TRABALHO PARA A DISCIPLINA DE TÉCNICAS DE PROGRAMAÇÃO DO CURSO DE SISTEMAS DE INFORMAÇÃO DA UTFPR: KNIGHT'S QUEST

Daniel Zaki Sommer, Enzo Westphal Tacla
danielsommer@alunos.utfpr.edu.br, enzotacla@alunos.utfpr.edu.br

Disciplina: **Técnicas de Programação – ICSE20 / S73** – Prof. Dr. Jean M. Simão
Departamento Acadêmico de Informática – DAINF - Campus de Curitiba
Curso Bacharelado em Sistemas de Informação
Universidade Tecnológica Federal do Paraná - UTFPR
Avenida Sete de Setembro, 3165 - Curitiba/PR, Brasil - CEP 80230-901

Resumo – Neste projeto de Técnicas de Programação, o grupo se dedicou ao desenvolvimento do jogo em C++, batizado de "Knight's Quest", visando aprimorar suas habilidades em engenharia de software, especialmente na programação orientada a objetos. O jogo apresenta duas fases com diferentes níveis de dificuldade, cada uma contendo 2 tipos de inimigos e 2 tipos de obstáculos na primeira fase, e 3 tipos de inimigos e 3 tipos de obstáculos na segunda. Para garantir uma base sólida durante o desenvolvimento, o grupo iniciou o processo considerando os requisitos estabelecidos e criou uma modelagem utilizando Diagrama de Classes na Linguagem de Modelagem Unificada (UML). Essa etapa foi crucial para visualizar e organizar a estrutura do jogo antes de iniciar a implementação. Durante a fase de implementação em C++, foram incorporados conceitos básicos de Orientação a Objetos, como Classes, Objetos e Relacionamentos, além de explorar aspectos mais avançados, como o uso de Classes Abstratas e Polimorfismo. Esses elementos não apenas tornaram o código mais modular e reutilizável, mas também facilitaram a manutenção e expansão do jogo. Além disso, a escolha da biblioteca Simple and Fast Multimedia Library (SFML) para a implementação do jogo adicionou uma camada extra de complexidade, mas proporcionou benefícios significativos em termos de facilidade de uso e desempenho. Em resumo, o projeto não apenas proporcionou um ambiente prático para a aplicação de conceitos de engenharia de software, mas também permitiu que o grupo explorasse e implementasse de forma eficaz os princípios da programação orientada a objetos em um ambiente de desenvolvimento de jogos, enriquecendo assim suas habilidades técnicas e criativas.

Palavras-chave ou Expressões-chave: Técnicas de Programação em C++, Programação Orientada a Objetos, Diagrama de Classes e UML, Simple and Fast Multimedia Library (SFML).

1. INTRODUÇÃO

Este trabalho, inserido na disciplina de Técnicas de Programação, tem como proposta a compreensão e implementação de um jogo de plataforma em C++, aplicando os princípios da orientação a objetos, com foco em coesão e desacoplamento. O objetivo é demonstrar a capacidade de análise, modelagem e implementação de software, aplicando os conceitos aprendidos em sala de aula.

O jogo escolhido permite a ampla utilização dos recursos da linguagem C++ e está alinhado com os requisitos propostos. O método adotado segue o ciclo simplificado de Engenharia de Software, envolvendo a compreensão dos requisitos, modelagem por meio de diagramas de classes em UML, implementação em C++ orientado a objetos e realização de testes para garantir a funcionalidade do software. Essa abordagem estruturada assegura a qualidade do produto final.

O objetivo principal deste trabalho é criar e apresentar um jogo de plataforma em C++, utilizando os conceitos de orientação a objetos, ampliando a aplicação prática dos conhecimentos adquiridos em sala de aula e proporcionando uma experiência significativa no desenvolvimento de software.

