

TRABALHO PARA A DISCIPLINA DE TÉCNICAS DE PROGRAMAÇÃO DO CURSO DE ENGENHARIA DE COMPUTAÇÃO DA UTFPR: *WWII++*

Gabriel Cassol Bach, Gabriel Moro Conke
gabrielbach@alunos.utfpr.edu.br, gabrielconke@alunos.utfpr.edu.br

Disciplina: **Técnicas de Programação – CSE20 / S71** – Prof. Dr. Jean M. Simão
Departamento Acadêmico de Informática – DAINF - Campus de Curitiba
Curso Bacharelado em: Engenharia da Computação
Universidade Tecnológica Federal do Paraná - UTFPR
Avenida Sete de Setembro, 3165 - Curitiba/PR, Brasil - CEP 80230-901

Resumo – A disciplina de Técnicas de Programação exige o desenvolvimento de um software, no formato de um jogo de plataforma, para fins de aprendizado de técnicas de engenharia de software, além de programação orientada a objetos em C++. Para realizar este projeto, desenvolveu-se o jogo WWII++, em que um jogador tenta sobreviver diante de vários obstáculos e inimigos. Para o desenvolvimento do jogo foram considerados os requisitos textualmente propostos e elaborada modelagem (análise e projeto) via Diagrama de Classes em Linguagem de Modelagem Unificada (Unified Modeling Language - UML) usando como base um diagrama genérico proposto previamente. Posteriormente, o projeto foi desenvolvido em linguagem de programação C++, contemplando-se os conceitos usuais de Orientação a Objetos como Classe, Objeto e Relacionamento, bem como alguns conceitos avançados como Classe Abstrata, Polimorfismo, Gabaritos, Persistências de Objetos por Arquivos, Sobrecarga de Operadores e Biblioteca Padrão de Gabaritos (Standard Template Library - STL). Depois da implementação, os testes do jogo feitos pelos próprios desenvolvedores demonstraram sua funcionalidade conforme os requisitos e a modelagem elaborada. Por fim, ressalta-se que o desenvolvimento em questão foi uma fonte valiosa de aprendizado e experiência para os discentes.

Palavras-chave ou Expressões-chave: Programação Orientada a Objetos; C++; Jogo de Plataforma; Engenharia de Software.

Abstract - *Programing Techniques is a subject that requires the development of software in the shape of a platform videogame with the objective of learning software engineering techniques, as well as object-oriented programming in C++. To carry out this project, the game WWII++ was developed, where a player tries to survive against many enemies and obstacles. For the development of the game the textually proposed requirements were considered and modeling was elaborated (analysis and design) via Class Diagram in Unified Modeling Language (UML) using a previously proposed generic diagram as a base. Posteriorly, the project was developed using C++ programming language, contemplating usual Object Orientation concepts such as classes, objects, relationships, as well as some advanced concepts, such as abstract classes, polymorphism, templates, object persistence through files, operator overload, and the Standard Template Library (STL). After the implementation, the game tests made by the developers demonstrated its functionality according to the requirements and the modeling elaborated. At last, it stands out that the development in question was a valuable source of learning and experience for the students.*

Keywords or Key-expressions. Object-Oriented Programming; C++; Platform Videogames; Software Engineering.

INTRODUÇÃO

Este documento busca relatar o desenvolvimento do projeto final da disciplina de Técnicas de Programação, cujo objetivo é desenvolver habilidades fundamentais para programadores e engenheiros da computação, como: trabalhar em equipe; capacidade de criar e desenvolver projetos utilizando conceitos básicos de Engenharia de Software; programar orientado a objetos em C++. Ademais, todo o projeto descrito neste relatório foi baseado em

modelos disponibilizados pelo professor da disciplina [1]. Com estes objetivos, desenvolveu-se o jogo de plataforma que será apresentado subsequentemente.

Para realizar o trabalho, os alunos leram o documento fornecido pelo docente que contém toda a explicação do projeto e seguiram o ciclo clássico de engenharia de software, modelando a aplicação em formato de diagrama de classes a partir dos requisitos fornecidos e, posteriormente, implementaram o programa e testaram-o o máximo possível.

O funcionamento do jogo, sua implementação e os conceitos de Orientação a Objetos utilizados serão detalhados nos tópicos abaixo.

O JOGO

WWII++ é um jogo de plataforma no qual um ou dois soldados tentam sobreviver a obstáculos e inimigos divididos em duas fases, as quais se diferenciam pelas entidades encontradas, cenários e níveis de dificuldade.

Quando ocorre a execução do programa o menu principal é exibido e o usuário pode escolher dentre algumas opções, como a quantidade de jogadores e qual fase será executada. Além disso, existe o menu de pausa, que é executado quando o usuário está dentro de uma fase e deseja pausar, ressalta-se que esse menu permite ao jogador voltar ao menu principal, salvar a jogada e retornar ao jogo.

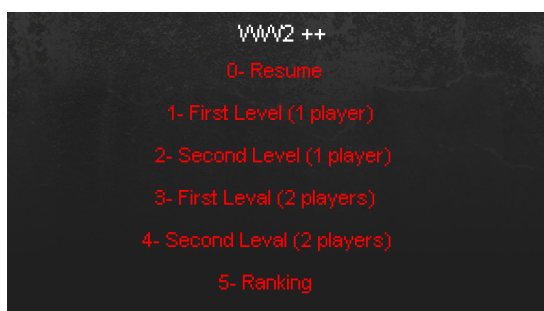


Figura 1: menu principal

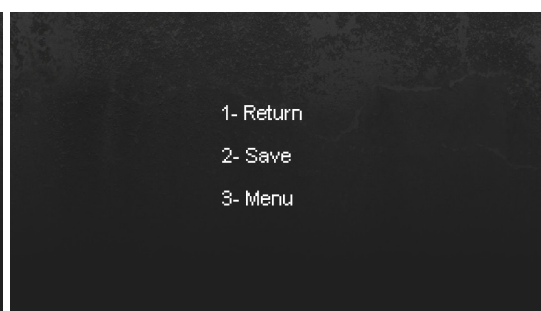


Figura 2: menu de pausa

O objetivo de cada nível é fazer com que um jogador ultrapasse todos os obstáculos e inimigos para chegar à posição final da fase, essa que se encontra no canto superior direito no primeiro nível e canto inferior direito no segundo nível. Nota-se que a quantidade de obstáculos e inimigos em cada fase é aleatória, podendo variar de 3 a 5 instâncias.

