4
Arquitetura e Linguagens para Jogos

Jogos de computador requerem alto desempenho, o que tem levado os profissionais a adotarem linguagens de programação tais como, C++, C# e Java para o desenvolvimento de jogos de grande porte.

A grande maioria dos motores de jogos são desenvolvidos em linguagem C++, e mais recentemente em C# ou Java. Essas linguagens são eficientes e permitem que o motor ofereça uma interface orientada a objetos, o que facilita o desenvolvimento dos jogos.

Este capítulo apresenta a arquitetura do motor de jogos e a linguagem de programação utilizada neste trabalho. Inicialmente é apresentada uma descrição e a arquitetura dos motores de jogos. Em seguida aborda-se o tópico de linguagens de *script* e por fim, apresenta-se Lua, a linguagem de *script* utilizada na implementação do *framework*.

4.1
Engine

Motor de jogo é um *framework* indispensável para o desenvolvimento de uma determinada classe de jogos. Em outras palavras, motores implementam funcionalidades e recursos comuns à maioria dos jogos de determinado tipo, permitindo que esses recursos sejam reutilizados a cada novo jogo criado [47]. Um motor de jogos deve atender, no mínimo, as seguintes necessidades: inteligência artificial de todos os objetos de jogo (*game objects*), física e renderização (processo de geração de informações – sonoras e gráficas – que serão transmitidas ao usuário final).

Em jogos, a função da inteligência artificial é imitar o comportamento racional dos seres humanos, simulando, através de algoritmos e heurísticas, o processo de raciocínio e ponderação da mente humana. O objetivo é dar realismo aos comportamentos dos personagens do jogo, isto é, seus atos, suas reações e seus diálogos devem convencer o jogador de que são personagens que pensam. A função da inteligência artificial em um motor de jogos é dar facilidade ao desenvolvedor para implementar algoritmos que trazem realismo ao jogo, como busca de caminhos e aquisição de conhecimento, tomada de

decisão, entre outros [48].

A função da Física é calcular o movimento, rotações e resposta de colisão de corpos rígidos aplicando física realista neles. Um motor de jogos de física usa as propriedades dos objetos como momento, torque ou elasticidade para simular comportamento de corpos rígidos. Isto não apenas produz resultados mais realistas, como também facilita o trabalho do desenvolvedor de escrever um *script* de comportamento.

A função da renderização em um motor de jogos é reduzir o processamento de objetos gráficos em uma cena com técnicas algorítmicas.

Os motores de jogos liberam o programador da configuração de detalhes de baixo-nível, relativos a hardware (placas gráficas, mouse, teclado, joystick, áudio), e é isto que provê a rapidez e facilidade de uso e de implementação.

Com o aumento do tamanho e complexidade dos jogos, os motores de jogos passaram a desempenhar um papel cada vez mais importante no processo de construção dos mesmos, pois são utilizados para abstrair diversos detalhes de implementação. Se um motor foi desenvolvido para criar um determinado gênero de jogo (ação, RPG, luta), ele poderá ser utilizado no desenvolvimento de vários jogos do mesmo gênero, porém com objetivos, enredos e cenários diferentes.

Assim sendo, a maior vantagem de usar um motor de jogos é que, se for construído em uma arquitetura modular, ele pode ser reutilizado para criar vários jogos diferentes entre si, que só iriam usar os módulos necessários do motor de jogos. Além disso, proporciona uma redução significativa no ciclo de vida de desenvolvimento deste tipo de aplicação. Os motores fizeram com que a etapa de programação do jogo fosse um processo mais automatizado, permitindo uma concentração maior nas atividades mais específicas do jogo, como seu roteiro, o comportamento de seus personagens, seus cenários e sua trilha sonora, entre outras.

O motor de jogos em C++, C++Play, teve que ser adaptado para receber a camada em Lua. O núcleo do C++Play é resultado de projetos anteriores do Laboratório VisionLab/Puc-Rio.

4.2 Arquitetura do motor C++Play

A arquitetura do motor foi projetada de acordo com o princípio de encapsulamento do paradigma de orientação a objetos (OO) e implementada na linguagem C++. Este motor de jogos 2D foi projetado para possuir uma arquitetura que permita a flexibilidade e a reusabilidade de seus componentes,

possibilitando a expansão de suas funcionalidades. A arquitetura geral do motor de jogos 2D proposto neste trabalho é apresentada na Figura 4.1.

