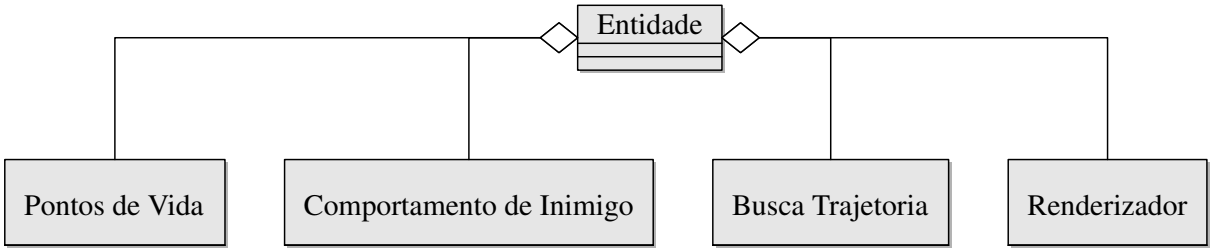


Figura 2.3: Exemplo de entidade usando componentes: cada um dos componentes adicionados representa uma característica ou comportamento.

Outro problema é que o código da classe *Inimigo* vai aparecer duas vezes na classe *ChefeAlien*. Se a variável `health` for definida na classe *Inimigo*, ambas as classes *Alien* e *Chefe* terão essa variável. Na classe *ChefeAlien*, `health` aparecerá duplicado. Além desses problemas, existe um outro, considerado um *anti-pattern* chamado de *The Blob*. É observado em hierarquias grandes onde muitos objetos compartilham funcionalidades semelhantes. Suponha uma hierarquia grande de veículos e inimigos, com sub-hierarquias de veículos voadores e terrestres. Devemos agora inserir o inimigo do tipo *Dragon*, que também é um veículo tanto terrestre quanto voador. Em qual parte da hierarquia devemos inserir essa classe?

A solução geralmente é alterar o código de uma classe pai das hierarquias em questão, inserindo código que nunca é diretamente usado pela própria classe mas sim pelas suas classes herdeiras completamente distintas (por exemplo, inserir código da hierarquia de Veículos e Inimigos na classe *Entidade*), produzindo o *The Blob*.

2.4 Arquiteturas baseadas em componentes

Diante dos problemas que surgem quando a arquitetura de um jogo é baseada em herança, uma outra arquitetura que melhor permite a extensão e adaptação das entidades se faz necessária. Em contraste com a hierarquia de comportamentos, podemos introduzir uma arquitetura baseada em componentes [31, 42]: entidades são construídas através de composição de funcionalidades, que por sua vez são definidas através de componentes.

A relação "é um", representada pela utilização de herança, é desconstruída e se transforma numa relação do tipo "tem um". Por exemplo, na Figura 2.3, a entidade *Inimigo* possui os componentes *Pontos de Vida*, *Comportamento de Inimigo*, *Busca Trajetoria* e *Renderizador*. Se o componente *Comportamento de Inimigo* implementa um "comportamento de inimigo", para transformar essa entidade em um *Alienígena*, bastaria adicionar, por exemplo, o componente *AlienBehaviour*.

Podemos estender o código apresentado no Capítulo 2 com a classe base para componentes, como apresentado na Listagem 2.5. Na Listagem 2.6 estão definidos dois componentes que podem ser adicionados às entidades: *Pontos de Vida*, por exemplo, trataria dos pontos de vida de um personagem (quantos restam, se o personagem morreu, se está envenenado, etc), enquanto o componente *Renderizador* implementaria funcionalidades de renderização.

Definimos então entidades em termos de componentes, utilizando mecanismos padrões de composição (atributos de instância definidas em uma classe), como na Figura 2.7.

```
class Componente {
  public:
    /* Entidade container */
    Entidade * container;

    /* Construtor: componente so pode ser construido
     * com uma entidade. */
    Componente(Entidade * entidade)
     : container(entidade) { };

    /* Metodo update deve ser reimplementado
    * por cada componente criado */
    virtual void update() = 0;
};
```

Listagem 2.5: Exemplo de uma classe *Componente* que pode ser usada para implementar uma arquitetura baseada em componentes.

```
class Pontos_de_Vida : public Componente {
  public:
    int health;
    virtual void update() {
      /* verifica health, verifica se o personagem morreu. */
    }
};

class Renderizador : public Componente {
  public:
    Imagem * imagem;
    virtual void update() {
      /* coloca a imagem na tela */
    }
};
```

Listagem 2.6: Exemplo de alguns componentes utilizando a classe *Componente* definida na Listagem 2.5 através de herança.

```
class Inimigo : public Entidade {
  public:
    Componente * health, * renderer, * behaviour;
    Inimigo() {
      health = new Pontos_de_Vida(this);
      renderer = new Renderizador(this);
          behaviour = new Comportamento_de_Inimigo(this);
    }
    virtual void update() {
      health->update();
      renderer->update();
      behaviour->update();
  }
};
```

Listagem 2.7: Entidade *Inimigo* utilizando componentes *Pontos de Vida* e *Renderizador* definidos anteriormente.

Entretanto, a simples composição possui alguns dos mesmos problemas de arquiteturas baseadas herança, principalmente o alto acoplamento. Por exemplo, a entidade representada pela classe *Inimigo*, na Listagem 2.6, precisa conhecer tipos e interfaces de seus componentes para que funcione corretamente [42].

Além disso, a interface *Entidade*, como apresentada na Figura 2.2, não só possibilita como incentiva a criação de hierarquias para representar entidades, pois possui métodos não implementados (por exemplo, *update()*). Por último, ainda corremos o risco de continuar com o *anti-pattern The Blob*.

Já que todos os componentes devem ser definidos herdando de uma classe pai, podemos criar uma classe *Entidade* capaz de conter um número ilimitado de componentes, de maneira a não fazer distinção entre eles, mas que ainda possam influenciar no comportamento das entidades a que pertencem [42], como na Listagem 2.8. Não mais herdamos da classe *Entidade* para definir entidades de um jogo, as definimos adicionando componentes que possuem funções específicas, contidas em uma única classe. Por exemplo, podemos criar a entidade *Inimigo* tal qual apresentado na Listagem 2.9.

```
class Entidade {
  list<Componente *> componentes;
  public:
    void update() { /* atualiza componentes */
      for (auto i = componentes.begin(); i != componentes.end(); i++)
        (*i)->update();
    }
    void adicionar(Componente * com) {
      componentes.push_back(com);
    }
};
```

Listagem 2.8: Atualização da classe *Entidade* que agora é capaz de ter vários componentes adicionados e tratá-los de maneira uniforme.

```
1 list<Entidade *> entidades; /* lista de entidades */
2 Entidade * inimigo = new Entidade(); /* novo inimigo */
3 inimigo->adicionar(new Pontos_de_Vida(inimigo));
4 inimigo->adicionar(new Renderizador(inimigo));
5 inimigo->adicionar(new Comportamento_de_Inimigo(inimigo));
6 inimigo->adicionar(new Busca_Trajetoria(inimigo));
7 entidades.push_back(inimigo);
8 run(entidades);
```

Listagem 2.9: Utilização da nova arquitetura de componentes para criar uma entidade inimigo.

Com essa nova arquitetura, reduzimos o nível de acoplamento, além de possibilitar que entidades sejam redefinidas de maneira clara e sem desperdício de código, até mesmo em tempo de execução. É possível também alterar a entidade que está sendo criada simplesmente trocando os componentes que possui.

Para resolver o problema do dragão que é tanto inimigo e veículo (exposto na Seção 2.3), adicionamos por exemplo uma instância do componente *DragonBehaviour* para definir o comportamento de dragão e esse se utiliza do componente *Pontos de Vida* para saber se foi derrotado pelo jogador. Neste o caso, os componentes *TerrestrianVehicle* e *FlyingVehicle* são adicionados, transformando a entidade em um veículo que pode ser montado.

Embora arquiteturas baseadas em componentes minimizem os problemas das hierarquias usadas para mapear entidades, outros problemas são trazidos à tona. Por exemplo, num contexto em que as entidades são definidas através de subclasses, já se conhece de antemão todos os atributos e métodos de uma entidade. Para se comunicar com ela é simplesmente uma questão de usar um método público exportado por sua classe. No caso de uma arquitetura baseada em componentes, como um componente sabe quais outros componentes estão adicionados à entidade? Como se comunicam entre si, já que não se conhecem? Por exemplo, na Listagem 2.7, como o componente *Comportamento de Inimigo* se comunica com o componente *Pontos de Vida* para descobrir se a entidade foi derrotada ou não?

Podemos assumir que alguns componentes sempre estarão presentes. Por exemplo, um componente do tipo *Transform*, que determina atributos de posição, tamanho e rotação de uma entidade certamente é candidato a estar presente. Esses componentes podem ganhar o seu espaço reservado na classe *Entidade*, usando o método padrão de composição, independente de terem sido explicitamente adicionados ou não. Dessa forma, um componente do tipo *Pontos de Vida* pode fazer parte da lista de variáveis padrão e o componente *Comportamento de Inimigo* pode fazer uma consulta à sua entidade *container*, seguir a referência até a instância do componente *Pontos de Vida* e acessar diretamente o valor dos pontos de vida.

Entretanto, esta abordagem é indesejável, por não lidar com outros tipos de componentes não previstos, além de reintroduzir a dependência e acoplamento valendo-se da referência direta ao componente *Pontos de Vida*, impossibilitando que esse seja removido ou trocado por outro.

É possível manter o código com dependências reduzidas através de técnicas de trocas de mensagem, em contraste com componentes padrão e referência direta. Através desta

abordagem, a entidade pode agir não só como um *container* de componentes mas também como um roteador de mensagens. Isto é, um componente que deseja saber quantos pontos de vida ainda restam, utiliza a entidade para fazer *broadcast* da mensagem `getPontos de Vida`. Se houver um componente *Pontos de Vida* adicionado, esse deve receber e tratar essa mensagem.

```
1  class Entidade {
2    public:
3      /* entidades e Metodos anteriormente adicionados */
4      template <class T>
5      void enviarMensagem(string mensagem, T & conteudo) {
6        list<Componente *>::iterator i;
7        for (i = componentes.begin(); i != componentes.end(); i++)
8          (*i)->novaMensagem(mensagem, conteudo);
9      }
10 }
```

Listagem 2.10: Exemplo de entidade capaz de realizar troca de mensagens entre os componentes adicionados.

Esta abordagem tem a vantagem de manter os componentes anônimos entre si e portanto mantém a redução no acoplamento. O código da Listagem 2.10 exemplifica um código para troca de mensagens. Nesse exemplo, um componente *Pontos de Vida* poderia receber a mensagem *"getPontos de Vida"*, calcular os pontos de vida restantes e talvez enviar uma nova mensagem *"healthValue"*, que seria percebida e processada pelo componente *DragonBehaviour* ou qualquer outro componente interessado.

Há, entretanto, um grande *overhead* na troca de mensagens: se existir apenas um componente interessado na mensagem *"getPontos de Vida"*, diversos componentes receberiam mensagens que não lhes interessa. É possível usar o padrão *Observer*, que reduziria esse ruído.