2.EXPLICAÇÃO DO JOGO

Este capítulo fornece uma visão abrangente do jogo "Knight's Quest", destacando sua dinâmica, mecânicas e elementos-chave. Não abordamos a modelagem ou implementação, focando na experiência do usuário. "Knight's Quest" é um jogo de plataforma 2D estratégico. Os jogadores exploram fases, enfrentam desafios, inimigos únicos e obstáculos traiçoeiros. A interface oferece opções para escolher fases, verificar rankings e realizar ações cruciais para progredir. O próximo capítulo detalha a implementação dos requisitos funcionais, desde o menu até a gestão de colisões e efeitos de 'gravidade', criando uma experiência envolvente. O jogo apresenta inimigos como o "Cogumelo" (envenenador), "Olho Voador" (afeta mobilidade) e o temível "Chefão" com estágios de raiva. Obstáculos como "Espinho", "Slime" e "Serra" exigem estratégias diferentes. Plataformas, paredes e cenários intrincados aumentam a complexidade, demandando decisões rápidas e precisas. Os jogadores podem salvar e pausar o jogo. A pontuação está ligada à neutralização de inimigos, com um sistema de ranking registrando o desempenho dos jogadores.

3. DESENVOLVIMENTO DO JOGO NA VERSÃO ORIENTADA A OBJETOS

O presente capítulo apresenta uma análise detalhada dos requisitos funcionais do jogo, fornecendo *insights* sobre a implementação de cada elemento essencial para a experiência do usuário. A Tabela 1 destaca os requisitos identificados, juntamente com exemplos de situações correspondentes, sua implementação e a avaliação de seu cumprimento. Ao longo deste capítulo, examinaremos a concretização de requisitos fundamentais, desde a apresentação gráfica do menu de opções até a gestão eficaz de colisões entre jogadores, inimigos e obstáculos. Destaca-se a ênfase na diversidade, evidenciada pela presença de distintos inimigos e obstáculos, cada um contribuindo para a complexidade e desafio do jogo. A implementação de mecânicas específicas, como a gestão de colisões e a aplicação de 'gravidade', será discutida em detalhes, proporcionando uma compreensão abrangente do funcionamento interno do jogo.

Além disso, são apresentadas observações sobre a realização parcial de determinados requisitos, delineando áreas específicas que requerem aprimoramento. Destaca-se o desenvolvimento dos inimigos 'Cogumelo', 'Olho Voador' e 'Chefão', cada um com características distintas que acrescentam camadas de estratégia ao jogo. Da mesma forma, os obstáculos, representados por 'Espinho', 'Slime' e 'Serra', introduzem desafios únicos, contribuindo para a riqueza da experiência de jogo.