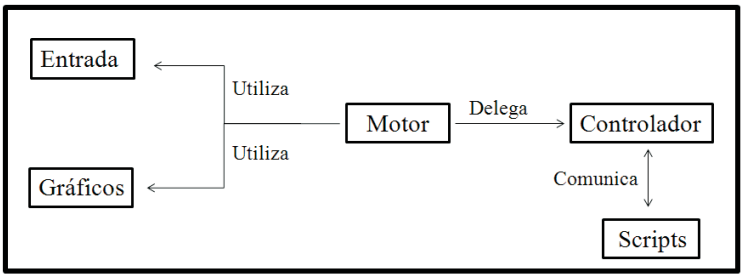

Figura 4.1: Arquitetura Geral do motor de jogos 2D C++Play. O motor se comunica com o sistema de *scripts* e utiliza os sistemas gráfico e de entrada.

Na arquitetura do C++Play, o motor de jogos se divide em módulos responsáveis por tarefas específicas, chamadas sistemas. Tais sistemas possuem diferentes representações e podem ser definidos por um conjunto de classes ou por uma classe apenas. Os sistemas são definidos apenas para fornecer uma separação de tarefas e uma organização de alto nível para o motor de jogos. O sistema de aplicação é o sistema central. Ele se comunica com o sistema de *scripts* e utiliza os sistemas gráfico e de entrada. A seguir, apresentam-se as principais características do motor de jogos C++Play.

O sistema Motor é responsável pelo controle do *loop* principal do jogo. Ele acessa outros sistemas que realizam tarefas específicas. O sistema de aplicação é responsável pela inicialização do programa e dos outros sistemas, assim como do término do programa e finalização correta dos outros sistemas. Ele define um conjunto de funções predefinidas que permitem ao programador de aplicação controle sobre a inicialização, o término e as ações ocorridas para cada frame do jogo.

O sistema de aplicação possui uma classe principal *Singleton* que permite que as diferentes partes do sistema possuam visibilidade das classes mais importantes como o sistema gráfico e o sistema de entrada e saída chamado *GameEngine*. *Singleton* é um *design pattern* (padrão de projeto[1]) que tem como objetivo garantir que exista apenas uma instância de uma certa classe a qualquer instante e em qualquer ponto de um sistema. A estrutura do Singleton exibida na Figura 4.2 mostra que esse padrão de projeto é um método estático (Instance()) que permite que clientes acessem sua instância única.

[1]Padrões de projeto são soluções elegantes e reutilizáveis para problemas recorrentes que encontram-se diariamente no processo de desenvolvimento de aplicativos para o mundo real. Os padrões são geralmente definidos como soluções já testadas para problemas recorrentes. O termo refere tanto à descrição de uma solução que se pode ler quanto a uma instância daquela solução utilizada para resolver um problema particular. A obra de Gamma *et al.* [49] é uma referência clássica a padrões de projeto.

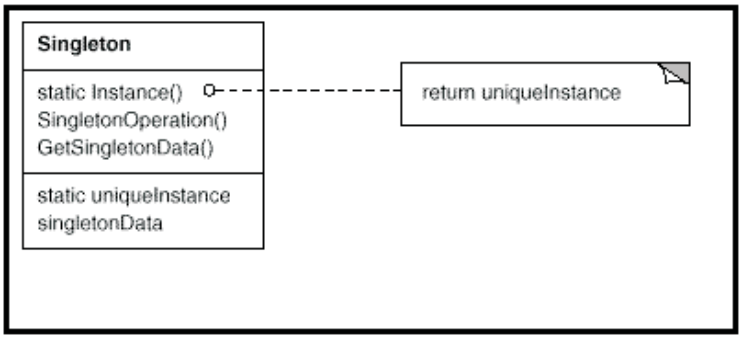

Figura 4.2: Estrutura do Singleton

A inicialização e as funções executadas ao término da aplicação são associadas a uma interface denominada GameStateController. A classe *GameStateController* é abstrata[2] e permite a execução de lógica arbitrária. Como o sistema de aplicação, ela divide a execução em *time slice*s, permitindo ao programador o controle de eventos específicos como processamento de frame.

O sistema de estado é o principal responsável pela flexibilidade do sistema. Enquanto a classe de aplicação encapsula as funções de baixo nível associadas à plataforma e aos métodos globais de controle, o sistema de estado permite que o programador tenha um controle fino das ações que deverão ser apresentadas ao usuário final.

O motor de jogos provê também uma classe base utilizada para qualquer objeto que necessite de lógica específica de atualização. *GameObject* é uma classe abstrata que define um objeto com posição no espaço e a capacidade de receber mensagens de atualização e realizar operações de desenho customizadas.

