UNIVERSIDADE FEDERAL DE SANTA CATARINA – UFSC

CENTRO TECNOLÓGICO

DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

RELATÓRIO DE DESENVOLVIMENTO DO PROJETO FINAL DE SISTEMAS OPERACIONAIS I

ERIC FERNANDES EVARISTO

VICTOR HUGO BONACINA PERGHER

FLORIANÓPOLIS, 27 DE JUNHO DE 2023

SUMÁRIO

1. Introdução

Este relatório tem como objetivo documentar os detalhes de desenvolvimento do projeto final da disciplina de Sistemas Operacionais 1, realizado no semestre 23.1. O projeto consiste no desenvolvimento do jogo Stellar Warfare, um atirador 2D no qual o jogador assume o controle de uma nave espacial e enfrenta inimigos, acumulando pontos ao destruí-los. O objetivo principal do jogo é derrotar todos os inimigos, desviando das suas balas. À medida que o jogador avança no jogo e derrota os inimigos, a velocidade aumenta progressivamente, proporcionando uma dificuldade crescente. A vitória é alcançada somente quando todos os inimigos da velocidade máxima são derrotados.

Para possibilitar um ambiente de jogo dinâmico e responsivo, foi adotada a abordagem de desenvolvimento concorrente. Nesse sentido, a biblioteca de implementação de threads, desenvolvida ao longo da disciplina de Sistemas Operacionais 1, foi utilizada. Essa biblioteca forneceu suporte para a execução simultânea de diferentes partes do jogo, permitindo a criação de uma experiência de jogo mais fluída e interativa.

Ao longo deste relatório, exploraremos os principais aspectos do desenvolvimento do jogo Stellar Warfare, desde a análise de requisitos até a implementação detalhada das funcionalidades. Também discutiremos a arquitetura do sistema, destacando o papel da biblioteca de threads na criação de um ambiente de jogo concorrente.

2. Análise de Requisitos

2.1. Requisitos funcionais

Requisito	Descrição
Movimentação do jogador	O jogador deve ser capaz de controlar a nave utilizando as teclas de direção do teclado.
Movimentação dos inimigos	As naves inimigas devem ser capazes de no mínimo dois padrões de movimento diferentes.
Disparos	O jogador deve ser capaz de atirar nos inimigos com a tecla espaço do teclado, assim como os inimigos devem atirar no jogador.
Inimigos	O jogo deve gerar inimigos para atacar o jogador.
Colisões	O jogo deve ser capaz de lidar com colisões do tipo nave com nave, nave com bala, e bala com bala.
Pontuação	O jogador deve ganhar 100 pontos toda vez que abate um inimigo.
Vida	O jogador deve ter três vidas, e caso seja atingido por uma bala inimiga ou colida com uma nave inimiga, deve perder uma delas.
Velocidade	Ao derrotar quatro inimigos, as naves inimigas devem aumentar a sua velocidade, de modo que existem três níveis de velocidade diferentes.
Vitória	O jogador será vitorioso quando ele derrota todos os inimigos na velocidade máxima.
Derrota	O jogador será derrotado quando a sua vida chegar a zero.

2.2. Requisitos não funcionais

Requisito	Descrição
Linguagem	O jogo deve ser implementado usando a linguagem C++.
Orientação	O jogo deverá ser desenvolvido usando o paradigma de orientação a objetos do C++.
Concorrência	O jogo deve ser concorrente e possuir um número mínimo de sete threads.
Desempenho	O jogo deve ser responsivo e executar de forma fluída.
Interface de usuário	A interface de usuário deve ser intuitiva e de fácil uso.
Estabilidade	O jogo deve ser estável e livre de erros, evitando travamentos e falhas inesperadas.
Memória	Não deve existir vazamento de memória.
Gráficos	Os gráficos deverão ser desenvolvidos usando a biblioteca SFML e devem ser de qualidade satisfatória.

3. Ferramentas e tecnologias

Nesta seção, as principais ferramentas e tecnologias utilizadas no desenvolvimento do jogo Stellar Warfare serão apresentadas a seguir:

3.1. Linguagem de programação

C++: A linguagem de programação usada para o desenvolvimento do jogo. O C++ é conhecido por sua eficiência, suporte a orientação a objetos e recursos avançados, tornando-o uma ótima opção para o desenvolvimento de jogos.

3.2. Bibliotecas

Biblioteca de threads: A biblioteca desenvolvida ao longo do semestre é utilizada para suporte à programação concorrente por meio de threads. A biblioteca é capaz de fornecer uma interface para criação, sincronização e gerenciamento de threads em nível de usuário, o que possibilita a criação de um ambiente de jogo concorrente.