Outro problema é garantir a "autenticidade" dos emissores. Se o nome da mensagem for arbitrário, qualquer componente poderia enviar mensagens do tipo *"setPontos de Vida"*, causando confusão. Por último, a troca de mensagens torna a depuração uma tarefa muito mais complicada: se a entidade é usada para distribuir mensagens, a identidade do remetente se torna mais difícil de descobrir.

2.5 Arquiteturas orientadas a dados e *scripting*

Nos exemplos até agora apresentados, todo o jogo é definido em termos de código fonte que será compilado em um executável. Embora o uso da arquitetura de componentes reduza a quantidade de unidades a serem compiladas no caso de uma alteração nas entidades de um jogo, uma recompilação do todo ou de uma porção do código ainda é necessária sempre que se adiciona ou se remove um componente, ou ainda, sempre que se muda o valor inicial de uma variável. Isso pode parecer irrelevante no início do desenvolvimento de um jogo mas se torna um fator limitante ou, no mínimo, incômodo para quem está interessado em mudar alguns poucos valores. A solução ideal para problemas como esse é uma arquitetura orientada a dados.

Nessa arquitetura, entidades são definidas fora do código fonte. Isto é, desejamos definir entidades e configurar seus atributos iniciais de forma que isso não cause uma dispendiosa recompilação. A responsabilidade do código fonte do jogo e da *engine* incluirá carregar essas configurações previamente definidas e concretizá-las.

Os benefícios de uma arquitetura orientada a dados se tornam visíveis quando usado em conjunto com uma arquitetura flexível o suficiente para construir entidades em tempo de execução (ao contrário de arquiteturas baseadas em herança, que são imutáveis). Numa arquitetura baseada em componentes, arquivos de configuração são usados para definir composições e cada componente por sua vez é programado para funcionar da maneira reusável (com exceção dos componentes de comportamento específico). Um exemplo da definição da entidade *Dragon* no formato *XML* está na Listagem 2.11.

```
<entidade nome="Dragao">
  <componente tipo="Pontos de Vida" health="100" />
  <componente tipo="Comportamento de Dragao" />
  <componente tipo="Posicao" x=0 y=0 z=0 />
  <componente tipo="Render" image="dragon.png" />
</entidade>
```

Listagem 2.11: Exemplo da definição de uma entidade do tipo *Dragon* usando a linguagem de marcação *XML*.

Para acelerar ainda mais o processo de desenvolvimento e reduzir o número de recompilações é possível utilizar uma linguagem de programação interpretada para representar comportamentos, pois podem ser executadas num contexto de outra aplicação. Essa prática é conhecida como *scripting* e possibilita que entidades possam ser alteradas em tempo de execução, apenas recarregando o *script* que é responsável por seu comportamento. Após a carga da entidade definida num arquivo XML (por exemplo), o motor executa os *scripts* associados, que podem vir na forma de componentes, em seu *game loop*.

A habilidade de ser executado como parte de outra aplicação em tempo de execução permite que valores (como os pontos de vida de um personagem ou a velocidade de um carro) possam ser inspecionados e alterados enquanto se joga, permitindo uma grande flexibilidade não só no desenvolvimento e teste de novos cenários mas também no *debugging* desses jogos.

2.6 Discussão

Motores de jogos emergiram da necessidade de se criar arquiteturas de jogos reusáveis, reduzindo o custo de seu desenvolvimento. Uma coleção de diferentes técnicas, arquiteturas e ferramentas é empregada para se produzir um motor que possa ser usado em diferentes projetos e por equipes multidisciplinares. Além disso, o desenho final desse motor define como um jogo é criado ao redor dele. Por exemplo, a arquitetura orientada a dados permite que ferramentas como editores de mapa e entidades surjam para auxiliar o *level designer*[7] a testar diferentes combinações de entidades até chegar na configuração

[7] *Level designers* cuidam das fases do jogo, balanceando o número e a dificuldade dos inimigos, etc

ideal. Da mesma maneira, uma arquitetura baseada em componentes permite que componentes sejam produzidos por equipes distintas em momentos diferentes e ainda assim serem reutilizados em projetos futuros.

Cada arquitetura apresentada possui vantagens e desvantagens sobre as outras, mas no fim o que dita a arquitetura de um motor de jogos é o seu propósito. Motores de jogos que são desenhados para cobrir um nicho específico (por exemplo, motores de jogos de tiro) não precisam fornecer a flexibilidade de uma arquitetura baseada em componentes. Nesses casos, uma hierarquia concisa e clara é mais importante. Por outro lado, motores de jogos de propósito geral devem ser construídos em bases de arquiteturas maleáveis, permitindo que o desenvolvedor possa tirar proveito disso para moldar o seu jogo da forma como queira, criando novos componentes quando necessário.

Embora a arquitetura baseada em componentes seja mais flexível que uma hierárquica por exemplo, não vêm sem o seu próprio conjunto problemas. Particularmente, a comunicação entre componentes e entidades ou reintroduz o acoplamento que buscamos reduzir (fazendo referências diretas a entidades e seus componentes) ou impõe um grande *overhead* de um sistema de trocas de mensagens que não o próprio do paradigma orientado a objetos.

Para se tirar melhor proveito de uma arquitetura baseada em componentes, aumentando o seu grau de reusabilidade e reduzir sua complexidade, um sistema mais eficiente de comunicação entre componentes deve tomar lugar da referência direta e da passagem de mensagens. Aliando esse sistema a uma arquitetura orientada a dados e técnicas que promovem o reuso pode se criar um framework extensível, que permite o desenvolvimento de componentes independentes e desacoplados entre si, mas que são capazes de compartilhar dados relevantes em tempo de execução.

Tendo essa ideia como motivação, criou-se um motor de jogos de nome *Gear2D*. Seu modelo de componentes sustenta comunicação com baixo nível de acoplamento, permitindo que componentes sejam trocados até mesmo em tempo de execução. A discussão sobre o *design* da arquitetura e sua implementação serão discutidos a partir do próximo capítulo.

Capítulo 3

Gear2D

Embora a composição seja um mecanismo suportado no paradigma orientado a objetos, a atribuição de novas propriedades, de forma dinâmica e independente de uma hierarquia única, não é um recurso comumente explorado no design orientado a objetos; apesar de oferecer meios para adaptar as entidades dos jogos após uma definição inicial. Ou seja, embora seja possível inserir e remover componentes em tempo de execução como nas arquiteturas estudadas no Capítulo 2, a maioria das linguagens de programação que suportam o paradigma orientado a objeto não permitem, nativamente, a inserção e remoção de novos atributos em suas classes uma vez que o programa esteja em execução.

Para motores de jogos de propósito geral, poder inserir arbitrariamente atributos em uma entidade em tempo de execução é interessante pois, dada a flexibilidade da inserção e remoção de componentes durante o *gameplay*, não se pode prever de antemão todos os dados exportados e/ou requeridos pelos componentes adicionados a uma entidade após sua definição inicial.

A fim de contornar as limitações apresentadas, foi desenvolvida a *Gear2D*, um motor de jogos de propósito geral e baseado em componentes, com sua arquitetura voltada para suportar uma maior flexibilidade dos jogos eletrônicos e suas entidades.

Para garantir a recombinação de componentes na *Gear2D* foi necessária a construção de um *framework* base para prover a habilidade de adicionar e remover componentes em tempos distintos de execução. Baseando-se na abordagem *Adaptive Object Model* [45], a informação sobre a composição de cada entidade foi elevada ao nível do metadado através de arquivos de configuração[1]. Além disso, o suporte à adaptação em tempo de execução foi implementado através do padrão *Property* [45, 22].

Para evitar a necessidade da referência direta a atributos e métodos entre componentes, no intuito de promover reuso e evitar acoplamento, a metáfora do *Blackboard* [8] foi implementada: componentes operam sobre uma tabela de parâmetros e atributos (*Properties*) compartilhada entre eles para produzir um ou mais resultados desejados (Figura 3.1), sem a necessidade de fazer referência. Isso permite que componentes possam ser removidos em tempo de execução, sem que haja dependência de compilação entre eles.

Gear2D utiliza a tabela de parâmetros não apenas para armazenar valores relevantes, mas também para a comunicação entre os componentes através do mecanismo *signal and slots*, semelhante ao provido pela plataforma Qt [5]: cada parâmetro aceita *listeners*, que

[1] Isso também classifica *Gear2D* como um motor orientado a dados (*data-driven*) [4]

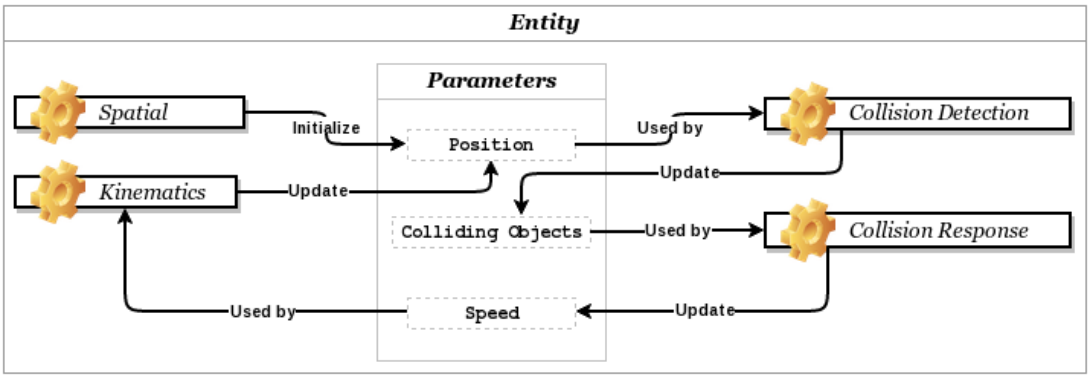


Figura 3.1: Fluxo de execução dos componentes posição, cinemática, dinâmica e colisão. Componente de cinemática usa parâmetros de aceleração e velocidade para alterar parâmetros de posição, que são utilizados pelo componente de colisão para detectar colisões expondo-as num parâmetro que é usado pelo componente de dinâmica para produzir alterações nos parâmetros de aceleração e velocidade.

são notificados para cada nova alteração realizada num parâmetro que lhes interessa, provendo meios para a realização de chamadas de métodos, completando o padrão *Observer*.

Em resumo, as responsabilidades do *framework* implementado na *Gear2D* são:

- Carregar arquivos de configuração para a disposição inicial dos objetos e componentes da aplicação.
- Permitir a inserção e remoção de componentes em tempo de execução.
- Garantir que cada componente obtenha uma fatia de tempo para seu próprio processamento.
- Permitir a comunicação entre os componentes, disponibilizando para isso uma tabela extensível de parâmetros e atributos.
- Carregar novas entidades em tempo de execução através de uma *API* disponibilizada.

Utilizando tais características do *framework*, é possível criar componentes que, juntos, completam um motor de jogos e possibilitam através de seu uso a construção de um jogo eletrônico.

3.1 Classificação de componentes

Entre os focos do *framework* apresentado está o reuso e a colaboração. Utilizando a vantagem do desacoplamento entre componentes é possível criar diferentes componentes parametrizados pelo mesmo conjunto de atributos mas que produzam comportamentos distintos ou que possuam implementações independentes.

Para tal, uma classificação de componentes se torna necessária, de maneira que estejam agrupados por similaridade, permitindo que um componente possa ser substituído

por outro quando necessário, característica importante em, por exemplo, sistemas multi-plataforma (comportamentos iguais porém com implementações diferentes).

