Jogo Geometry Wars

Luis Filipe Antunes Rodrigues[1]

[1]Universidade Federal do Rio Grande do Sul - UFRGS

lfarodrigues@inf.ufrgs.br

1. Introdução

O desenvolvimento de software é uma área que desafia constantemente as habilidades de programadores, exigindo a aplicação prática de conceitos avançados de programação. A motivação para este trabalho surge da necessidade de integrar teoria e prática no estudo de modelos de linguagem de programação, aplicando esses conhecimentos na criação de um jogo que simule um ambiente dinâmico e interativo, como o estilo popularizado por Geometry Wars.

O problema definido para este trabalho consiste na implementação de um jogo que não apenas recrie a jogabilidade intensa e fluida de Geometry Wars, mas que também seja construído com uma arquitetura de software robusta, utilizando princípios como orientação a objetos, herança, polimorfismo e técnicas eficientes de alocação de memória. A complexidade reside em garantir que essas estruturas proporcionem um código modular, reutilizável e otimizado, capaz de lidar com as exigências de um jogo em tempo real.

Os objetivos deste trabalho incluem a criação de um jogo funcional que implemente os conceitos teóricos abordados ao longo da disciplina, com ênfase na aplicação prática de técnicas de design de software orientado a objetos e na otimização de desempenho. Além disso, busca-se avaliar como as decisões de arquitetura influenciam diretamente a jogabilidade e a performance do jogo.

Os principais resultados alcançados demonstram que a implementação dos conceitos de orientação a objetos, herança e polimorfismo permitiu um desenvolvimento ágil e estruturado, enquanto as técnicas de alocação de memória contribuíram para a eficiência do jogo, mantendo a responsividade e a fluidez mesmo sob alta carga de processamento.

A estrutura deste relatório é organizada da seguinte forma: a Seção 2 apresenta a fundamentação teórica em programação orientada a objetos. A Seção 3 detalha a metodologia empregada no desenvolvimento do jogo, incluindo as decisões de design e a implementação dos principais componentes. Na Seção 4, são discutidos os resultados obtidos e as lições aprendidas ao longo do projeto. A Seção 5 traz as conclusões e sugestões para trabalhos futuros. Finalmente, a Seção traz a bibliografia utilizada.

2. Fundamentação Teórica

2.1. Orientação a Objetos

A orientação a objetos (OO) é um paradigma de programação que organiza o software em torno de "objetos,"que são instâncias de classes, representando tanto dados quanto comportamentos. Este paradigma tem suas raízes nos anos 1960, mas se popularizou nas décadas seguintes, especialmente com o surgimento de linguagens como Smalltalk, C++, e posteriormente Java e Python. A orientação a objetos trouxe uma abordagem modular e

intuitiva para a construção de sistemas complexos, permitindo que desenvolvedores criem estruturas de código mais próximas do mundo real.

2.1.1. Princípios Fundamentais

A orientação a objetos é sustentada por quatro pilares principais: **abstração**, **encapsulamento**, **herança** e **polimorfismo**.

- **Abstração**: Abstração refere-se ao processo de identificar os aspectos essenciais de uma entidade do mundo real e representá-los em uma classe. A abstração permite que os desenvolvedores se concentrem em características relevantes e ocultem os detalhes desnecessários para o contexto específico do problema.
- **Encapsulamento**: Encapsulamento é o princípio que protege os dados internos de uma classe, expondo apenas as interfaces necessárias para a interação com outros objetos. Isso é realizado através de modificadores de acesso, como *private*, *protected*, e *public*, que controlam a visibilidade dos atributos e métodos da classe. O encapsulamento promove a segurança e a integridade dos dados, impedindo acessos não autorizados e modificações acidentais.
- **Herança**: Herança é o mecanismo que permite que uma nova classe (subclasse) seja derivada de uma classe existente (superclasse), herdando seus atributos e métodos. Este princípio promove a reutilização de código, permitindo que funcionalidades comuns sejam definidas em uma superclasse e estendidas ou especializadas em subclasses. A herança também facilita a criação de hierarquias de classes, onde comportamentos genéricos podem ser refinados para atender a necessidades específicas.
- **Polimorfismo**: Polimorfismo permite que objetos de diferentes classes sejam tratados como instâncias de uma classe comum, geralmente através de uma interface ou uma classe base. Existem duas formas principais de polimorfismo: *sobrecarga* e *sobrescrita*. A sobrecarga permite a existência de múltiplos métodos com o mesmo nome, mas com assinaturas diferentes. A sobrescrita, por outro lado, permite que uma subclasse forneça uma implementação específica de um método que já foi definido em sua superclasse. O polimorfismo oferece flexibilidade e extensibilidade, permitindo que o código seja adaptado a novos requisitos com o mínimo de modificações.