Biblioteca SFML (Simple and Fast Multimedia Library): Utilizada para a implementação de recursos multimídia, como gráficos e entrada de usuário. A biblioteca oferece uma abstração de baixo nível para gráficos e manipulação de janelas, permitindo a criação de uma interface de usuário interativa e responsiva.

3.3. Ambiente de desenvolvimento integrado

Visual Studio Code: Utilizado como ambiente de desenvolvimento integrado (IDE) para escrever, depurar e testar o código-fonte do jogo. O Visual Studio Code foi selecionado por sua facilidade de uso, suporte a extensões e recursos avançados de edição e depuração.

3.4. Controle de versão

Git: Utilizado como sistema de controle de versão para o gerenciamento do código-fonte do jogo. O Git permite o controle eficiente das alterações no código, facilitando o trabalho em equipe e a rastreabilidade das modificações.

3.5. Plataforma de compilação

Compilador G++ (parte do GNU Compiler Collection): Utilizado como compilador para a linguagem C++. O GCC é uma ferramenta amplamente utilizada e suportada que fornece um ambiente de compilação robusto e eficiente.

3.6. Plataforma de distribuição

Código-fonte: O jogo Stellar Warfare é distribuído como código-fonte, permitindo personalização e modificação pelos jogadores. A compilação é realizada através do comando *make* no terminal, utilizando o arquivo *Makefile* presente no diretório do projeto. O *Makefile* contém as regras de compilação e vinculação necessárias para criar o executável do jogo.

3.7. Outras ferramentas

PlantUML: A ferramenta PlantUML foi utilizada no projeto para criar diagramas UML, como diagramas de classes, sequência e casos de uso. Com a sintaxe baseada em texto, o PlantUML permite a representação visual dos elementos do projeto, facilitando a compreensão da arquitetura e do design do jogo.

4. Design e Arquitetura

Nesta seção, apresentaremos a visão geral do design e arquitetura do Stellar Warfare, descrevendo as decisões de planejamento, as principais classes e gerenciadores do projeto.

4.1. Planejamento

Inicialmente, foram feitas algumas decisões quanto a organização. O jogo funcionaria com gerenciadores como a Scene, Renderer, Input e UserInterface. O jogador, inimigos e balas seriam entidades que iriam interagir entre si com o controle de colisões e eventos realizados na classe Scene. Da mesma forma, os elementos da interface seriam gerenciados pela UserInterface. Os eventos de input seriam enviados pelo Input e seriam repassados para a Scene e consequentemente para o jogador. O Renderer ficaria responsável por chamar os métodos de renderização de cada gerenciador, neste caso, Scene e UserInterface. Os gerenciadores chamam então os métodos de renderização das suas entidades gerenciadas.

Foi feito o uso de herança com todas as classes que possuem threads. Uma classe ThreadContainer define os comportamentos de uma classe com thread. Também foi usada herança com o jogador, inimigos e balas. O jogador e os inimigos são especializações de naves (classe Spaceship) e as naves, juntamente com as balas, são especializações de entidades (classe Entity). Entity é uma classe abstrata que nunca é instanciada na prática, sendo assim a sua utilização depende da especialização feita em outra classe que define alguma entidade do jogo. Por fim, a herança foi usada também nas classes Image e Text, ambas herdam de Widget, que por sua vez define um elemento da interface.

De modo geral o objetivo desta arquitetura é promover uma comunicação simples entre as múltiplas camadas de funcionamento do jogo, como input, renderização e lógica de gameplay, e ao mesmo tempo permitir a flexibilidade na modificação do comportamento das entidades.

No total. o jogo possui 8 threads, sendo que a Scene possui uma thread adicional que não fazia parte dos requisitos originais do projeto.

4.2. Classes principais

- Scene: Simula e gerencia a lógica de gameplay;
- Game: Faz a ligação e contém todos os gerenciadores;
- Input: Detecta eventos de entrada e os comunica para os outros gerenciadores;
- Renderer: Renderiza a Scene e UserInterface.

4.3. Diagrama de classes

O diagrama de classes é uma poderosa ferramenta capaz de auxiliar na visualização da estrutura do código e as interações entre as classes, fornecendo uma representação visual clara da arquitetura jogo. Neste trabalho, optamos por não inclui-lo nesse documento devido a perda de qualidade ao alterar seu tamanho e formato, no entanto o documento completo pode ser visualizado na pasta *docs* do projeto.

5. Implementação

5.1. Gerenciadores

5.1.1. Scene

A classe Scene gerencia o comportamento do gameplay, como por exemplo: a inicialização e finalização do gameplay, criação e destruição do jogador, criação e destruição dos inimigos, criação e destruição das balas e colisões. Os movimentos e tiro são processados com base nas “intenções” registradas pelas entidades. Cada entidade registra a sua intenção de movimento ou tiro quando ela pode se movimentar ou atirar, existe um controle de tempo que impede as entidades de registrarem mais movimentos do que o possível para as suas respectivas velocidades ou atirarem mais do que as suas cadências de tiro.