Dessa maneira, componentes são classificados segundo uma família e um tipo, identificados de maneira única. Tipos são pertencentes a uma família e estas denotam uma funcionalidade que deve ser executada de forma compatível por todos os componentes desta família. Em outras palavras, uma família é usada para agrupar componentes por similaridade. Por exemplo, a família de renderização agrupa componentes que são destinados a produzir conteúdo visível.

3.2 Implementação do *framework*

Na *Gear2D*, os componentes são disponibilizados através de bibliotecas dinâmicas carregadas apenas em tempo de execução, sendo instanciados na medida em que são adicionados às entidades, permitindo que apenas o código necessário seja carregado e executado. Isso é possível através de *APIs* para carga de bibliotecas dinâmicas como a *libdl*[2], existente na maioria dos ambientes POSIX (*Portable Operating System Interface*).

O padrão de programação *Abstract Factory* [14] foi aplicado para a instanciação de entidades e seus componentes. Na fábrica de componentes, a família e tipo do componente são usados como parâmetros para a busca da biblioteca dinâmica, e esta exporta um método que é responsável por instanciar corretamente o componente, retornando para a *Gear2D* a respectiva referência (Figura 3.2).

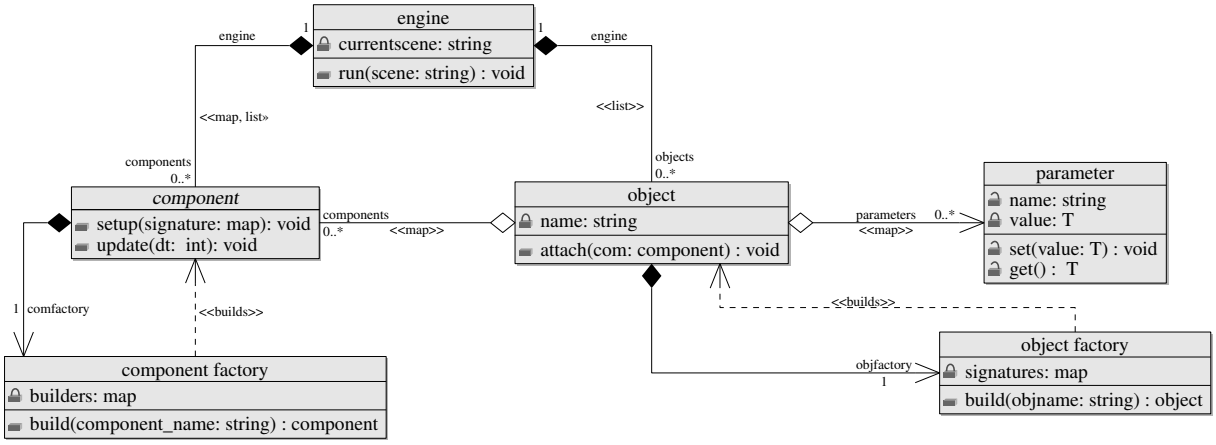


Figura 3.2: Classes do *framework*. A classe *engine* possui uma lista de todos os objetos (classe *object*) carregados, bem como todas as instâncias de componentes (classe *component*). Os objetos mantêm a lista de componentes adicionados a eles e a tabela de parâmetros (classe *parameter*). A classe *object_factory* é responsável por carregar as assinaturas das entidades, instanciando-as através de seu método *build()*, adicionando os componentes requeridos. Componentes são carregados através da classe *component_factory*.

Para saber a configuração inicial de cada entidade incluindo quais componentes deve instanciar, o *framework* carrega uma tabela de parâmetros dos arquivos de configuração (em formato *YAML*[3]). Esses arquivos são chamados de *assinatura* (um exemplo será

[2] http://linux.die.net/man/3/dlopen

[3] http://www.yaml.org, *YAML ain't markup language*

dado na Seção 3.4) e possuem os valores iniciais para parâmetros relevantes usados por cada componente à ser adicionado a entidade em questão. Uma vez em memória, essa assinatura é repassada ao componente recém construído para que possa analisar *(parse)* os parâmetros que lhe interessam e inicializá-los na tabela compartilhada de parâmetros, além de inicializar estruturas internas e/ou privadas do próprio componente necessárias para seu correto funcionamento.

Durante a execução do laço de eventos, os componentes são agrupados por família e executados em sequência. Isto é, todos os componentes da família de cinemática são executados, em seguida todos da família de detecção de colisão são executados, e assim por diante. Durante o tempo de execução de cada instância dos componentes, a tabela de parâmetros pode ser acessada (Figura 3.1) e seus valores lidos ou alterados conforme a necessidade, gerando o *gameplay* desejado.

Para o desenvolvimento do *framework* (e também dos outros componentes multimídia), foi utilizada a biblioteca *Simple DirectMedia Layer* (SDL[4]) para prover a *API* multiplataforma para a carga dinâmica de bibliotecas compartilhadas. SDL é uma biblioteca multimídia multiplataforma e de código aberto.

3.3 Modelo de Componentes e sua *API*

Gear2D é, em sua essência, uma camada de software que provê, entre outras características, um canal de comunicação entre componentes desacoplados, que são os blocos para a construção de games em motores de jogos baseados em componentes. Esta seção descreve o *design* do modelo de componentes da Gear2D e também como este pode ser utilizado.

3.3.1 Ciclo de vida dos componentes

A *API* exportada pelo motor de jogos Gear2D possui duas responsabilidades principais: (a) permitir comunicação entre componentes adicionados a uma dada entidade e, (b) prover funcionalidades úteis a criação do jogo como carga e descarga de entidades e cenas. Através dessa interface e do canal de comunicação, componentes podem prover diversas funcionalidades (por exemplo, detecção de colisão) que podem ser usadas por outros componentes.

Componentes para a Gear2D obedecem a interface abstrata *component*, que requer a implementação dos métodos *setup* e *update*, que correspondem aos dois estados mais importantes no ciclo de vida de um componente.

O estado de *setup* representa a fase de inicialização, executado uma vez a cada nova instância de um componente. A assinatura da entidade a qual o componente fará parte é passada para esse método, que deve interpretar seu conteúdo e inicializar a tabela de parâmetros. A partir desse ponto o componente passa a fazer parte da fila de execução de componentes.

A cada iteração do *game loop* os componentes têm sua chance de realizar simulações baseadas em tempo e outras ações requeridas para completar o *gameplay*, como renderi-

[4] http://www.libsdl.org

zação, simulações de corpo rígido, etc. Essa fase é a mais importante pois é a que de fato define o *gameplay*.

3.3.2 A *API* de Componentes

As funcionalidades do *framework* central da Gear2D estão disponíveis através da *API* de componentes: as fases de *setup* e *update* gerenciadas pelo motor são métodos abstratos implementados pelos componentes ao mesmo tempo que as funcionalidades disponibilizadas ao desenvolvedor estão disponíveis pela mesma classe.

A comunicação entre componentes é o núcleo da Gear2D e portanto sua utilização deve ser de fácil uso e entendimento. A leitura e escrita na tabela de parâmetros é feita através dos métodos *read*, para acessar e ler um valor na tabela de parâmetros e *write* para escrever um valor na mesma tabela. Como a linguagem C++ não disponibiliza nativamente reflexão de classes [18], ambos os métodos necessitam o nome e o tipo do parâmetro desejado.

Inicialmente, para ler o valor de um parâmetro x, o desenvolvedor deve usar uma sentença como `int x = read<int>("x")`. Para dar o valor 10 ao parâmetro x deve-se usar uma sentença como `write<int>("x", 10)`. Esses métodos foram abstraídos em uma classe *wrapper* parametrizável, *link*, que representa um elo transparente entre um parâmetro e um componente. Declara-se o elo x ao parâmetro *"x"* através da sentença `auto x = fetch<int>("x")`. Pode-se então usar a variável x de maneira transparente como um inteiro. Para escrever o valor *10* ao parâmetro *"x"* obtido anteriormente, usa-se uma sentença como `x = 10`.

Parâmetros suportam o padrão *Observer*, notificando componentes interessados sempre que um novo valor é atribuído. Um componente pode se tornar *listener* de um parâmetro usando a função *hook*, passando o nome do parâmetro a ser escutado. Quando há mudança de valor, o método *handle* (ou um outro *callback* de escolha do desenvolvedor) dos componentes interessados é chamado para que tenham a chance de lidar com a mudança. Esse mecanismo pode ser usado para simular chamadas de funções entre componentes.

Outros métodos providos pela *API* são:

- *destroy*: remove a entidade pai e os outros componentes adicionados a ela;
- *spawn*: carrega e inicializa uma nova entidade;
- *locate*: procura uma entidade por nome;
- *clone*: clona a entidade pai, incluindo seus parâmetros.

3.4 Criando o *Gameplay*

Diferentes motores de jogos e outras ferramentas de criação apresentam maneiras distintas de representar o *gameplay*. Em motores baseados em componentes, é natural que a dinâmica de jogo também seja representada utilizando componentes, se beneficiando dos outros componentes agregados numa mesma entidade. Essa abordagem também é utilizada na *Gear2D*. Nesta seção apresentamos alguns aspectos associados a implementação de dois jogos (*Pongroids* e *ReIgnice*) utilizando a *Gear2D*.

3.4.1 *Pongroids*

Pongroids (Figura 3.3) é um jogo para até dois jogadores inspirado no clássico Pong [44] com tema espacial (uma partida de tênis jogada no espaço sideral). As raquetes representam os jogadores e são controladas através do teclado; o objetivo é fazer a bola atravessar o campo adversário sem ser rebatida por asteroides ou pelo outro jogador. Quando ocorre uma colisão entre um asteroide e a bola, o asteroide se divide em outros de menor tamanho é a bola rebatida para o lado oposto.

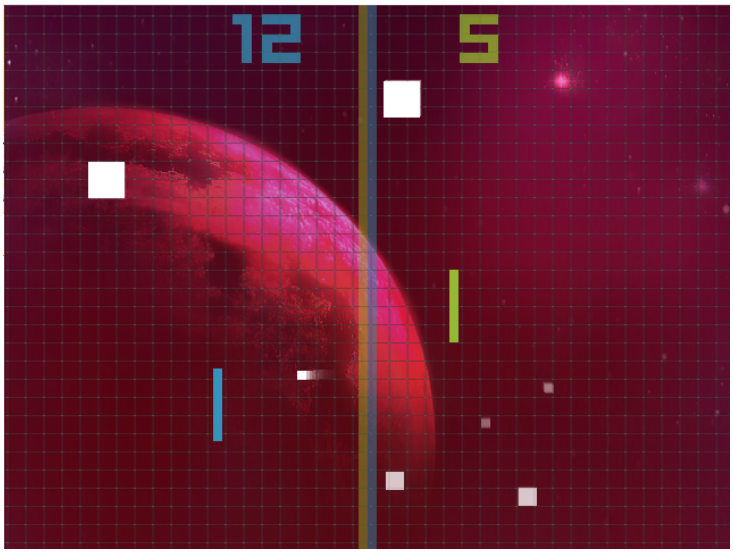


Figura 3.3: Captura de tela do jogo *Pongroids*

A construção do *Pongroids* além de servir como espaço de teste para componentes, foi essencial para identificar componentes desejáveis num motor de jogos e componentes específicos de *gameplay*. Conforme mencionado, as definições iniciais de cada entidade são criadas através de arquivos de configuração; cada arquivo corresponde à definição de uma entidade. Nessa definição está uma lista de componentes para serem adicionados e parâmetros iniciais a serem passados para esses componentes (como mostra a Listagem 3.1).

```
# lista de componentes a adicionar
attach: collider dynamics renderer

# detectar colisao
collider.tag: bar
collider.ignore: asteroid leftfield

# parametros referentes a renderizacao
renderer.surfaces: bar=barleft.bmp
bar.alpha: 1.0
```

Listagem 3.1: barleft.yaml: Arquivo representando o jogador no campo esquerdo. O parâmetro *attach:* especifica a lista de componentes a adicionar. Os outros parâmetros indicam valores iniciais a serem usados pelos componentes no momento de sua adição. Por exemplo o componente *renderer* utiliza *renderer.surfaces* para a listagem de imagens a carregar e renderizar.