2.1.2. Vantagens e Aplicações

A orientação a objetos oferece várias vantagens no desenvolvimento de software, incluindo modularidade, reutilização de código, e facilidade de manutenção. Em sistemas grandes e complexos, o uso de OO permite a criação de componentes independentes que podem ser desenvolvidos, testados, e modificados de forma isolada, facilitando o trabalho em equipe e a evolução do software ao longo do tempo.

Além disso, a OO é particularmente útil em simulações e jogos, onde a representação de entidades do mundo real (como jogadores, inimigos, itens, etc.) como objetos com propriedades e comportamentos distintos se alinha naturalmente com o paradigma. No

contexto do jogo desenvolvido neste trabalho, o uso de orientação a objetos foi essencial para modelar as diferentes entidades e suas interações de forma estruturada e eficiente.

2.1.3. Desafios na Implementação

Apesar de suas vantagens, a orientação a objetos também apresenta desafios. Um dos principais é a correta definição das hierarquias de classes, que pode se tornar complexa em sistemas grandes, levando ao fenômeno conhecido como "explosão de classes"ou à criação de classes com responsabilidades mal definidas. Outro desafio é o gerenciamento de memória, especialmente em linguagens que não oferecem coleta automática de lixo (*garbage collection*), exigindo que os desenvolvedores sejam cuidadosos com a alocação e desalocação de objetos para evitar problemas como vazamentos de memória.

Neste trabalho, esses desafios foram enfrentados através de um planejamento cuidadoso da arquitetura do jogo e da adoção de boas práticas de design, como o princípio da responsabilidade única (SRP) e o uso de padrões de projeto, como o padrão *Factory* para criação de objetos, garantindo que o sistema permanecesse coeso e fácil de manter.

3. Implementação e Arquitetura do Software

3.1. Linguagem de Programação Utilizada

O desenvolvimento do jogo foi realizado utilizando a linguagem de programação C++. C++ é amplamente reconhecida por sua eficiência e controle sobre recursos de hardware, o que a torna uma escolha ideal para o desenvolvimento de jogos. Entre os principais benefícios do uso de C++ estão:

- **Desempenho**: C++ permite a programação de baixo nível, possibilitando a otimização do código para uso eficiente da CPU e da memória. Isso é crucial em jogos, onde a performance pode impactar diretamente na experiência do usuário.
- **Controle sobre Recursos**: A capacidade de gerenciar manualmente a alocação e desalocação de memória em C++ permite um controle preciso dos recursos, evitando desperdícios e maximizando a eficiência, o que é particularmente importante em ambientes com recursos limitados.
- **Portabilidade**: C++ é uma linguagem altamente portátil, suportada por diversas plataformas e sistemas operacionais, facilitando a adaptação do jogo para diferentes ambientes.
- **Suporte a Paradigmas Múltiplos**: C++ suporta múltiplos paradigmas de programação, como a programação orientada a objetos, a programação genérica e a programação procedural, permitindo maior flexibilidade no design do software.

Essas características fazem do C++ uma escolha natural para o desenvolvimento de jogos, especialmente em projetos que requerem alto desempenho e controle detalhado sobre o hardware.

3.2. Arquitetura do Software

A arquitetura do software foi cuidadosamente planejada para atender às necessidades de um jogo dinâmico e de alta performance. O núcleo do sistema é um motor gráfico personalizado baseado em DirectX, que foi desenvolvido em outro momento do curso. Esse motor gráfico é um pacote completo que oferece suporte a diversas funcionalidades essenciais para o desenvolvimento de jogos.