As colisões são processadas para todas as entidades que estão se movimentando na iteração do processamento. Para processar as naves são utilizados os semáforos, trancando-as durante o processamento, assim pode-se garantir o sincronismo entre as threads.

5.1.2. Input

A classe é simples e realiza apenas a detecção da entrada. A cada tecla reconhecida pressionada, um evento é disparado e comunicado para os outros gerenciadores. Os gerenciadores possuem semáforos para sincronizar o tratamento dos eventos e a transição para os novos estados.

5.1.3. Renderer

O Renderer apenas chama os métodos de renderização da Scene e da UserInterface. Além disso, o renderer também apaga a tela no início da renderização e exibe o conteúdo renderizado no final.

5.1.4. UserInterface

A classe UserInterface, diferentemente dos outros gerenciadores, não possui uma thread própria. O motivo desta implementação é o de que o comportamento da interface é apenas reativo ao gameplay e a UserInterface não iria realizar nenhuma tarefa intensiva durante o fluxo do jogo.

5.2. Algoritmos dos inimigos e progressão

Os inimigos nascem em um dos cantos da tela conforme o tempo de 2 segundos requisitado no projeto. Quando o imigo nasce, um algoritmo é atribuído aleatóriamente à ele. Existem 5 algoritmos:

- *random*: O inimigo realiza movimentos totalmente aleatórios e tem 1,5625% de chance de atirar a cada iteração do loop de comportamento.
- *Spinner*: O inimigo irá girar no próprio eixo e tem 50% de chance de atirar a cada iteração do loop de comportamento.
- *Walker*: A nave inimiga anda em linha reta até econtrar um obstáculo, quando o encontra, ela desvia para alguma outra direção e continua em linha reta. A chance da nave atirar neste algoritmo é de 0,39% a cada iteraçaõ do loop de comportamento.
- *Hunter*: Neste algoritmo a nave irá perseguir o jogador até colidir com ele. A chance de atirar é de 0,195%, a mais baixa de todos os algoritmos.
- *Shooter*: O inimigo irá se movimentar horizontalmente até se alinhar com o jogador verticalmente e então irá atirar. Neste algoritmo a nave irá lentamente se apróximar do jogador a cada vez que ela se realinhar verticalmente. A nave sempre atira quando está alinhada com o jogador verticalmente.

A progressão do jogo segue o requisito de 3 níveis com velocidades diferentes. As velocidades para cada nível são, respectivamente: 4, 8 e 16 unidades por segundo. As unidades neste caso se referenciam a uma célula no mundo de jogo.

5.3. Estruturas de dados

Para implementar o armazenamento dos inimigos, balas e tempos de criação de inimigos, foi utilizada uma classe especial DynamicArray. A DynamicArray recebe o argumento de tamanho mínimo na sua criação, esse tamanho é alocado inicialmente. Conforme mais elementos são adicionados na array, ela aloca mais memória. Um detalhe importante é que a DynamicArray não desaloca os espaços alocados durante o seu ciclo de vida.

Esta classe foi criada pois as estruturas padrões como std::vector possuiam uma remoção ineficiente, e ao mesmo tempo a utilização de listas

encadeadas não permitia o acesso aleatório eficiente. Então foi implementada uma solução que oferecesse escalabilidade na alocação, remoção com baixo custo e acesso aleatório eficiente.

A maior desvantagem seria o uso crescente da memória, porém como o número de entidades no jogo é bem limitado, isso não se torna um problema na prática para este caso.

6. Interface de Usuário (UI)

Nesta seção, descreveremos a interface de usuário do jogo, incluindo seu layout, elementos visuais e interações principais:

6.1. Layout e organização

- Tela do inicio: Apresenta a tela inicial do jogo, com o estado não inicializado e com a opção de iniciar um novo jogo utilizando a tecla P.

- Tela de jogo: Exibe a jogabilidade principal, com o jogador, inimigos, balas e informações relevantes como score, hp e level.

- Tela de pausa: Quando o jogador pressiona a tecla P durante o jogo, ele pode ser pausado. O mapa do jogo fica congelado em posição estática e o estado é atualizado no canto superior direito.

- Tela de fim de jogo: Todas as entidades são deletadas e ficamos com um mapa vazio, assim como o estado de gameover no canto superior direito.