O sistema de entrada permite acesso ao estado do teclado e *mouse* para uma determinada aplicação. As classes *Keyboard* e *Mouse* atuam como gerenciadores de entrada, recebendo eventos do usuário quando ele digita/clica alguma tecla do dispositivo. *Keyboard* verifica se existem métodos atribuídos a cada tecla pressionada e, quando existir, chama os objetos responsáveis pela ação de acordo com cada tecla. E a classe *Mouse* retorna a posição x e y do mouse e verifica se algum botão foi pressionado.

O sistema gráfico se divide em duas camadas básicas: *Graphics Manager* e a *Scene Manager*. A camada *Graphics Manager* é responsável pela interação do motor de jogos com as camadas de desenho da plataforma. Ela é definida para ser o mais leve possível, sendo capaz de permitir a integração dos métodos de desenho em um método controlado pelo sistema de aplicação. A classe *Scene*

[2]Classes abstratas são modelos de classes, então, não podem ser instanciadas diretamente com o new, elas sempre devem ser herdadas por classes concretas

define uma abstração de alto nível para criação de jogos baseados em mundos formados por blocos conhecidos como *Tiles*. Esse modelo de organização de mundo é o mais encontrado em jogos 2D. Essa classe divide o mundo em sub-camadas distintas: *Backdrops*, *TileLayer* e *Overlays*. *Backdrops* são os cenários de fundo onde o usuário possui pouca ou nenhuma interação, servindo na maioria das vezes como itens de adorno gráfico. *TileLayer* é a sub-camada que apresenta o conjunto de blocos que define o mundo ao qual um avatar é inserido. Ela fornece métodos para detecção de colisão com os blocos do mundo. E a sub-camada de *Overlays* representa o local onde os *sprites* dos objetos dinâmicos de jogo, controlados pelo jogador, ou inteligência artificial são desenhados. Esses objetos são representados por sprites. Um *overlay* pode ser entendido como uma camada inicialmente transparente onde os *sprites* do avatar ou inimigos são desenhados.

A classe *Sprite* representa os principais objetos gráficos a serem manipulados em jogos 2D. Os *sprites* representam em geral todas as poses que um objeto pode ficar em suas animações. Sem *sprites* animados não seria possível realizar as complexas animações de personagens vistas em jogos 2D. Por exemplo, uma caminhada pode ser representada através de uma alternância entre um *sprite*, que exibe o personagem pisando com o pé direito, seguido por outro *sprite* no qual o mesmo personagem esteja pisando com o pé esquerdo, dando uma sensação de que ele está caminhando. A Figura 4.3 retrata o exemplo acima.

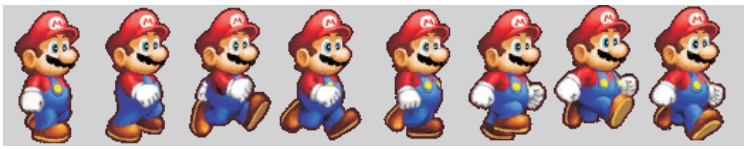

Figura 4.3: Representação de uma caminhada do jogo super Mario World [50], reproduzida sob fair use policy

Por fim, o sistema *scripts* se comunica com o controlador para ter acesso aos métodos e objetos implementados pelo motor. Este sistema representa o código fonte do jogo no qual contém a lógica do jogo. A Figura 4.4 apresenta o diagrama de classes do motor de jogos C++Play.

A arquitetura geral do motor de jogos 2D proposto neste trabalho é apresentada na Figura 4.1.

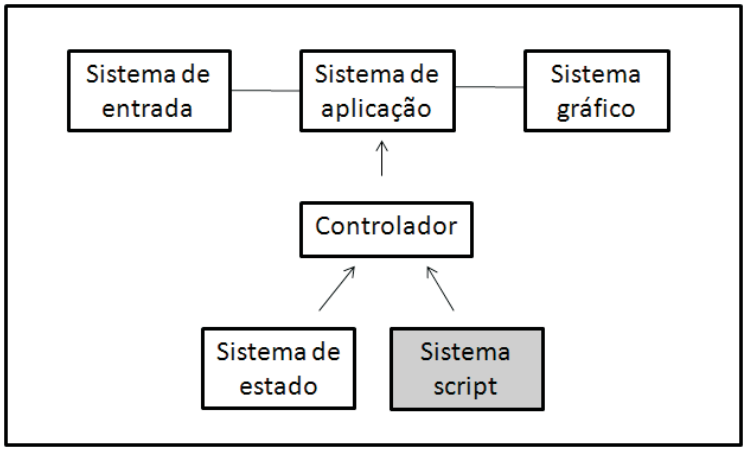

Figura 4.4: Diagrama de classes

4.3 Linguagens de script

Os *script*s provêem uma maneira de modificar certas funcionalidades do jogo (ou definições de dados) sem precisar compilar novamente o código-fonte da aplicação, após as alterações. O objetivo é isolar essas partes, para que possam ser editadas independentemente.