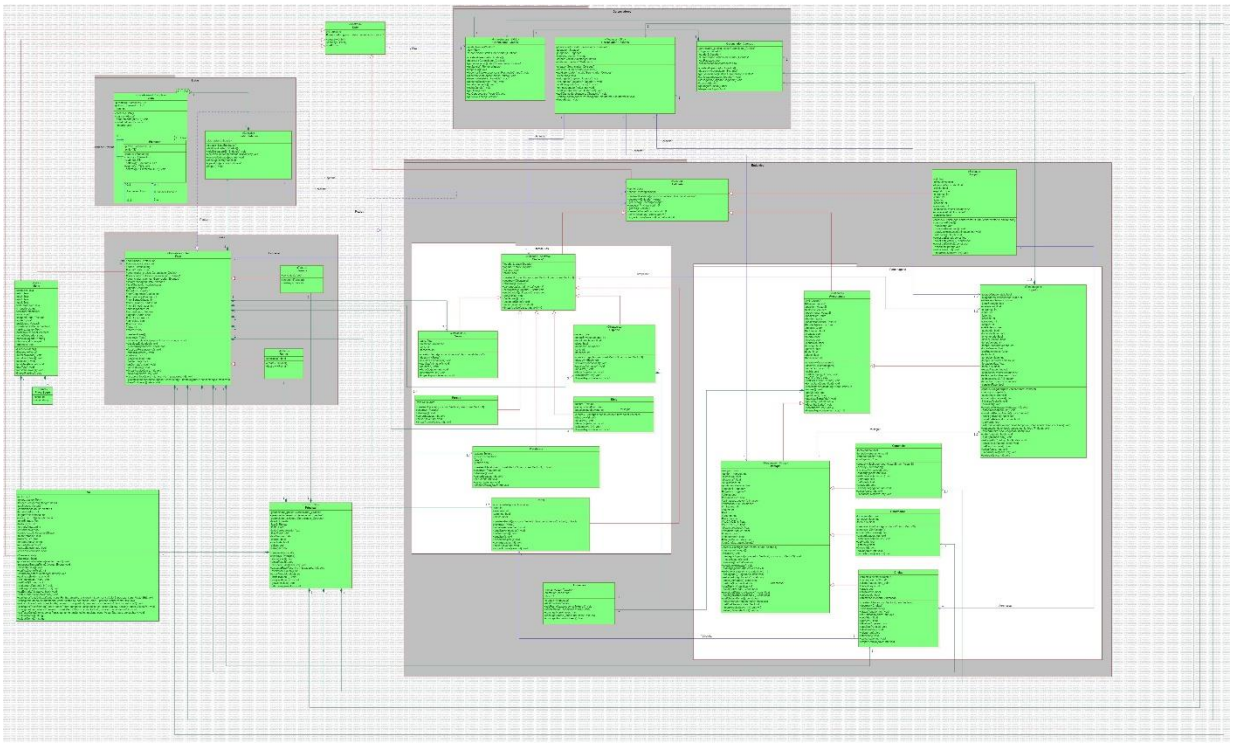

Tabela 1. Lista de Requisitos do Jogo e exemplos de Situações.

N.	**Requisitos Funcionais**	**Situação**	**Implementação**
1	Apresentar graficamente menu de opções aos usuários do Jogo, no qual pode se escolher fases, ver colocação (*ranking*) de jogadores e demais opções pertinentes (previstas nos demais requisitos).	Requisito previsto inicialmente e realizado.	Requisito cumprido com suporte da SFML via Classe Tela e Classe Menu (que funciona como um gerenciador dessas Telas).
2	Permitir um ou dois jogadores com representação gráfica aos usuários do Jogo, sendo que no último caso seria para que os dois joguem de maneira concomitante.	Requisito previsto inicialmente e realizado.	Requisito cumprido inclusive via classe Jogador cujos objetos são agregados em jogo, podendo ter um ou dois por fase criada, ficando a critério do usuário no Menu.
3	Disponibilizar ao menos duas fases que podem ser jogadas sequencialmente ou selecionadas, via menu, nas quais jogadores tentam neutralizar inimigos por meio de algum artifício e vice-versa.	Requisito previsto inicialmente e realizado.	Requisito realizado completamente porque as Classes Floresta e Ruínas (ambas derivadas da classe Fase e, portanto, fases distintas) podem ser executadas via menu ou sequencialmente, quando a primeira é finalizada. Essa lógica de conectá-las é feita via Classe Principal.

4	Ter pelo menos três tipos distintos de inimigos, cada qual com sua representação gráfica, sendo que ao menos um dos inimigos deve ser capaz de lançar projéteis contra o(s) jogador(es) e um dos inimigos deve ser um 'Chefão'.	Requisito previsto inicialmente e realizado.	Requisito realizado completamente, porque as Classes derivadas de Inimigo: Cogumelo, Olho Voador e Chefão, representam diferentes inimigos com diferentes comportamentos, atributos e funcionalidades.
5	Ter a cada fase ao menos dois tipos de inimigos com número aleatório de instâncias, podendo ser várias instâncias (definindo um máximo) e sendo pelo menos 3 instâncias por tipo.	Requisito previsto inicialmente e realizado.	Requisito atendido, pois os tipos de inimigos e o número máximo permitido para cada tipo estão definidos nos arquivos .txt de cada fase. A criação aleatória de entidades é realizada por meio da função "bool Aleatorizar(char character)", que tem uma probabilidade de 1/3 de retornar "false". Isso impede a criação de uma entidade que já tenha excedido um número mínimo pré-definido (o mínimo é de 3 instâncias para cada tipo distinto de inimigo).
6	Ter três tipos de obstáculos, cada qual com sua representação gráfica, sendo que ao menos um causa dano em jogador se colidirem.	Requisito previsto inicialmente e realizado.	Requisito realizado completamente, porque as Classes derivadas de Obstáculo: Serra, Espinho e Slime representam diferentes obstáculos com diferentes comportamentos, atributos e funcionalidades.
7	Ter em cada fase ao menos dois tipos de obstáculos com número aleatório (definindo um máximo) de instâncias (*i.e.*, objetos), sendo pelo menos 3 instâncias por tipo.	Requisito previsto inicialmente e realizado.	Requisito atendido, pois os tipos de obstáculos e o número máximo permitido para cada tipo estão definidos nos arquivos .txt de cada fase. A criação aleatória de entidades é realizada por meio da função "bool Aleatorizar(char character)", que tem uma probabilidade de 1/3 de retornar "false". Isso impede a criação de uma entidade que já tenha excedido um número mínimo pré-definido (o mínimo é de 3 instâncias para cada tipo distinto de obstáculo).
8	Ter em cada fase um cenário de jogo constituído por obstáculos, sendo que parte deles seriam plataformas ou similares, sobre as quais pode haver inimigos e podem subir jogadores.	Requisito previsto inicialmente e realizado.	O requisito é satisfeito pelas classes "Plataforma" e "Parede" dentro do namespace "Obstáculos", que, em conjunto com o "Gerenciador de Colisões", desempenham um papel crucial na interação com as fases e na dinâmica do movimento tanto dos jogadores quanto dos inimigos presentes no jogo.