3.2.1. Motor Gráfico

O motor gráfico desenvolvido inclui uma série de módulos que cobrem as principais necessidades de um jogo:

- **Renderização de Gráficos e Texto**: O motor inclui funções para renderização eficiente de gráficos 2D e 3D, bem como suporte para exibição de texto. A integração com DirectX garante que a renderização seja realizada de maneira otimizada, utilizando recursos gráficos avançados.
- **Áudio**: O motor possui suporte integrado para reprodução de áudio, permitindo a implementação de efeitos sonoros e música de fundo, fundamentais para a imersão do jogador.
- **Cálculos de Física e Tipos de Dados**: O motor oferece um módulo para cálculos de física básicos, necessários para a simulação de movimentos e colisões. Além disso, inclui tipos de dados personalizados para facilitar operações matemáticas comuns em jogos, como vetores e matrizes.
- **Classe `Object`**: A classe `Object` é a base para todos os objetos do jogo. Ela pode ser herdada para criar objetos específicos, e inclui métodos virtuais como `update()` e `draw()`, que são sobrecarregados pelas subclasses para definir o comportamento e a aparência dos objetos.
- **Interface `Game`**: A interface `Game` define a estrutura básica de um jogo, incluindo métodos virtuais para `initialize()`, `update()`, `draw()` e `finalize()`. Essa interface garante que todos os jogos desenvolvidos com o motor sigam um padrão consistente, facilitando o desenvolvimento e a manutenção.
- **Classe `Scene`**: A classe `Scene` atua como um contêiner para objetos do jogo, organizando-os em diferentes cenas. Isso permite a separação lógica de diferentes partes do jogo (por exemplo, menus, níveis) e facilita a transição entre elas.
- **Classe `Engine`**: A classe `Engine` é responsável pela configuração e inicialização do jogo. Ela gerencia a tela do jogo e executa o loop principal do jogo, que inclui a atualização dos objetos, a renderização e o cálculo do tempo de frame. A `Engine` inicia o jogo a partir de uma instância da interface `Game`, garantindo que as funções de `update()`, `draw()` e `calculateFrameTime()` sejam chamadas no ciclo adequado. Isso proporciona uma estrutura sólida para o controle do fluxo do jogo e a integração dos diferentes módulos.

3.3. Principais Funções e Algoritmos

A seguir, são descritas as principais funções e algoritmos implementados no jogo em formato de pseudo-código:

3.3.1. Sistema de Colisão

O sistema de detecção de colisão é essencial para a jogabilidade, garantindo que as interações entre os objetos do jogo sejam realistas e respondam adequadamente às ações do jogador. O algoritmo de detecção de colisões utiliza *Bounding Boxes*, permitindo a verificação eficiente de interseções entre objetos.

```
bool checkCollision(Rectangle a, Rectangle b) {
```

```
    return (a.x < b.x + b.width &&
            a.x + a.width > b.x &&
            a.y < b.y + b.height &&
            a.y + a.height > b.y);
}
```

3.3.2. Sistema de Gerenciamento de Entidades

O sistema de gerenciamento de entidades controla a criação, atualização e destruição de todos os objetos no jogo. Ele garante que cada entidade seja processada de forma eficiente e que recursos sejam liberados quando não forem mais necessários.

```
void updateEntities(List<Entity> entities) {
    for (Entity entity : entities) {
        entity.update();
        if (entity.isDestroyed()) {
            entities.remove(entity);
        }
    }
}
```

3.3.3. Sistema de Partículas

O sistema de partículas é responsável por efeitos visuais como explosões, fumaça, faíscas, e outros elementos que enriquecem a experiência visual do jogo. Esse sistema permite a criação e manipulação de milhares de partículas em tempo real, com comportamento individual definido por regras como velocidade, direção, e tempo de vida.

```
void updateParticles(ParticleSystem& ps) {
    for (Particle& p : ps.particles) {
        p.position += p.velocity;
        p.life -= 1;
        if (p.life <= 0) {
            p.isAlive = false;
        }
    }
    ps.removeDeadParticles();
}
```

3.4. Desafios e Soluções

Durante a implementação do jogo, foram enfrentados diversos desafios técnicos, como a otimização do desempenho e a gestão de recursos. Para mitigar esses problemas, foram adotadas técnicas como *object pooling* para gerenciar a criação e destruição de objetos de maneira eficiente e otimização dos algoritmos de detecção de colisão.