De acordo com [51], a vantagem de se usar linguagens de *script*s é poder executar os *script*s em um ambiente isolado da aplicação. Dessa forma, é possível que um *script* mal feito seja desativado e/ou impedido de ser executado, o que contribui para aumentar a segurança da aplicação. Além disso, o desenvolvimento da aplicação torna-se mais flexível, pois partes do programa controladas por *script*s podem ser implementadas de maneira independente, por pessoas que não estejam envolvidas com o desenvolvimento principal. Em outras palavras, essas pessoas não precisam requisitar que a aplicação seja compilada novamente, a cada alteração, para que os resultados possam ser conferidos.

As linguagens de *script* fornecem usualmente um nível de abstração maior que linguagens estáticas como C++ e Java [52], podendo ser usadas de forma mais simples, facilitando, para o programador, o uso de funcionalidades e componentes pré-existentes. Tipicamente, essas linguagens funcionam acopladas a programas hospedeiros implementados em linguagens compiladas tradicionais como C e C++.

A diferença entre linguagens de *script* e as linguagens de programação tradicionais (compiladas) é que os *script*s são interpretados em tempo de execução, ao passo que nas linguagens de programação tradicionais, o código-fonte é transformado em código de máquina em tempo de compilação pelos

projetistas do sistema, e a partir desse momento não pode mais ser alterado. É essa característica que permite que uma única aplicação contenha diferentes tipos de configurações e comportamentos, dependendo exclusivamente do *script* que está sendo executado por ela.

Acoplar uma linguagem de *script* em um jogo traz vários benefícios. A linguagem de *script* pode ser usada para efetivamente implementar o *script* do jogo, para definir objetos e seus comportamentos, para gerenciar os algoritmos de inteligência artificial e controlar os personagens, e ainda para tratar os eventos de entrada. Uma linguagem de *script* também desempenha um papel importante nas etapas de prototipação, teste e depuração. A escolha de uma linguagem de *script* simples permite ainda que seja dado a roteiristas e artistas acesso programável ao jogo, a fim de que eles possam experimentar novas ideias e variações. Esses profissionais conduzem a maior parte do desenvolvimento real do jogo, mas não são em geral programadores profissionais, e não estão familiarizados com técnicas sofisticadas de programação.

Atualmente existem várias linguagens de *script* disponíveis, dentre estas, pode-se citar as linguagens Lua, Ruby, Python, AngelScript, GameMonkey, Squirrel e JavaScript. Para esse projeto, a linguagem Lua foi escolhida por ser de alto nível, possuir sintaxe simples e de fácil aprendizagem, possuir, também, código aberto e um alto desempenho comparada às outras linguagens.

4.4 Lua

Lua [53] é uma linguagem de programação poderosa, rápida e leve, projetada para estender aplicações. Isso quer dizer que ela foi projetada para ser acoplada a programas maiores que precisem ler e executar programas escritos pelos usuários. Ela combina sintaxe simples, para programação procedural, com poderosas construções para descrição de dados, baseadas em tabelas associativas e semântica extensível. É tipada dinamicamente, é interpretada a partir de *bytecodes*, para uma máquina virtual baseada em registradores, e tem gerenciamento automático de memória com coleta de lixo incremental. É, atualmente, a linguagem de *script* mais usada em jogos. Essas características fazem de Lua uma linguagem ideal para configuração, automação (*scripting*) e prototipagem rápida.

Lua é uma linguagem embutida, com sintaxe semelhante à de Pascal mas com construções modernas, como funções anônimas, inspiradas no paradigma funcional, e poderosos construtores de dados. Isso faz com que Lua seja uma linguagem de grande expressão com uma sintaxe familiar.

Além disso, Lua é uma linguagem de *script* extensível, projetada para

oferecer meta-mecanismos que possibilitam a construção de mecanismos mais específicos. Com isso, é fácil adequar Lua às necessidades da aplicação, sem comprometer as suas características básicas, mantidas desde a sua criação, tais como portabilidade, pequeno tamanho e simplicidade. Por ser uma linguagem de extensão, Lua trabalha acoplada a uma aplicação hospedeira. Essa aplicação pode criar e ler valores armazenados em Lua, executar funções de Lua e registrar funções C no ambiente de Lua. As funções C registradas em Lua, por sua vez, podem ser invocadas de programas Lua. Dessa forma, podemos conciliar as facilidades de uma linguagem de *script* oferecidas por Lua com a eficiência das linguagens C e C++.