Com o reúso do motor *Gear2D*, a construção do *Pongroids* exigiu um total de 502 linhas de código distribuídas em cinco componentes:

- *lobby:* Componente que implementa o menu da tela inicial do jogo. Utiliza o componente de teclado para controlar propriedades de renderização de texto e para carregar o tipo de partida escolhida (um ou dois jogadores).
- *partida:* Componente que implementa as regras da partida, isso é, a contagem de pontos e a atualização das imagens de fundo baseado no tempo do jogo.
- *asteroide:* Componente para regular quantos e quais asteroides (pequenos, médios ou grandes) estão em jogo e quantos novos asteroides seriam instanciados quando ocorre uma colisão.
- *raquete:* Componente que toca um som quando ocorre a colisão entre a raquete e a bola. Note que o controle da raquete é feito através do componente de controle de personagens, que alteram parâmetros de aceleração baseado em teclas acionadas pelo teclado, configuráveis também através de parâmetros. Tal componente não é específico do jogo *Pongroids*, mas sim disponibilizado juntamente com o motor.
- *raquete-ia:* Componente que simula a inteligência artificial utilizada em uma das raquetes quando a partida é para apenas um usuário.

Todos os componentes precisam ser atribuídos a entidades para que possam funcionar e a comunicação entre duas entidades ocorre de forma flexível, através de parâmetros. Em outras palavras, um componente de uma entidade pode ler ou escrever parâmetros em outras entidades. Para tal, a *API* da *Gear2D* oferece mecanismos para recuperar instâncias de entidades bem como acessar parâmetros de entidade.

Por exemplo, o componente *lobby* cria instâncias dos menus e submenus dependendo do *input* do usuário, controlando suas propriedades diretamente para refletir as escolhas do jogador. Já o componente *match* controla a pontuação, registrando-se como *listener* nos parâmetros de posicionamento da bola do jogo. Sempre que avançar uma das linhas de fundo ele ajusta a pontuação de acordo.

3.4.2 *ReIgnice*

Ignice é um jogo do gênero *side-scroller*; o personagem é representado por uma ave que possui dois estados: fogo e gelo. O desafio do jogo consiste na habilidade do jogador em evitar meteoros que são do elemento contrário. Is to é, enquanto a ave é de fogo, deve evitar meteoros de gelo e acertar meteoros de fogo. É possível inverter o estado da ave no decorrer da fase.

ReIgnice é um clone do jogo *Ignice*, com o intuito de fazer uma comparação entre duas abordagens distintas. *Ignice* foi originalmente construído apenas com a biblioteca SDL e uma arquitetura moldada para a sua construção. *ReIgnice* foi construído utilizando a *Gear2D*, reusando a maioria dos componentes já criados mas em uma outra configuração.

Na implementação do *ReIgnice*, três novos componentes foram criados:

- *fenix:* Componente adicionado à entidade da fênix, o personagem principal do jogo. Suas tarefas principais compreendem alterar a posição do personagem baseando-se

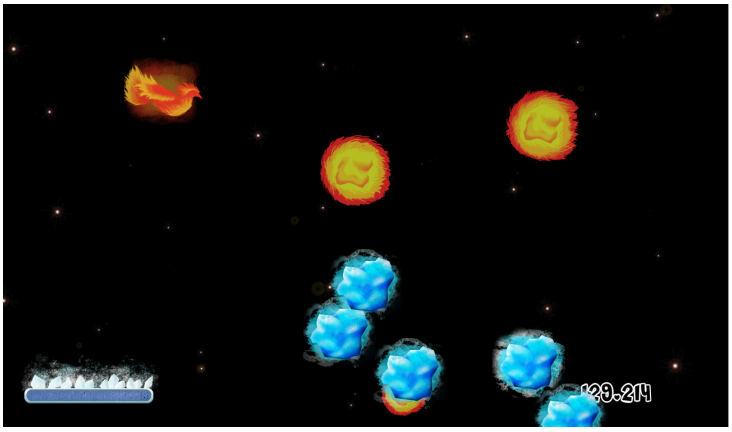


Figura 3.4: Captura de tela do jogo *ReIgnice*

em parâmetros de posição do mouse (que são reportados através do componente mouse), trocar a imagem atual do personagem (fogo, gelo e transições entre os elementos), gerenciar pontos de vida, entre outros.

- *meteoro:* Componente utilizado nos meteoros de fogo e gelo. Note que o mesmo componente é utilizado para todos os tipos e tamanhos de meteoros.

- *splash:* Componente utilizado na tela. Carrega o sai do jogo de acordo com a interação com o mouse.

Comparativamente com *Pongroids* (Subseção 3.4.1), foram observados muitos componentes em comum utilizados: componentes de renderização, cinética e colisão foram extensivamente usados nas duas implementações, sem alterações. O código original de *Ignice*, que foi criado usando apenas a biblioteca SDL, é constituído de 2404 linhas de código C++, enquanto *ReIgnice*, em seus três componentes, totaliza 383 linhas de programação e 286 em arquivos de configuração (semelhantes aos descritos na Subseção 3.4.1), mantendo os aspectos originais de *gameplay*.

Capítulo 4

Validação de *design*

Embora a Gear2D tenha sido usada com sucesso para criar e portar jogos menores para fins de teste de suas capacidades (Capítulo 3) e para desenvolver componentes de sua biblioteca padrão de componentes, ainda era necessário validar seu design com demandas maiores e mais complexos.

A Gear2D foi avaliada em dois aspectos: desenvolvimento de componentes reusáveis não-triviais para estender as funcionalidades já apresentadas e o desenvolvimento de jogos de requisitos e gameplay mais elaborados dos que os apresentados no Capítulo 3.

Esse capítulo discute o *framework* do ponto de vista do desenvolvedor de componentes através do processo de desenvolvimento dos componentes não-triviais *Pathfinder2D* (Seção 4.1) e *lua-proxy* (Seção 4.2), em conjunto com os desenvolvedores Igor Rafael de Souza e Luiggi Monteiro Reffatti, respectivamente [11]. É discutido também o processo de desenvolvimento do jogo *Naval Warfare*, submetido para o Festival de Jogos do SBGames IX (2012), em um time de cinco desenvolvedores (Matheus Pimenta, Antônio Martino, Guilherme Costa, Davi Diniz, Tiago Galvão).

4.1 Componente de *Pathfinding*

A relevância (e complexidade) da Inteligência Artificial (IA) nos jogos digitais é cada vez maior, ao ponto que quase todos os jogos produzidos hoje apresentam algum elemento de IA [27]. Isso pode ser notado tanto em jogos simples, onde a IA deve cuidar de tarefas triviais quanto em jogos mais complexos, onde a qualidade do *gameplay* é diretamente associada com o desafio apresentado. Uma IA adequada é crucial para o sucesso de vários gêneros, como RPGs *(Role Playing Games)* e jogos de estratégia.

Quase todos os jogos incluem algum tipo de personagem não jogável (ou NPCs, do inglês *Non Playable Character*) que pode se mover pelo ambiente. Considerando que há sentido nessa ação, é esperado que essa seja executada apropriadamente, usando a rota de menor custo e evitando ficar preso em obstáculos. A técnica que possibilita isso é chamada *pathfinding* e é uma tarefa inerente a vários jogos tais como *StarCraft 2 (Blizzard)* e *Battlefield 3 (Electronic Arts)*. *Pathfinding* é uma das aplicações mais comuns de IA e é portanto um bom candidato para se tornar componente reusável.

4.1.1 Implementação

Pathfinder2D é um componente não trivial que objetiva descobrir uma trajetória apropriada entre a posição corrente de uma unidade qualquer até o seu destino. Essa tarefa aparentemente simples pode ter muitas implementações diferentes, cada qual com seu próprio conjunto de vantages em desvantagens em termos de aplicabilidade e recursos necessários. *Pathfinder2D* aborda a tarefa do *pathfinding* de duas maneiras distintas: a representação do espaço e a busca de uma trajetória nessa representação.

A abordagem usual para representar o espaço em jogos digitais é através de *grids* e malhas de navegação *(navigation meshes)* [41], que devem ser escolhidas de acordo com a tarefa a ser executada. Uma *grid* muito grande pode representar o espaço de maneira imprecisa, enquanto uma muito pequena pode resultar em um espaço de busca muito grande. Mesmo que seja definida de maneira razoável, a geometria dos objetos no espaço podem influenciar o resultado.

Uma implementação mais abrangente usa malhas de navegação, um grafo que representa o espaço em divisões de polígonos convexos que são usados como nós. Dentro de cada polígono, um agente deve ser capaz de se mover de qualquer ponto a outro em linha reta. Como cada triângulo é um polígono convexo, tal malha pode ser obtida triangulando o espaço do ambiente [28]. *Pathfinder2D* implementa o método de triangulação chamada *DCDT*: *Dynamic Constrained Delaunay Triangulation* [21][1], que maximiza o angulo mínimo de todos os ângulos de triângulos (criando uma malha mais balanceada), resultando em melhor performance em ambientes dinâmicos.

Com uma representação apropriada do espaço, a busca por uma trajetória que leva o agente para o seu objetivo é feita aplicando o algoritmo A* [19], que não só encontra a solução ótima, mas, quando usado com uma função de heurística razoável, o faz com o menor esforço necessário. Aplicado a uma malha triangulada, a versão do algoritmo (chamado de *Triangulation A** [12]) usa uma função adaptada de custo para computar a trajetória a fim de garantir que a função é consistente.

4.1.2 Integração com a Gear2D

Na Gear2D o ambiente é representado através de um espaço bidimensional contínuo, então o componente deve encontrar a trajetória partindo da posição atual da entidade (x_1, y_1) até o seu destino (x_2, y_2). Tais parâmetros são facilmente obtidos através da tabela de parâmetros, fornecidos por um componente espacial *space2d*. A trajetória resultante é representada em uma sequência de pontos (x, y) que a entidade pode usar para se locomover. Essa informação é disponibilizada para outros componentes na mesma tabela de parâmetros.

A fim de criar um pequeno jogo para demonstração e ainda assim manter o componente *Pathfinder2D* responsável por apenas encontrar e retornar um caminho, dois outros componentes foram criados: *Commander* e *Follower2D*.