9	Gerenciar colisões entre jogador para com inimigos e seus projéteis, bem como entre jogador para com obstáculos. Ainda, todos eles devem sofrer o efeito de alguma 'gravidade' no âmbito deste jogo de plataforma vertical e 2D.	Requisito previsto inicialmente e realizado.	O requisito é atendido pela classe "Personagem" ao aplicar a gravidade. Por sua vez, a classe "Gerenciador_Colisoes" atua como um componente essencial ao oferecer mecanismos para lidar com colisões entre diferentes entidades no jogo. Ele facilita a interação entre jogadores, inimigos e obstáculos.
10	Permitir: (1) salvar nome do usuário, manter/salvar pontuação do jogador (incrementada via neutralização de inimigos) controlado pelo usuário e gerar lista de pontuação (*ranking*). **E** (2) Pausar e **Salvar/Recuperar** Jogada.	Requisito previsto inicialmente, mas realizado apenas PARCIALMENTE.	Tal requisito foi realizado PARCIALMENTE pois o sistema possui um mecanismo de Salvar as informações das Entidades em até 3 arquivos por Classe, porém, não é possível recuperar esses dados durante a execução do Jogo.
Total de requisitos funcionais apropriadamente realizados. *(Cada tópico vale 10%, sendo que para ser contabilizado deve estar realizado efetivamente e não parcialmente)*			**90%** (noventa por cento).

4. EXPLICAÇÃO DAS CLASSES E SUAS RELAÇÕES

Este capítulo explora a arquitetura de classes do projeto, detalhando as classes principais e suas relações. Cada classe é apresentada com sua função específica, destacando a hierarquia e interconexões entre elas.

- Entidade: Classe abstrata que serve como base para diversas entidades no jogo, com métodos virtuais puros estabelecendo um padrão para entidades específicas.

- Personagem: Derivada de Entidade, encapsula as características gerais de um personagem no jogo. Introduz elementos específicos como velocidade, direção e informações sobre a saúde do personagem.

- Jogador: Representa o personagem controlado pelo jogador. Derivada de Personagem, herda características gerais de movimentação e interação, mas também introduz funcionalidades específicas do jogador, como ataques, pulos e movimentos.

- Inimigo: Estende a classe Personagem, adicionando comportamentos específicos para personagens hostis, como perseguição, movimentação aleatória e lógica de ataque.

- Chefao: Derivada de Inimigo, incorpora atributos e métodos que definem características exclusivas de chefões no jogo. Representa inimigos de nível superior, com ataques especiais, estágios de raiva e desafios mais complexos.

- Obstaculo: Classe abstrata fundamental para representar elementos que podem afetar negativamente os personagens do jogo. Herda de Entidade e contém atributos e métodos comuns a todos os obstáculos, como a capacidade de causar dano aos jogadores e a habilidade de colidir com outros elementos do jogo.

- Plataforma: Especialização de Obstáculo, modela elementos que servem como superfícies transitáveis pelos jogadores. Herda de Obstaculo, influenciando a jogabilidade.

- Fase: Representa uma fase do jogo, herdando de Ente e encapsulando lógica específica para uma fase. Utiliza diversos gerenciadores para coordenar diferentes facetas da jogabilidade, mantendo listas de personagens e obstáculos. Possui métodos para atualização de personagens, controle de câmera, verificação de conclusão da fase, entre outros.