6.2. Elementos da interface

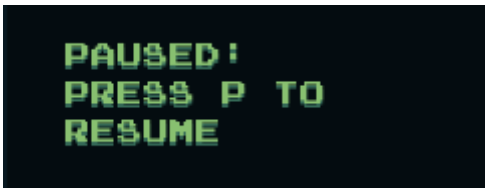
- Nave do jogador: O jogador é representando graficamente pela seguinte sprite:

- Naves dos inimigos: Os inimigos são representados graficamente pela seguinte sprite:

- Balas: As balas são representadas graficamente por uma única célula da grid do jogo:

- Título: O título fica sempre presente no canto superior direito do jogo:

- Estado do jogo: O rótulo do estado do jogo fica sempre presente em baixo do título, e serve para informar ao jogador as possíveis ações que ele pode tomar:



- Score: Informa ao jogador a pontuação acumulada nessa instância do jogo:

- Vida: Informa ao jogador a quantidade de vidas que ele ainda possui:

- Level (Velocidade): Informa ao jogador o level do jogo que ele atingiu:

6.3. Interações e controle

- Controles de movimento: O jogador utiliza as teclas direcionais (setas) para mover a nave para cima, baixo, direita e esquerda.

- Disparo de balas: O jogador utiliza a tecla *espaço* para atirar balas, com o objetivo de abater os inimigos.

- Colisões: O jogo detecta colisões entre entidades (naves e balas), e aplica as regras de dano ou pontuação correspondentes.

7. Compilação e distribuição

Nesta seção, descreveremos o processo de compilação e distribuição do Stellar Warfare, incluindo as etapas para a geração do executável e da disponibilização do jogo.

7.1. Compilação

- Requisitos de compilação:

  ➔ Compilador G++ (talvez adicionar a versão);
  ➔ Make;
  ➔ SFML;
  ➔ libpng;

- Instalação de dependências para plataformas de distribuição Debian:

  ➔ G++ e MAKE : *sudo apt install build-essential*
  ➔ SFML : *sudo apt-get install libsfml-dev*
  ➔ libpng : *sudo apt-get install libpng-dev*

- Comando de compilação:

Para compilar o projeto basta usar o comando *make* no diretório raiz do projeto (que possui o arquivo *makefile*), e se tudo der certo, um executável será criado na pasta *bin*.

Para rodar o executável pode-se usar o comando "*./bin/Spaceship game"*.

7.2. Distribuição

O processo de empacotamento do jogo ocorre através do uso de um Makefile. O Makefile define as regras de compilação e construção do jogo, onde são configuradas variáveis como o compilador, bibliotecas e opções de compilação. Ao executar o Makefile, os arquivos de código-fonte são compilados em arquivos de objeto e, em seguida, vinculados para formar o executável do jogo. Esse executável é armazenado no diretório bin. A regra "clean" permite limpar os arquivos de objeto e remover o executável gerado, facilitando a limpeza do projeto.

O jogo suporta apenas a plataforma Linux atualmente, e pode ser encontrado para distribuição no repositório:

https://github.com/ErFer7/INE5412-OS

8. Conclusão

O desenvolvimento do projeto foi concluído dentro do prazo estabelecido, atendendo a todos os requisitos funcionais e não funcionais definidos. Um aspecto desafiador, no entanto, foi a implementação da biblioteca de threads em nível de usuário, devido à natureza complexa das próprias threads. Embora o volume de código não tenha sido tão grande, o código em si apresentou uma complexidade significativa e exigiu um grande esforço para compreensão. No entanto, a estratégia adotada durante a disciplina, que consistiu em desenvolver pequenos fragmentos de código ao longo do semestre e receber feedback contínuo, foi fundamental para facilitar o processo de desenvolvimento.

Durante essa jornada, adquirimos um conhecimento mais aprofundado sobre o funcionamento dos sistemas operacionais, especialmente no que diz respeito à concorrência. Também ampliamos nosso entendimento sobre arquiteturas concorrentes e aprimoramos nossas habilidades específicas em C++ e engenharia de software. Além disso, a integração da biblioteca SFML nos permitiu explorar conceitos relacionados à construção de interfaces de usuário interativas e gráficas.

Embora tenhamos alcançado os objetivos estabelecidos, reconhecemos que existem oportunidades de melhoria no jogo. Por exemplo, é possível aprimorar a fluidez e a responsividade dos controles para proporcionar uma experiência de jogo mais satisfatória. Além disso, explorar algoritmos mais robustos para o padrão de movimentação dos inimigos poderia adicionar um nível extra de desafio e diversidade ao gameplay. Também consideramos a possibilidade de implementar novas funcionalidades, como a introdução de power-ups, para tornar a experiência ainda mais dinâmica.

No geral, o desenvolvimento deste projeto nos proporcionou valiosas oportunidades de aprendizado e crescimento, tanto em termos técnicos quanto em habilidades de colaboração e resolução de problemas.