O componente *Commander* age como uma ponte entre o usuário (usando o componente *mouse*) e as entidades do jogo, permitindo que o jogador dite as ações da unidade. Esse componente lida apenas com o comando *"move to* (x, y)*"*, indicando o destino final da

[1] Em tradução livre: Triangulação Restrita e Dinâmica de Delaunay

unidade. Esse comando é um dos mais usados em jogos do gênero estratégia [32], onde o usuário pode controlar múltiplas unidades e ordenar diferentes comandos a elas.

Uma vez que a trajetória é definida, o componente *Follower2D* especifica como a unidade vai se mover entre os pontos. Esse componente é responsável por alcançar a coordenada final, que inclui evitar colisões com objetos dinâmicos que podem estar no caminho. Idealmente, a rota tomada deve ser uma linha reta entre cada par de pontos, mas a trajetória pode se tornar mais complexa para evitar objetos que se movem dentro do mesmo triângulo, usando por exemplo técnicas de *steering behavior* [33]. Na Figura 4.1 temos uma captura de tela do jogo *Katch-Up* criado para demonstrar os componentes.

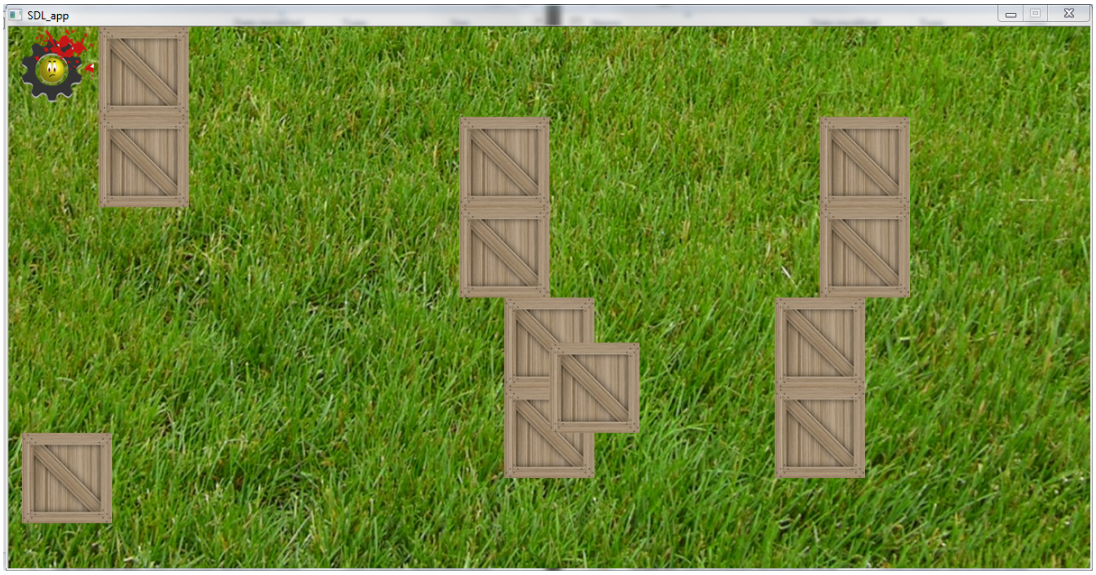


Figura 4.1: Katch-Up: Jogo feito para demonstrar a capacidade do componente *Pathfinder2D*.

Os componentes da Gear2D se comunicam livremente uns com os outros através da tabela de parâmetros, enquanto a implementação do padrão *Observer* via métodos *hook* e *handle* permite que eles interajam. Isso, junto com a relação direta entre o espaço contínuo do componente *space2d* e a implementação do *Pathfinder2D*, permitiu que esses três componentes comunicassem entre si e com o motor. Além disso, graças ao desacoplamento garantido pela Gear2D, é permitido que os componentes *Follower2D* e *Commander* funcionem facilmente com outras implementações de *pathfinding*, bem como permite que o *Pathfinder2D* funcione com outros componentes de interface e movimentação.

Ao todo, o código que envolve os 3 componentes mencionados contém aproximadamente 2400 linhas de código fonte, ignorando comentários e espaços em branco. Dessas, apenas 300 representam a implementação realizada para interfacear com a Gear2D. Comparativamente com o motor (excluindo-se os componentes padrão, que são externos), o código dos componentes para implementação de *Pathfinding* são maiores em quase mil linhas.

4.2 Scripting em Lua

O design da Gear2D [10] permite que componentes sejam desenvolvidos separadamente do motor, exportados em uma biblioteca dinâmica que é carregada em *runtime*. Essa característica facilita alguns aspectos do desenvolvimento e possivelmente reduz o tempo de compilação, mas a biblioteca dinâmica deve ser recompilada a cada alteração feita ao código do componente. Enquanto isso é tolerável, esse processo começa a se tornar improdutivo para tarefas que requerem mudanças frequentes ao código (por exemplo, ajustando parâmetros de *gameplay*).

A solução natural para esse problema é permitir o desenvolvimento de componentes usando uma *scripting language* [43], interpretada em tempo de execução ou compilada com técnicas do tipo *JIT (Just In Time)* [3]. Scripts permitem interfaces em alto nível, uma característica particularmente útil para programadores ainda inexperientes ou para que *game designers* possam experimentar diretamente com o *gameplay* sem a necessidade de trabalhar com ferramentas de desenvolvedor.

A maneira menos intrusiva de integrar *scripting* na Gear2D, sem que seja necessário alterar o código fonte do motor, é através de um componente capaz de carregar outros componentes escritos em outras linguagens, agindo como um *proxy* entre componente e motor. A linguagem de programação Lua é descrita por seus autores como sendo "uma linguagem de *scripting* embutida poderosa, rápida e leve", sendo considerada a linguagem de *scripting* mais usada na indústria de jogos [43]. Lua é usada extensivamente em jogos como *World of Warcraft (Blizzard)*[13] e motores de jogos como *CryEngine2 (Crytek)*[9] e *LÖVE*[23].

4.2.1 Implementação

Lua usa cadeias de caracteres *(strings)* como identificadores nativamente, facilitando integração com a tabela de parâmetros da Gear2D já que ambos tem métodos de acessos a valores bastante similar.

Como a máquina virtual que interpreta a linguagem Lua consome pouca memória, cada entidade pode ter o seu próprio ambiente de interpretação, evitando que o código do componente tenha que gerenciá-los por conta própria, além de garantir a independência de cada instância do componente. Isso permite que a interface com o motor seja transparente e de fácil compreensão. Quando uma nova instância do componente *lua-proxy* é carregada, as funções que definem a interface entre o motor e o componente em lua são adicionadas ao ambiente, preparando a *API* em Lua para o desenvolvedor.

Durante a fase de *setup* (Seção 3.3) do componente *lua-proxy*, os componentes em Lua destinados a serem adicionados àquela entidade e suas dependências são carregadas no ambiente de interpretação. Cada um desses componentes é representado por uma tabela em Lua e têm suas meta-tabelas *(meta-tables)* e funções internas alteradas para se adequar a interface com a Gear2D.

4.2.2 Integração com a Gear2D

Assim como todo componente da Gear2D, o *lua-proxy* deve acessar a tabela de parâmetros da entidade através dos métodos *read* e *write*, que requerem a declaração explícita

do tipo do parâmetro (como descrito na Seção 3.3). A fim de prover uma interface simples ao programador Lua, as funções de acesso devem ser inseridas na *meta-table* [20] de cada componente de modo que leituras e escritas são lidadas transparentemente [2]. Entretanto, é importante manter essa transparência apenas para parâmetros que devem ser compartilhados via a tabela de parâmetros da entidade, para evitar populá-la com variáveis locais do componente em Lua.

A solução para essas preocupações vem através da declaração explícita dos nomes e tipos das variáveis que devem ser sempre escritas e lidas da tabela de parâmetros. A necessidade de explicitar os tipos dessas variáveis é um problema considerando a flexibilidade da linguagem Lua, mas é essencial para manter a compatibilidade com componentes criados em C++, permitindo o uso de tipos arbitrários como *structs* ou *classes*. A função usada para dizer ao componente *proxy* quais parâmetros devem ser exportados recebe dois parâmetros: o nome do parâmetro e seu tipo.

4.3 Naval Warfare

Naval Warfare é um jogo de estratégia para dois jogadores baseado em turnos [30], em torno do tema "batalha naval". O objetivo de cada jogador é destruir o porto inimigo. A versão implementada apresenta 2 (duas) facções: Vikings e Piratas. Cada facção possui 3 (três) tipos de embarcações a serem colocados em batalha: Corvetas, que são embarcações leves e rápidas, com pouco poder de fogo e resistência, Fragatas, que representam embarcações com médio poder de ataque, resistência e velocidade e Cruzadores, embarcações de grande porte e com alta resistência mas que não se movem com rapidez.

O primeiro jogador representará os Vikings e o segundo os Piratas. Começa-se com um porto cada e certa quantia em moedas de ouro, usadas para custear novas embarcações para a batalha (o custo aumenta conforme o tamanho da embarcação). As jogadas acontecem em rodadas, constituídas de dois passos: uma fase de planejamento, com um turno para cada jogador planejar suas ações ou comandos e a fase de simulação quando cada embarcação executa a ação comandada. Na fase de planejamento, cada jogador, em seu turno, compra quantos barcos achar necessário considerando a quantidade de ouro disponível. Além disso, é possível comandar cada embarcação a se mover para um destino no mapa, considerando-se seu alcance máximo por turno e o alcance de seus canhões, bem como o alcance dos canhões das embarcações inimigas. Durante a fase de simulação, barcos inimigos serão bombardeados caso estejam em distância menor ou igual ao alcance de seus canhões, mesmo em movimento. Barcos destruídos revelam um baú de tesouros que podem ser coletados, gerando mais moedas de ouro.

As fases e os turnos são arranjados da seguinte forma. No início é sorteado um jogador para começar *("jogador 1")*, que compra barcos e planeja a movimentação de suas unidades. Assim que finalizado, o outro jogador *("jogador 2")* tem sua vez para realizar as mesmas ações, levando em consideração a jogada do jogador anterior. Ao terminar o planejamento começa a fase de simulação: embarcações se movem até o destino alvejando unidades inimigas no processo, incluindo os portos. Ao fim dessa fase, o jogo retoma a fase de planejamento e o último jogador a comandar sua frota é o que começará nessa

[2] Por exemplo, em Lua $x = 10$ resulta na chamada correspondente *write<int>("x", 10)*

(a) Início do jogo

(b) Jogo em andamento

(c) Fim de jogo

Figura 4.2: *Naval Warfare*: (a): Jogo em fase de planejamento, turno do primeiro jogador. (b) Alcance de movimento (círculo branco) e de ataque (círculo vermelho) da fragata. (c) Finalização do jogo com um dos jogadores vitoriosos.

nova rodada. Esse ciclo continua até que um dos portos seja destruído, dando vitória ao jogador que o destruiu.

4.3.1 Design e pré-desenvolvimento

O desenvolvimento do jogo *Naval Warfare* teve como objetivo determinar o nível de praticidade do modelo de componentes da Gear2D na implementação de jogos eletrônicos em equipes de pequeno a médio porte, levantar pontos a serem melhorados em sua curva de aprendizagem e identificar problemas no motor e solucioná-los.