- Menu: Representa o menu principal do jogo, herdando de Ente e contendo instâncias de diferentes telas. Utiliza uma pilha para controlar a transição entre telas, além de lidar com informações sobre o número de jogadores, coordenadas do mouse, texturas e sprites. Lida com a exibição de rankings, entrada de nomes de jogadores e interações com botões.

- Principal: Classe coordenadora principal do jogo, gerencia fases, gráficos, eventos e colisões. Atua como orquestradora, garantindo o fluxo adequado durante a execução. Mantém instâncias das fases (Floresta e Ruínas) e lida com a alocação e recuperação dessas fases, bem como a execução delas. Controla o estado do jogo, indicando se o jogador foi derrotado, se a fase foi concluída e se é necessário salvar ou carregar o jogo.

2. TABELA DE CONCEITOS UTILIZADOS E NÃO UTILIZADOS

Neste capítulo, serão apresentadas as análises e reflexões sobre os conceitos fundamentais aplicados ao longo do desenvolvimento do projeto. A estrutura do código foi meticulosamente organizada, respeitando as boas práticas de programação e abordando desde conceitos elementares até tópicos mais avançados da programação orientada a objetos.

Tabela 2. Lista de Conceitos Utilizados e Não Utilizados no Trabalho.

N.	Conceitos	Uso	Onde / O quê / Justificativa em uma linha
1	**Elementares:**		
1.1	- Classes, objetos. & - Atributos (privados), variáveis e constantes. & - Métodos (com e sem retorno).	Sim	Todos .hpp e .cpp, como nas classes Jogador e Fase, nos *namespaces* Entidades e Fases, respectivamente.
1.2	- Métodos (com retorno *const* e parâmetro *const*). & - Construtores (sem/com parâmetros) e destrutores	Sim	Na maioria dos .hpp e .cpp, como na classe Jogador no *namespace* Entidades.
1.3	- Classe Principal.	Sim	Main.cpp & Principal.hpp/.cpp
1.4	- Divisão em .h e .cpp.	Sim	Em todo o desenvolvimento, cada classe implementada foi separada em .hpp e .cpp.
2	**Relações de:**		
2.1	- Associação direcional. & - Associação bidirecional.	Sim	Em vários dos .hpp e .cpp, como nas classes Inimigo e Jogador.
2.2	- Agregação via associação. & - Agregação propriamente dita.	Sim	Em vários dos .hpp e .cpp, como nas classes Fase com as classes Obstáculo e Entidades.
2.3	- Herança elementar. & - Herança em diversos níveis.	Sim	Em alguns dos .hpp e .cpp, como nas classes nos *namespace Entidades*.
2.4	- Herança múltipla.	Sim	Precisamente nos .hpp e .cpp, das classes OlhoVoador, Cogumelo e Chefao.

3	**Ponteiros, generalizações e exceções**		
3.1	- Operador *this* para fins de relacionamento bidirecional.	Sim	Como nos 3 Gerenciadores (Gráfico, de Colisões e de Eventos).
3.2	- Alocação de memória (*new* & *delete*).	Sim	Como na classe Menu.
3.3	- Gabaritos/*Templates* criada/adaptados pelos autores (*e.g.*, Listas Encadeadas via *Templates*).	Sim	Como nas Classes Lista e ListaEntidade, do *namespace* Lista.
3.4	- Uso de Tratamento de Exceções (*try catch*).	Não	
4	**Sobrecarga de:**		
4.1	- Construtoras e Métodos.	Não	
4.2	- Operadores (2 tipos de operadores pelo menos – Quais?).	Não	
---	**Persistência de Objetos (via arquivo de texto ou binário)**		
4.3	**- Persistência de Objetos.**	Sim	Parcialmente. Os objetos têm a capacidade de armazenar seus atributos em arquivos de salvamento, porém não os recuperam.
4.4	**- Persistência de Relacionamento de Objetos.**	Não	
5	**Virtualidade:**		
5.1	- Métodos Virtuais Usuais.	Sim	Como na Classe Entidade.
5.2	- Polimorfismo.	Sim	Como na Classe Entidade e suas derivadas.
5.3	- Métodos Virtuais Puros / Classes Abstratas.	Sim	Como na Classe Ente e Entidade.
5.4	- Coesão/Desacoplamento efetiva e intensa com o apoio de padrões de projeto.	Sim	Como nas Classes do *namespace* Gerenciadores, que foi utilizado o padrão Singleton.