3.4.1. Aplicação do Paradigma Funcional

Embora o jogo tenha sido desenvolvido predominantemente utilizando o paradigma orientado a objetos, a aplicação do paradigma funcional poderia ter oferecido várias vantagens. O paradigma funcional é baseado em funções puras, imutabilidade e funções de ordem superior, o que pode levar a um código mais limpo e modular. Algumas áreas onde o paradigma funcional poderia ser aplicado incluem:

- **Gerenciamento de Estado**: Utilizar funções puras para calcular estados e transições pode ajudar a manter o código mais previsível e livre de efeitos colaterais. Isso poderia simplificar o gerenciamento de estado de entidades e cenas, melhorando a clareza e a manutenção do código.
- **Processamento de Eventos**: A abordagem funcional poderia ser usada para tratar eventos de forma mais declarativa, utilizando funções de ordem superior para transformar e combinar eventos. Isso pode simplificar o código responsável pela lógica de eventos, como entrada do jogador e interações entre objetos.
- **Renderização e Atualização de Partículas**: O sistema de partículas poderia se beneficiar da abordagem funcional, utilizando funções puras para calcular o próximo estado das partículas. Isso poderia facilitar a criação de efeitos complexos e permitir a composição de diferentes efeitos visuais de forma modular.
- **Paralelismo e Concorrência**: O paradigma funcional é naturalmente mais adequado para programação paralela e concorrente, uma vez que funções puras e dados imutáveis evitam problemas de sincronização. Isso poderia melhorar o desempenho do jogo ao permitir a execução paralela de cálculos de física e atualizações de entidades.

  Adotar o paradigma funcional em um projeto de jogo pode exigir um ajuste significativo na abordagem de desenvolvimento, mas oferece a oportunidade de criar um código mais robusto e modular, que pode ser mais fácil de testar e manter a longo prazo.

4. Resultados Obtidos

A implementação do jogo no estilo Geometry Wars gerou uma série de resultados que demonstram a eficácia das escolhas técnicas e metodológicas empregadas no desenvolvimento. Os principais resultados obtidos são descritos a seguir:

4.1. Desempenho e Estabilidade

O jogo apresentou um desempenho robusto e estável durante os testes, mesmo em cenários com grande quantidade de objetos e partículas em tela. As técnicas de otimização, como *object pooling* e algoritmos eficientes de detecção de colisão, foram bem-sucedidas em garantir uma taxa de quadros consistente e uma experiência de jogo fluida. A utilização de C++ e DirectX contribuiu significativamente para a eficiência e o gerenciamento eficaz dos recursos.

4.2. Funcionalidade do Motor Gráfico

O motor gráfico personalizado desenvolvido para o projeto provou ser altamente funcional e flexível. As principais funcionalidades implementadas, como:

- **Renderização de Gráficos e Texto**: A renderização de gráficos 2D e 3D foi realizada de maneira eficiente, com suporte para a exibição de texto integrado. Isso proporcionou uma apresentação visual atraente e clara.
- **Áudio**: O sistema de áudio integrou de forma eficaz efeitos sonoros e música, aumentando a imersão do jogador.
- **Cálculos de Física e Tipos de Dados**: O módulo de cálculos de física permitiu uma simulação realista de movimentos e colisões, enquanto os tipos de dados personalizados facilitaram operações matemáticas complexas.
- **Classe `Object`, Interface `Game` e Classe `Scene`**: Essas estruturas proporcionaram uma base sólida para a criação e gerenciamento de objetos e cenas, permitindo uma organização clara e modular do código.
- **Classe `Engine`**: A classe `Engine` gerenciou com sucesso a configuração e inicialização do jogo, além de executar o loop principal do jogo de maneira eficiente.

4.3. Sistema de Partículas

O sistema de partículas implementado foi eficaz na criação de efeitos visuais dinâmicos, como explosões e fumaça. O gerenciamento em tempo real das partículas, incluindo a atualização e remoção, foi realizado de forma eficiente, mantendo a performance do jogo estável e permitindo uma experiência visual atraente.

4.4. Interface e Usabilidade

A interface do usuário foi projetada para ser intuitiva e responsiva. Os controles foram bem recebidos durante os testes, e a disposição dos elementos na tela proporcionou uma navegação fácil e uma experiência de jogo agradável.

5. Conclusões

A implementação do jogo no estilo Geometry Wars atingiu os objetivos estabelecidos e demonstrou a eficácia das abordagens e tecnologias empregadas. As conclusões principais do projeto são:

5.1. Eficiência das Tecnologias Utilizadas

A escolha da linguagem C++ e da API DirectX foi fundamental para alcançar um alto desempenho e controle detalhado sobre os recursos do sistema. A capacidade de gerenciar manualmente a memória e otimizar o código foi crucial para garantir uma experiência de jogo fluida e estável.