Montou-se uma equipe de 5 (cinco) desenvolvedores, com pouca ou nenhuma experiência com o desenvolvimento de jogos. Para alguns deles era a primeira vez que tinham contato com motores de jogos, especialmente numa arquitetura baseada em componentes.

Antes de dar início ao desenvolvimento, foi levantada uma discussão sobre diferenças entre arquiteturas baseadas em heranças e componentes para definir comportamento de

entidades e como a última permite um maior grau de flexibilidade. Em seguida houve uma revisão dos *Design Patterns* potencialmente úteis para o desenvolvimento, como *Composite*, *Observer* e *Property*. Por último, o modelo de componentes e a *API* da *Gear2D* foi introduzido aos desenvolvedores.

O próximo passo então foi elencar ideias de qual jogo a equipe gostaria de fazer, considerando a tecnologia disponível, a curva de aprendizagem, *deadlines* e principalmente se seria um jogo que eles gostariam de jogar. Desse processo obteve-se o tema (batalha naval de Piratas vs Vikings) e a mecânica (estratégia em turnos). Nas reuniões seguintes escreveu-se um breve documento de *game design* agregando e formalizando novas ideias. Uma vez completo, esse documento serviu como guia para o processo de desenvolvimento [36].

Em linhas gerais, pode-se sumarizar o documento de *design* nos seguintes pontos:

- Jogo para dois jogadores.
- Estratégia baseada em turnos.
- Três unidades: Corvetas, Fragatas e Cruzeiros, com alcances de movimento e tiro diferentes.
- Moedas de ouro são coletadas dos destroços dos barcos.
- Tela de entrada, tela de menu, *gameplay* e *game over*.
- Turnos alternados por rodadas.
- O jogador perde quando seu porto é destruído.
- Estilo visual semelhante aos mapas cartográficos da era das grandes descobertas (séculos XIV-XVI).

O processo de desenvolvimento foi baseado no *framework Scrum* de gerenciamento de processos ágeis [37]: a equipe trabalhava em *sprints* semanais com objetivos bem definidos, usando um *Kanban-board* [1] customizado para *Scrum*, com reuniões todas quintas e sextas-feiras. Nas quintas a equipe se reunia para compartilhar problemas que não puderam resolver ou que tornou impossível atingir suas metas semanais, com discussões diversas sobre o jogo e troca de experiências entre os desenvolvedores. Nas sextas-feiras discutia-se os objetivos para a próxima semana.

O desenvolvimento foi divido em características do jogo e suas unidades, o que ditou a divisão da equipe: cada desenvolvedor ficaria responsável por uma *feature* completa. Decidiu-se que havia a necessidade de ao menos um componente para cada entidade e inicialmente se pensou nos seguintes:

- O componente *barco* cuidaria do comportamento de uma embarcação, desde sua movimentação e ataque, passando por comandos de movimentação dados, até a decisão de quando a embarcação tinha sido destruída. Os barcos que tivessem terminado de se mover durante a fase de simulação reportariam a entidade que tivesse o componente *porto* que estavam prontos.

- *porto* é o componente que cuida do porto, isto é, quantos pontos de vida têm, se há ouro suficiente para construir o barco pedido e cuida também de efetivamente carregar e criar os barcos pedidos pelo usuário e manter meta-informações sobre eles (quantos estavam em jogo, se foram destruídos, etc.). Esse componente mantinha uma lista para todos os barcos do jogador para que pudesse dizer, na fase de simulação, se todos os barcos já tinham atingido o seu destino.

- O componente *match* cuidaria da meta-informação do jogo, ou seja, de quem era o turno, em qual rodada estavam, quantas se passaram, quantos pontos cada jogador tinha e se a condição de vitória tinha sido atingida. Esse componente guardava referências aos barcos, para consultar informações em sua tabela de parâmetros.

Entretanto, durante o desenvolvimento, surgiram diversos outros componentes adicionais reusáveis para suportar as características necessárias para o funcionamento do jogo. Entre eles, os mais notáveis são:

- *mainmenu* é o componente que cuida do menu principal da tela de entrada, escutando as teclas de seleção (setas) e iniciando o jogo no modo correto.

- *fade* faz animações no estilo *fade-in* e *fade-out* nas imagens ajustando seu valor de transparência. Usado principalmente nas animações do menu inicial.

- *painel* é o componente que cuida do menu lateral de cada jogador para a seleção de qual unidade deve ser criada, conversando com a entidade que tinha o componente porto para que fossem criadas novas unidades.

4.3.2 Ambiente de desenvolvimento

Ambos, o motor de jogos e seus componentes, dependem do sistema de *build* chamado CMake, pois, por ser multiplataforma e capaz de gerar arquivos de *build* para várias combinações de plataformas diferentes *(Windows/MinGW, Windows/VisualStudio, Linux/GCC, etc)*, é um bom software pra ajudar a amplificar o publico da Gear2D, mantendo-a o mais independente de plataforma possível.

Na tentativa de manter também os novos componentes desenvolvidos para o jogo multiplataforma decidiu-se manter o sistema de *build* compatível. Entretanto, era um sistema até então desconhecido pelos desenvolvedores, e foi o primeiro obstáculo a ser enfrentado. Escrever regras de *build* para o *CMake* mostrou-se difícil no começo e manteve-se uma dificuldade constante durante todo o processo pois a linguagem usada para tal era desconhecida para a maioria dos desenvolvedores.

Junto a barreira do sistema de *build*, configurar o ambiente apropriadamente para desenvolver e testar componentes foi outro obstáculo para alguns desenvolvedores dado que o código fonte do motor não fazia parte diretamente dos componentes mas sim era representado como uma biblioteca dinâmica que precisava estar presente em tempo de ligação e execução. A configuração desse ambiente foi feita de forma manual (ajustando os caminhos de ligação nas regras de *build*) pois não havia uma arquitetura padrão de diretórios/arquivos e, dada a natureza multiplataforma do motor, não havia um lugar padrão para residir as bibliotecas da *Gear2D*.

Ficou claro que a dificuldade em configurar um ambiente de desenvolvimento necessário para trabalhar com o motor representava uma barreira significativa para desenvolvedores iniciantes na *Gear2D* pois a disposição de suas bibliotecas não é usual. Maneiras de superar esses problemas estão em desenvolvimento e serão discutidas no Capítulo 5 como trabalhos futuros.

4.3.3 Entidades

Entidades são os *containers* dos componentes que os dão comportamento, intervindo como o canal de comunicação entre eles, mantendo a tabela de parâmetros compartilhados. Entretanto, a interface de programador da *Gear2D* faz a interação entre componentes de maneira transparente, tornando desconhecido o espaço em que reside a tabela de parâmetros. A única forma direta de interação com uma entidade é a definição de sua assinatura, estabelecido pelo desenvolvedor, contendo o conjunto de valores iniciais daquela entidade.

Para o *Naval Warfare*, as seguintes entidades representam o núcleo do jogo:

- As entidades *barco-pequeno*, *barco-médio* e *barco-grande* representam os três tipos de barco jogáveis e seriam *containers* do componente *barco*. Entre outros parâmetros, são especificados o raio de alcance de movimento e de ataque do barco, seus pontos de vida, sua velocidade de movimentação e o seu valor em moedas de ouro. Atributos auxiliares relacionados a interface de usuário também foram definidos aqui, como por exemplo a posição relativa da barra que indica os pontos de vida restantes do barco.

- As entidades *porto-p1* e *porto-p2* são, respectivamente, representações do porto para o jogador 1 e 2. São *containers* do componente *porto* e definem a posição do porto, seus pontos de vida, o raio definindo a distância máxima em que novos barcos aparecerão, onde é exibido a quantidade de moedas de ouro do jogador, etc.

- A entidade *partida* é *container* do componente homônimo e contém informações sobre posicionamento e imagens para a maior parte da interface de usuário como as mensagens indicando o começo e fim dos turnos e rodadas além de carregar as entidades dos portos.

Além destas foram criadas várias entidades auxiliares, como por exemplo a entidade *painel*, *container* do componente de mesmo nome, para representar o menu para a construção de novos barcos, apresentado na lateral de cada porto, além de entidades que tomariam conta das telas de início, pausa e fim de jogo e seus componentes correspondentes.

4.3.4 Componente: *barco*

A equipe começou pelo componente *barco*, que representa a implementação do comportamento de cada barco. Essa decisão foi tomada pois as embarcações são a parte mais importante de toda a mecânica do jogo e tornaria concreta a percepção que os desenvolvedores tinham sobre o *gameplay*, permitindo ajustes de balanceamento até a versão definitiva.

Em linhas gerais, as responsabilidades desse componente se resumiram em:

(1) Receber a assinatura das entidades de cada barco, inicializando a tabela de parâmetros de acordo.

(2) Receber do componente de detecção de colisão a notificação de unidades próximas para que possa atacar e de colisões com barcos inimigos.

(3) Informar unidades em seu raio de ataque que está sendo infligido dano a elas e tratar essas notificações.

(4) Receber comandos de novos destinos a serem alcançados no mapa, na fase de simulação.

(5) Manter informações a respeito de seu estado na fase de simulação, como por exemplo se chegou ao seu destino.

(6) Manter a interface de usuário exibindo representações para o raio de ataque e o raio de movimentação.

Em (1), além de inicializar a tabela de parâmetros com a assinatura definida para a entidade, foi necessário realizar alguns ajustes finais. Como o componente *barco* é responsável por todos os tipos de embarcações, o tipo do barco e sua facção (Viking ou Pirata) influenciou na imagem a ser carregada. Além disso foi preciso ajustar valores para a detecção de colisão de acordo com o raio de ataque. Como a entidade *container* desse componente era carregada apenas por demanda do usuário (quando havia uma nova embarcação a ser colocada em batalha), algumas das inicializações ainda são feitas tardiamente (por exemplo, a informação de qual aquele navio pertencia), pelos componentes *partida* e *porto*, associados às entidades de mesmo nome.

Percebeu-se que a biblioteca de componentes disponíveis para os desenvolvedores seria insuficiente para tratar alguns dos casos, especialmente o caso (2). Era planejado usar o componente *collider/collider2d*, que é capaz de detectar intersecções entre retângulos *(axis-aligned bounding box)*, para tratar o caso em que um barco estaria no raio de ataque. Isso seria implementado inicializando o retângulo de colisão da entidade com valores virtuais equivalentes ao tamanho do raio de ataque, entretanto, o componente até então era incapaz de usar múltiplos retângulos de colisão por entidade, impedindo que fosse usado para detectar também colisões reais entre barcos.

A solução desenvolvida para contornar essa limitação envolvia receber a notificação de que um barco havia colidido com o retângulo representando o alcance de ataque e então recalcular a colisão no próprio componente *barco* para verificar se eles estavam colidindo diretamente. Ficou evidente que era necessário um componente de detecção de colisões mais robusto. Uma solução que está em desenvolvimento será discutida no Capítulo 5 como trabalho futuro.

Para que os barcos inflijam e recebam dano [Caso (3)], usa-se o esquema de colisão já mencionado, aliado ao padrão *Observer* implementado através dos métodos *hook()* e *handle()*. Cada instância do componente *barco* escuta o parâmetro *"collideswith"*, que é preenchido pelo componente de detecção de colisão. O valor desse parâmetro indica o objeto que está colidindo, que, nesse caso, representa um outro barco no raio de ataque. Com essa informação, o componente barco é capaz de escrever na tabela de parâmetros do outro barco, avisando que está o atacando.