6	**Organizadores e Estáticos**		
6.1	- Espaço de Nomes (*Namespace*) criada pelos autores.	Sim	Foram utilizados os *Namespaces* pré-determinados pelo modelo do professor.
6.2	- Classes aninhadas (*Nested*) criada pelos autores.	Não	
6.3	- Atributos estáticos e métodos estáticos.	Sim	Como na Classe Fase e na Classe Jogador.
6.4	- Uso extensivo de constante (*const*) parâmetro, retorno, método...	Sim	Como na Classe Fase e na Classe Jogador.
7	Standard Template Library *(STL)* e String OO		
7.1	- A classe Pré-definida *String* ou equivalente. **&** - *Vector* e/ou *List* da *STL* (p/ objetos ou ponteiros de objetos de classes definidos pelos autores)	Sim	Como nas classes Personagem ou na Classe Chefão, ambas do *namespace* Entidades::Personagens.
7.2	- Pilha, Fila, Bifila, Fila de Prioridade, Conjunto, Multi-Conjunto, Mapa **OU** Multi-Mapa.	Sim	Como na Classe ListaEntidade, na Classe Fase e na Classe Gerenciador_Colisoes.
---	**Programação concorrente**		
7.3	- *Threads* (Linhas de Execução) no âmbito da Orientação a Objetos, utilizando Posix, C-Run-Time **OU** Win32API ou afins.	Não	
7.4	- *Threads* (Linhas de Execução) no âmbito da Orientação a Objetos com uso de Mutex, Semáforos, **OU** Troca de mensagens.	Não	
8	**Biblioteca Gráfica / Visual**		

8.1	- Funcionalidades Elementares. **&** - Funcionalidades Avançadas como: ● tratamento de colisões ● duplo *buffer*	Sim	O Gerenciador_Colisoes é responsável por verificar e tratar colisões entre os elementos do jogo, como jogadores, inimigos e obstáculos. Esse gerenciador assegura que as interações entre personagens, inimigos e obstáculos no jogo ocorram de maneira apropriada, incluindo danos causados por colisões e tratamento adequado de movimentos sobre plataformas.
8.2	- Programação orientada e evento efetiva (com gerenciador apropriado de eventos inclusive) em algum ambiente gráfico. **OU** - *RAD – Rapid Application Development* (Objetos gráficos como formulários, botões etc).	Não	
---	**Interdisciplinaridades via utilização de Conceitos de <u>Matemática Contínua e/ou Física</u>.**		
8.3	- Ensino Médio Efetivamente.	*Sim*	Foi utilizado o conceito de queda livre para aplicar gravidade nas Entidades e de lançamento vertical para o pulo dos jogadores.
8.4	- Ensino Superior Efetivamente.	*Não*	
9	**Engenharia de Software**		
9.1	- Compreensão, melhoria e rastreabilidade de cumprimento de requisitos. &	Sim	O grupo trabalhou intensivamente na busca pela reprodução mais fiel possível dos requisitos requisitados, através de recursos como o site da disciplina e reuniões com Monitores/Professor.
9.2	- Diagrama de Classes em *UML*.	Sim	Durante todo o desenvolvimento do Jogo foi elaborado um Diagrama de Classes para melhor organização e compreensão do código.
9.3	- Uso efetivo e intensivo de padrões de projeto *GOF*, *i.e.*, mais de 5 padrões.	Não	O número foi inferior a 5 padrões de projeto.
9.4	- Testes à luz da Tabela de Requisitos e do Diagrama de Classes.	Sim	A Tabela de Requisitos e o Diagrama de Classes fornecidos pelo professor foram os principais guias no desenvolvimento do jogo.
10	**Execução de Projeto**		