5.2. Sucesso do Motor Gráfico Personalizado

O motor gráfico desenvolvido mostrou-se eficaz em atender às necessidades do jogo. As funcionalidades implementadas foram integradas de maneira coesa, e o motor proporcionou uma base sólida para a renderização de gráficos, áudio e cálculos de física. As classes e interfaces criadas facilitaram a organização do código e permitiram uma estrutura modular e extensível.

5.3. Considerações Finais

O projeto forneceu uma oportunidade valiosa para aplicar conceitos avançados de programação e desenvolvimento de jogos, além de explorar a aplicação de técnicas de otimização e gerenciamento de recursos. A experiência adquirida durante o desenvolvimento do jogo será benéfica para futuros projetos e desafios na área de desenvolvimento de software e jogos.

Além disso, a consideração da aplicação do paradigma funcional revelou o potencial para criar um código mais modular e fácil de manter. Embora o projeto tenha sido realizado com enfoque na programação orientada a objetos, a integração de conceitos funcionais poderia ter oferecido vantagens adicionais em termos de clareza e eficiência.

O projeto demonstra que, com o planejamento adequado e a escolha correta das ferramentas e técnicas, é possível criar um jogo de alta qualidade que atenda às expectativas e ofereça uma experiência satisfatória para os jogadores.

6. Bibliografia

Para aprofundar os conhecimentos sobre os tópicos abordados neste relatório, seguem algumas referências úteis:

- **C++**:
  - *The C++ Programming Language* de Bjarne Stroustrup - [Link para o livro](https://www.amazon.com/Programming-Language-4th-Bjarne-Stroustrup/dp/032156
  - *Learn C++* - [Tutorial Online](https://www.learncpp.com/)
  - *C++ Reference* - [Documentação Oficial](https://en.cppreference.com/w/)
- **DirectX**:
  - *Introduction to DirectX* - [Tutorial Online](https://docs.microsoft.com/en-us/windows/win32/directx)
  - *DirectX 12 Programming Guide* - [Link para o livro](https://www.amazon.com/Directx-Programming-Guide-Benjamin-2008-07-07/dp/B00YMD4W6Y)
  - *DirectX Graphics Documentation* - [Documentação Oficial](https://learn.microsoft.com/en-us/windows/win32/direct3d12/directx-12-programming-guide)
- **Gerenciamento de Memória**:
  - *Effective C++: 55 Specific Ways to Improve Your Programs and Designs* de Scott Meyers - [Link para o livro](https://www.amazon.com/Effective-Specific-Improve-Your-Programs/dp/0321334876)
  - *Memory Management in C++* - [Tutorial Online](https://www.learncpp.com/cpp-tutorial/9-6-memory-management/)
  - *Dynamic Memory Management* - [Documentação Oficial](https://en.cppreference.com/w/cp
- **Orientação a Objetos**:
  - *Design Patterns: Elements of Reusable Object-Oriented Software* de Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides - [Link para o livro](https://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612)
  - *Object-Oriented Programming in C++* de Robert Lafore - [Link para o livro](https://www.amazon.com/Object-Oriented-Programming-C-4th-Edition/dp/0672323
  - *Object-Oriented Programming Concepts* - [Tutorial Online](https://www.geeksforgeeks.org/oriented-programming-oops-concept/)

- **Programação Funcional**:
  - *Functional Programming in C++: How to Improve Your C++ Programs Using Functional Programming Principles* de Ivan Čukčević - [Link para o livro](https://www.amazon.com/Functional-Programming-Improve-Programs-Principles/dp/1789536712)
  - *Introduction to Functional Programming* - [Tutorial Online](https://www.learncpp.com/cpp-tutorial/functional-programming/)
  - *Functional Programming Concepts* - [Tutorial Online](https://www.geeksforgeeks.org/functi programming/)

6.1. Repositório GitHub

O código-fonte do projeto e todos os materiais relacionados estão disponíveis no repositório GitHub. O acesso ao repositório pode ser feito através do seguinte link:

- *Repositório GitHub* - https://github.com/lfarodrigues/tf-MLP/tree/main