A funcionalidade (4) foi implementada através do uso do componentes *mouse* e *mouseover*. O primeiro apresenta a posição do mouse através dos parâmetros *"x"* e *"y"*, além dos botões pressionados, enquanto o último informa quando o mouse está sobre a região da entidade em questão. Quando um clique na entidade era detectado, exibia-se o raio de movimentação e de ataque [Figura 4.2(c)]. Nesse estado, o cursor mudava para indicar que o jogador estava selecionando o destino daquela embarcação.

A responsabilidade (5) era crucial para o desenrolar do *gameplay* pois resultava no parâmetro *"done"*, observado pelo componente *porto*, fazendo parte na decisão de que a fase de simulação tinha chegado ao seu fim para que se pudesse avançar para a próxima fase.

Implementar (6) foi relativamente trivial, bastando apenas manipular as imagens já carregadas através do componente *renderer/renderer2d*.

4.3.5 Componente: *porto*

O trabalho do componente porto foi criar novas unidades de acordo com a escolha do jogador, mantendo informações sobre a quantidade de moedas de ouro restantes. Além disso, a entidade porto possuía pontos de vida e o jogo era ganho quando o porto inimigo estava destruído.

Resumidamente, um porto deve:

(1) Inicializar-se de acordo com a assinatura da entidade.

(2) Manter comunicação com a entidade de menu de escolha de barco a ser criado.

(3) Criar novas embarcações de acordo com a escolha do jogador, mantendo uma lista de todos os seus barcos.

(4) Reportar-se à entidade *partida* indicando que todos seus barcos já haviam chegado ao destino.

(5) Receber dano de outros barcos.

(6) Informar a entidade *porto* que foi destruído.

Em (1), além de inicializar a tabela de parâmetros com a assinatura, o porto deveria ajustar a direção para qual as imagens dos barcos seriam espelhadas, isso é, se o porto estivesse à esquerda da tela, os barcos deveriam estar virados para a direita, e vice-versa. Além disso, o porto mantinha contadores como pontos de vida, quantos barcos foram fabricados e destruídos e quanto de ouro o jogador receberia por rodada.

A implementação de (2) foi através da entidade *painel*, que exibia um menu com os 3 (três) tipos de embarcações a serem criados pelo jogador. A entidade *porto* foi responsável por colocá-la em jogo e comunicar-se com ela através de sua tabela de parâmetros, sabendo qual barco deveria ser criado. Isso é, tendo criado a entidade *painel*, o componente *porto* poderia escutar ao seus parâmetros que considerasse relevante.

Para implementar (3) de maneira a garantir um único componente *barco* capaz de cuidar de todos os tipos de embarcações foi necessário, após a aparição da entidade *barco*, uma série de ajustes na mesma. Por exemplo, a imagem da embarcação era decidia

apenas no final, pois somente a entidade *porto* sabia da sua facção. Na criação de cada nova entidade representando uma embarcação, o porto guardava uma referência a essa instância em uma lista de embarcações. Por último, o componente *porto* configurava-se como um *listener* para o parâmetro *"done"*. Quando o componente *barco*, adicionado à entidade representando a embarcação, detectasse que tinha chegado ao seu destino no mapa, escreveria o valor `true` ao parâmetro *"done"* da entidade e o componente *porto* seria imediatamente notificado. Quando todos os seus barcos tiverem notificado a sua chegada ao destino, o componente *porto* escreveria `true` como valor ao parâmetro *"done"* de sua entidade (4).

Para (5) e (6), trivialmente, escutava-se ao parâmetro que informava a quantidade de dano recebido, subtraindo esse valor do total dos pontos de vida. Quando chegassem a 0, o componente *porto* escreveria `true` no parâmetro *"dead"*.

4.3.6 Componente: *partida*

O componente *partida* mantinha informações sobre as rodadas e turnos, sendo crucial para o desenrolar do jogo, implementando funcionalidades como: turnos, condições de vitória e derrota, partes interface de usuário, inicializar as entidades do porto, etc.

As competências desse componente são:

(1) Inicializar-se através da assinatura da entidade juntamente com sua tabela de parâmetros.

(2) Criar e inicializar partes das entidades *"porto-p1"* e *"porto-p2"*.

(3) Coordenar turnos, rodadas, vitórias e derrotas.

(4) Manter meta-informações e partes da interface de usuário.

Embora a lista de responsabilidades seja menor, são as de maior importância para o fluxo do jogo. Para implementar (1), o componente *partida* inicializa o parâmetro para manter o estado de pausa (mantendo o jogo em suspensão se estivesse pausado), além de se colocar como *listener* das teclas *tab*, *enter/return* e *ESC*, que, respectivamente, causa a exibição dos dados parciais da partida, indica o fim do turno do jogador e mostra o menu de pausa.

A implementação de (2) envolveu construir as entidades mencionadas e mantê-las como referência para indicar de qual jogador era o turno. Isso é, internamente, o componente *partida* usou as entidades de porto como representação do jogador, estabelecendo-se como observador do parâmetro *"done"* dos portos para saber se as embarcações daquele jogador já haviam chegado ao seu destino na fase de simulação. Assim que os dois portos reportassem o fim da movimentação de suas unidades, o componente *partida* entrava na fase de planejamento. Nessa fase, quando o jogador decidia que já havia dado os comandos necessários, pressionava a tecla *enter*, indicando que o turno do próximo jogador deveria acontecer. Quando um dos portos indicasse a derrota usando o parâmetro *"morto"* o componente partida traria a cena de fim de jogo, completando a implementação de (3).

Para implementar (4), durante todo o processo mencionado acima, meta-informação como a quantidade de barcos em jogo e destruídos foi mantida, junto com a quantidade de ouro recebida por cada jogador. Essas informações são visualizadas através da tecla *tab*.

4.4 Discussão

A fim de avaliar a flexibilidade do modelo de componentes da Gear2D, os componentes *Pathfinder2D* e *lua-proxy* foram criados. Ambos apresentam um nível de complexidade relativamente alto e que usualmente requerem alterações no núcleo do motor para dar suporte às respectivas funcionalidades.

O componente *Pathfinder2D* toma uma abordagem um tanto complexa para resolver o problema de planejar uma trajetória, mas explora seus benefícios: é capaz de funcionar em espaços contínuos, lida com um número reduzido de nós e resolve mudanças localmente. Essa abordagem encaixa bem com o posicionamento de entidades e objetos usados na Gear2D e também com sua representação de espaço. O uso do padrão *Observer* como meio de interação entre componentes da Gear2D provê meios para a simples integração e comunicação com outros componentes.

Linguagens de *scripting* são importantes no processo de desenvolvimento de jogos pois permitem que novas *features* sejam prototipadas e ajustadas com rapidez. Um motor de jogos com suporte a *scripting* pode facilitar bastante o ciclo de desenvolvimento de um jogo, sendo portanto uma necessidade nos jogos modernos.

O desenvolvimento do componente *lua-proxy* foi simples e mostrou que a Gear2D é versátil o suficiente para ter outros componentes capazes de fazer uma ponte com outras linguagens, embora a complexidade da implementação claramente depende no esforço necessário para integrar a linguagem escolhida e o motor. A diferença entre o sistema de tipos forçou uma simulação parcial de tipagem estática em Lua, entretanto esse problema pode ser minimizado em linguagens que o sistema de tipos seja mais próximo do sistema da linguagem C++.

O sucesso no desenvolvimento desses dois componentes indica que o modelo de componentes da Gear2D pode ser versátil o suficiente para que funcionalidades de complexidade arbitrária sejam integradas ao motor sem mudanças na *API* ou recompilações do código interno. Essa característica é de vital importância no processo de desenvolvimento de jogos, permitindo que novas funcionalidades sejam introduzidas com custo reduzido.

Além disso, o desenvolvimento do jogo *Naval Warfare* mostrou que a *Gear2D* cumpre seu papel como motor de jogos, permitindo e auxiliando o desenvolvimento de jogos por equipes multidisciplinares e de níveis de conhecimento diferentes. Seu modelo de componentes incentivou a divisão e a quebra de comportamentos em componentes desacoplados, sendo capaz de concretizar, através desses, um jogo eletrônico de maior escala.

Entretanto, durante o desenvolvimento dos componentes centrais *partida*, *porto* e *barco*, foram enfrentadas diversas situações que requeriam soluções emergenciais para implementar casos de uso não previstos no *design* da Gear2D e superar alguns efeitos colaterais causados por imposições do seu modelo de componentes.

A Gear2D mantém como tipos opacos ao desenvolvedor a classe *container* de componentes, chamada *object* e as classes de componentes alheios (por exemplo, o componente *collider2d* desconhece o tipo do componente que mantém parâmetros de posicionamento, *spatial2d*). Além disso, a tabela de parâmetros reside em um espaço desconhecido para o desenvolvedor, impedindo acesso se não pelos métodos *read()*, *write()* e seus derivados. Essas medidas foram pensadas com o intuito de proteger o desenvolvedor, mantendo a consistência da tabela de parâmetros enquanto permite implementações independentes de componentes.

Enquanto esse formato de interação serviu o propósito de manter a consistência de dados entre componentes, impôs restrições ao compartilhamento livre de dados entre componentes em entidades distintas. Isto é, embora permitisse que um componente se comunicasse através da tabela de parâmetros de outra entidade, essa solução é pouco escalável, tornando o compartilhamento de dados entre um grande conjunto de componentes (ou entre todos eles) uma tarefa no mínimo tediosa.

Em motores de jogos tradicionais onde se tem acesso direto aos repositórios de dados ou pode-se acessar diretamente dados em outros componentes, é simples eleger uma classe através do padrão *Singleton* para ser o repositório de dados "globais". Mesmo quando o uso desse padrão é possível, o seu uso descontrolado pode introduzir problemas, como os relacionados à sua ordem de inicialização [7]. Além disso, o uso de uma instância de acesso global não é *thread-safe*, exigindo trabalho necessário do desenvolvedor para gerenciar mecanismos de exclusão mútua externos (por exemplo, *"Double-Checked Locking"*) [35, 26]. Por último, a solução de um repositório de dados usando variáveis de classe ou instância introduzem recompilações e possíveis refatorações sempre que um novo atributo for introduzido.

A documentação da Gear2D até então era apenas o resultado da geração automática produzida pelo software *Doxygen*. Embora a *API* do motor seja repleta de comentários descritivos (a classe base dos componentes possui aproximadamente 30 métodos públicos e o mesmo arquivo tem mais 300 comentários usados na geração de sua documentação), a falta de exemplos mais complexos e/ou úteis certamente foi uma barreira no início do desenvolvimento, tornando a curva de aprendizagem da Gear2D ingrime.

Outro aspecto que foi notado durante o processo de criação do *Naval Warfare* é a falta de um kit de desenvolvimento para desenvolvedores iniciantes, evitando enfrentar os problemas da configuração do ambiente de desenvolvimento.

Naval Warfare foi submetido e aceito no XI Festival de Jogos Independentes para o Simpósio Brasileiro de Jogos e Entretenimento Digital (SBGames) em 2012.