10.1	- Controle de versão de modelos e códigos automatizado (via github e/ou afins). & - Uso de alguma forma de cópia de segurança (*i.e.*, *backup*).	Sim	Utilizou-se um repositório no Github para compartilhamento do código entre os participantes. link: https://github.com/danisommer/JOGO-PLATAFORMA
10.2	- Reuniões com o professor para acompanhamento do andamento do projeto. **[ITEM OBRIGATÓRIO PARA A ENTREGA DO TRABALHO]**	Sim	Reuniões realizadas nos dias 17/11, às 10h30 e 21/11, às 14h00.
10.3	- Reuniões com monitor da disciplina para acompanhamento do andamento do projeto. **[ITEM OBRIGATÓRIO PARA A ENTREGA DO TRABALHO]**	Sim	Reuniões com o Monitor Ariel realizadas nos dias 18/11, das 11h às 13h10 e 11/11, das 11h às 13h30. Ambos os integrantes estiveram presentes nesses períodos citados. Além disso, Daniel esteve presente no primeiro Peteco, enquanto Enzo esteve nos quatro.
10.4	- Revisão do trabalho escrito de outra equipe e vice-versa.	Sim	Otávio e Letícia, da S73.
Total de conceitos apropriadamente utilizados. (Cada grande tópico vale 10% do total de conceitos. Assim, por exemplo, caso se tenha feito metade de um tópico, então valeria 5%.)			**75%** (setenta e cinco por cento).

6. DIVISÃO DO TRABALHO

Este capítulo aborda a divisão do trabalho realizada durante o desenvolvimento do projeto, detalhando as atividades realizadas por cada membro da equipe e fornecendo uma visão geral das contribuições individuais. A tabela apresenta uma lista abrangente de atividades, juntamente com os responsáveis por cada uma delas. A divisão de tarefas foi crucial para o progresso eficiente do projeto, garantindo que diferentes aspectos, desde a implementação de animações até a criação de listas de entidades, fossem abordados de maneira coordenada. A colaboração entre os membros da equipe, liderada principalmente por Daniel e Enzo, foi fundamental para alcançar os objetivos estabelecidos. A natureza específica das contribuições de cada membro reflete uma abordagem equilibrada, com Daniel liderando a maioria das atividades e Enzo desempenhando um papel significativo em áreas específicas. Essa distribuição de trabalho proporcionou uma variedade de habilidades e perspectivas, enriquecendo assim o desenvolvimento do projeto.

Tabela 4. Lista de Atividades e Responsáveis.

Atividades.	**Responsáveis**
Implementação de Animações	Daniel
Implementação de Inimigos	Daniel
Implementação de Projéteis	Daniel
Implementação de Jogadores	Daniel
Implementação de Entidades	Daniel
Implementação de Obstáculos	Daniel
Implementação de Gerenciadores	Daniel
Implementação de Menus e Telas	Daniel
Implementação da parte Gráfica	Daniel
Implementação do Salvamento de Informação	Daniel
Implementação do Sistema de Ranqueamento	Daniel
Revisão e Organização do Código	Daniel
Implementação de Criação de Entidades	Daniel
Programação em C++	Mais Daniel que Enzo
Implementação da Classe Principal	Daniel e Enzo
Compreensão de Requisitos	Daniel e Enzo
Implementação de Fases	Daniel e Enzo
Relatório	Daniel e Enzo
Slides para apresentação	Daniel e Enzo
Implementação de Listas de Entidades	Daniel e Enzo
Diagramas de Classes	Mais Enzo que Daniel
Revisão do Trabalho	Daniel, Enzo, Otávio e Letícia

Com base na tabela acima, aqui está a distribuição aproximada do trabalho:

Daniel trabalhou em 100% das atividades ou as realizando ou colaborando nelas efetivamente.

Enzo trabalhou em 45% das atividades ou as realizando ou colaborando nelas efetivamente.

AGRADECIMENTOS PROFISSIONAIS

Por fim, dedicamos um agradecimento especial ao Monitor Ariel, que disponibilizou seu tempo para nos auxiliar no desenvolvimento e na compreensão dos requisitos estipulados.

REFERÊNCIAS CITADAS NO TEXTO

http://www.dainf.ct.utfpr.edu.br/~jeansimao/Fundamentos2/Fundamentos2.htm.

REFERÊNCIAS UTILIZADAS NO DESENVOLVIMENTO

[A] Site da disciplina:
https://pessoal.dainf.ct.utfpr.edu.br/jeansimao/Fundamentos2/TopicosTrab/TecProg_Trabalho_Instrucoes_BibliotecaGrafica.htm