Capítulo 5

Conclusões e Trabalhos Futuros

O uso de arquiteturas baseadas em componentes ao invés de hierarquias de entidades para definir comportamento já é um conceito bem aceito. Entretanto, para verdadeiramente tornar os componentes reusáveis, deve-se reduzir a quantidade de referências diretas entre eles, prática usualmente permitida mas não explorada em motores de jogos baseados em componentes (e.g, Unity3D).

A Gear2D é um motor de jogos que incorpora o reuso e desacoplamento no cerne de sua concepção, implementando um modelo de componentes que permite a livre comunicação enquanto reduz o acoplamento e *overhead* introduzido pelos métodos mais usados (chamada de funções e troca de mensagens). Esse *framework* se vale do uso de padrões e abordagens como *AOM* e *Blackboard*, expondo ao desenvolvedor funcionalidades como adaptação de entidades (incluindo troca de comportamento) em tempo de execução, escuta de parâmetros através do padrão *Observer*, entre outros. A construção do motor e de seu modelo de componentes foi explorado e testado brevemente com jogos simples como o *Pongroids* e o *ReIgnice*

Para avaliar a real flexibilidade do modelo de componentes, duas funcionalidades muito úteis foram incorporadas a sua biblioteca de componentes: *scripting* e *pathfinding*, através dos componentes *lua-proxy* e *Pathfinder2D*. O primeiro permite que novos componentes sejam criados usando a linguagem de *scripting* Lua tal que a comunicação entre esses e componentes criados em C++ seja transparente, enquanto o segundo concede a unidades a habilidade de encontrar uma trajetória entre obstáculos de maneira eficiente. Ambas as características são de grande valor para o desenvolvimento de jogos e demonstram que o modelo de componentes da Gear2D permite novas funcionalidades sem que seja necessário alterações no framework.

O sistema de comunicação entre componentes numa mesma entidade se mostrou efetivo no desenvolvimento do *Naval Warfare*, permitindo que componentes fossem desenvolvidos em paralelo e em plataformas diferentes, com mínimo esforço de de integração. Entretanto, a Gear2D não possuía um sistema de comunicação entre um grupo de componentes em entidades distintas de maneira eficaz. Outro obstáculo enfrentado foi a falta de componentes para detecção de colisão mais robustos. Percebeu-se também a dificuldade inicial em configurar um ambiente de desenvolvimento apropriado, dado que a instalação da Gear2D é pouco usual e o processo de criação de componentes depende do sistema de *build* CMake que por sua vez exige que regras sejam escritas em sua própria linguagem, demandando que desenvolvedores aprendam uma nova linguagem.

Para os problemas enfrentados em relação à configuração de um ambiente de desenvolvimento, está sendo desenvolvido um *Software Development Kit (SDK)* juntamente com exemplos, preparados para o desenvolvedor iniciante. Esse *SDK* será composto de (a) *scripts* que automatizam a descoberta do caminho de instalação da *Gear2D*, (b) assistentes que auxiliam a criação de novos componentes, (c) códigos de exemplos para diversas funcionalidades e (d) automatização da criação de um pacote final do jogo, contendo o executável do motor e suas bibliotecas necessárias, juntamente com os componentes utilizados. Espera-se que essa *SDK* facilite o trabalho de desenvolvedores principiantes na criação de jogos usando a Gear2D.

Um novo componente de detecção está sendo implementado para contornar a limitação da quantidade de *colliders* por entidade do componente existente, suportando um número arbitrário de geometrias de colisão, permitindo inclusive a detecção de intersecção entre *bounding boxes* e *bounding circles*. Esse componente virá acompanhado de um componente para simulação de corpos rígidos mais robusto e eficiente, com suporte a momento angular.

Em paralelo a essas implementações, o motor de jogos está sofrendo alterações na sua *API* pública para suportar a comunicação compartilhada entre componentes em entidades distintas de maneira transparente, através de um repositório de dados escalável capaz de guardar e compartilhar parâmetros de tipos abstratos. Essa *API* vai seguir os mesmos moldes da existente (*write()* e *read()* para parâmetros locais à entidade), também suportando o padrão *Observer*.

Planeja-se ainda criar um editor de entidades e de cenário para permitir a criação de jogos de maneira interativa, dispondo de interface gráfica para configurar e posicionar entidades. Componentes futuros incluem suporte à rede, replicando de maneira transparente tabelas de parâmetros de entidades, efetivamente sincronizando-as entre diferentes *hosts*, permitindo a criação de jogos multiusuários como *MMORPGs*. Planeja-se também disponibilizar aos desenvolvedores componentes com suporte a mapas criados através de conjuntos de *tiles*.

Gear2D e seus componentes são livremente distribuídos em `http://gear2d.com`, juntamente com sua documentação e código fonte, em licença LGPLv3. O leitor está convidado a participar do seu processo de desenvolvimento e testes através da página `http://github.com/Gear2D/`.

Referências

[1] D.J. Anderson. *Kanban*. Blue Hole Press, 2010. 31

[2] V. G.M.M.I. Aracic and K. Ostermann. Overview of CaesarJ. *Transactions on Aspect-Oriented Software Development*, 3880:135–173, February 2006. 2

[3] P. Biggar, E. de Vries, and D. Gregg. A practical solution for scripting language compilers. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 1916–1923, New York, NY, USA, 2009. ACM. 28

[4] S. Bilas. A Data Driven Game Object System. http://scottbilas.com/files/2002/gdc_san_jose/game_objects_paper.pdf, 2002. Acessado em agosto/2011. 2, 17

[5] J. Blanchette and M. Summerfield. *C++ GUI Programming with Qt 4*. Prentice Hall Open Source Software Development Series, 2008. 17

[6] T. Chatfield. Videogames now outperform hollywood movies, 2009. 1

[7] M.P. Cline, G. Lomow, and M. Girou. *C++ FAQs*. Pearson Education, 1998. 38

[8] I.D. Craig. Formal Techniques in the Development of Blackboard Systems. *IJPRAI*, 7(2):197–219, 1993. 17

[9] Crytek. CryENGINE 3. http://www.crytek.com/cryengine, 2013. Acessado em julho/2013. 28

[10] L. de Freitas, R. Bonifacio, and C. Castanho. Gear2D: um motor extensivel de jogos baseado em componentes. *X Simposio Brasileiro de Games e Entretenimento Digital*, 2011. 28

[11] L.G. de Freitas, L.M. Reffatti, I.R. de Sousa, A.C. Cardoso, C.. Castanho, R. Bonifácio, and G.N. Ramos. Gear2D: an extensible component-based game engine. In *Proceedings of the International Conference on the Foundations of Digital Games*, FDG '12, pages 81–88, New York, NY, USA, 2012. ACM. 25

[12] D. Demyen and M. Buro. Efficient triangulation-based pathfinding. In *Masters Abstracts International*, volume 45, 2006. 26

[13] Blizzard Entertainment. World of Warcraft. http://us.battle.net/wow/en/, 2013. Acessado em julho/2013. 28

[14] E. Gamma, R. Helm, R.E. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. 4, 19

[15] J. Gosling. *The Java language specification*. Prentice Hall, 2000. 2

[16] J. Gregory. *Game Engine Architecture*. A K Peters, Ltd, 2009. 1, 3, 5

[17] J. Gregory. What Is a Game Engine? In *Game Engine Architecture*, page 11. A K Peters, Ltd, 2009. 5

[18] M. Haller, J. Zauner, and J. Hartman. A generic framework for game development. *ACM SIGGRAPH and Eurographics Conference*, 2002. 21

[19] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, 1968. 26

[20] R. Ierusalimschy, W. Celes, and L.H. de F. Lua 5.1 Reference Manual. `http://www.lua.org/manual/5.1/manual.html`, 2011. Accessed in december/2011. 29

[21] M. Kallmann, H. Bieri, and D. Thalmann. Fully dynamic constrained delaunay triangulations. *Geometric Modelling for Scientific Visualization*, 3, 2003. 26

[22] C.G. Lasater. *Design Patterns*. Wordware Publishing, 2006. 17

[23] LÖVE. LÖVE. `https://love2d.org/`, 2013. Acessado em julho/2013. 28

[24] R. C. Martin. Java and C++: a critical comparison. 9(1):42–49, January 1997. 9

[25] L. McCulloch, A. Hofman, J. Tulip, and M. Antolovich. RAGE: a multiplatform game engine. In *Proceedings of the second Australasian conference on Interactive entertainment*, IE 2005, pages 129–131, Sydney, Australia, Australia, 2005. Creativity & Cognition Studios Press. 6

[26] S. Meyers and A. Alexandrescu. C++ and the Perils of Double-Checked Locking. *Dr. Dobb's Journal*, July 2004. 38

[27] A. Nareyek. Ai in computer games. *Queue*, 1(10):58–65, February 2004. 25

[28] F.P. Preparata and M.I. Shamos. *Computational geometry: an introduction*. Springer, 1985. 26

[29] PushButton Labs. Push Button Engine. `http://www.pushbuttonengine.com`, 2011. Acessado em agosto/2011. 2

[30] S. Rabin et al. Game Architecture. In *Introduction to Game Development*, Game Development Series, chapter 3, pages 267–297. Course Technology, 2008. 5, 8, 29

[31] S. Rabin et al. Programming Fundamentals. In *Introduction to Game Development*, Game Development Series, chapter 3, pages 221–251. Course Technology, 2008. 8, 9, 10

[32] L.M. Reffatti. DeltaPath: Um módulo simples para particionamento espacial e cálculo de trajetória, 2013. Disponível em Julho de 2013. 27

[33] C. Reynolds. Steering Behaviors For Autonomous Characters. *Miller Freeman Game Group, San Francisco, California*, pages 763–782, 1999. 27

[34] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable Units of Behaviour. In *ECOOP 2003*, pages 327–339, 2003. 2

[35] D.C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-oriented software architecture. Vol. 2, Patterns for concurrent and networked objects.* John Wiley, Chichester, England, 2000. 38

[36] P. Schuytema. *Design de games: uma abordagem prática.* Cengage Learning, 2008. 31

[37] K. Schwaber. Scrum development process. In *Proceedings of the 10th Annual ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA*, pages 117–134, 1995. 31

[38] R.P. Silva and R.T. Price. A busca de generalidade, flexibilidade e extensibilidade no processo de desenvolvimento de frameworks orientados a objetos. In *Proceedings of Workshop Iberoamericano de Engenharia de Requisitos e Ambientes de Software (IDEAS'98)*, pages 298–309, Abril 1998. 6

[39] G.B. Singh. Single versus multiple inheritance in object oriented programming. *SIGPLAN OOPS Mess.*, 5(1):34–43, January 1994. 2

[40] Unity Technologies. Unity3D. http://www.unity3d.com, 2011. Acessado em agosto/2011. 2

[41] P. Tozour. Search Space Representations. *AI Programming Wisdom 2*, pages 85–101, 2003. 26

[42] M. West. Evolve your Hierarchy. http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy, 2007. Acessado em agosto/2011. 2, 10, 12

[43] W. White, C. Koch, J. Gehrke, and A. Demers. Better scripts, better games. *Commun. ACM*, 52:42–47, March 2009. 28

[44] Wikipedia. Atari. "http://en.wikipedia.org/wiki/Atari", 2011. Acessado em agosto/2011. 22

[45] J.W. Yoder and R. Johson. The adaptive object-model architectural style. In *WICSA*, 2002. 17