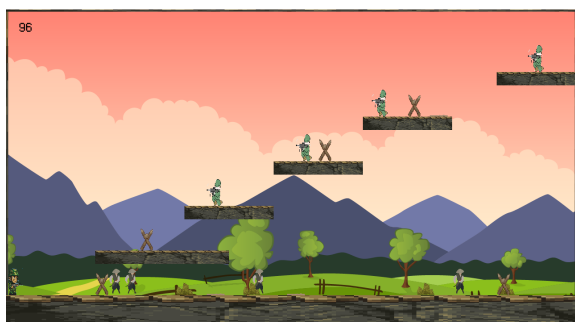


Figura 3: possível configuração nível 1

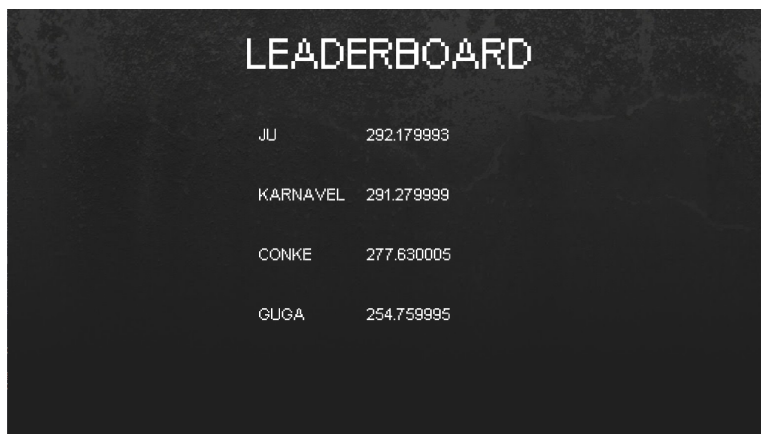


Figura 4: possível configuração nível 2

Cabe ressaltar que não há obrigatoriedade da eliminação dos adversários, no entanto um dos desafios adicionais é tentar acumular o maior número de pontos, sendo que para isso são relevantes fatores como o tempo necessário para chegar ao final da fase e a neutralização de inimigos, que é realizada através de projéteis lançados pelos jogadores. Há de se notar,

também, a presença de um chefe na segunda fase, esse que é representado por um militar e é capaz de matar o jogador com apenas um tiro.

Por fim, caso a pessoa tenha vencido o jogo, aparecerá a possibilidade dela cadastrar seu nome junto com sua pontuação. Se essa pontuação estiver dentro das quatro melhores, aparecerá no leaderboard.



LEADERBOARD	
JU	292.179993
KARNAVEL	291.279999
CONKE	277.630005
GUGA	254.759995

Figura 5: exemplo de leaderboard

DESENVOLVIMENTO DO JOGO NA VERSÃO ORIENTADA A OBJETOS

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

N.	Requisitos Funcionais	Situação	Implementação
1	Apresentar graficamente menu de opções aos usuários do Jogo, no qual pode se escolher fases, ver colocação (<i>ranking</i>) de jogadores e demais opções pertinentes.	Requisito previsto inicialmente e realizado.	Cumprido via classe <i>MenuCore</i> e suas respectivas derivadas.
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.	Cumprido via classes presentes nos <i>namespaces Menu, Levels e Characters</i> .
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.	Cumprido via classes <i>Menu, FirstLevel</i> e <i>SecondLevel</i> .
4	Ter pelo menos três tipos distintos de inimigos, cada qual com sua representação gráfica, sendo que ao menos um dos inimigos deve ser	Requisito previsto inicialmente e realizado.	Cumprido via classes dentro dos <i>namespaces Characters e Entities</i> .

	capaz de lançar projétil contra o(s) jogador(es) e um dos inimigos dever ser um 'Chefão'.		
5	Ter a cada fase ao menos dois tipos de inimigos com número aleatório de instâncias, podendo ser várias instâncias e sendo pelo menos 3 instâncias por tipo.	Requisito previsto inicialmente e realizado.	Cumprido via classes dentro do <i>namespace Characters</i> e <i>Levels</i> , com o auxílio dos arquivos de texto dentro da pasta <i>Data</i> .
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.	Cumprido via classes dentro do <i>namespace Obstacles</i> .
7	Ter em cada fase ao menos dois tipos de obstáculos com número aleatório de instâncias (<i>i.e.</i> , objetos), sendo pelo menos 3 instâncias por tipo.	Requisito previsto inicialmente e realizado.	Cumprido via classes dentro do <i>namespace Obstacles</i> e <i>Levels</i> , com o auxílio dos arquivos de texto dentro da pasta <i>Data</i> .
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.	Cumprido via classes dentro dos <i>namespaces Obstacles</i> , <i>Levels</i> e <i>Managers</i> .
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 da gravidade no âmbito deste jogo de plataforma vertical e 2D.	Requisito previsto inicialmente e realizado.	Cumprido via classe <i>Collision_Manager</i> e classes dentro do <i>namespace Entities</i> .
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 (<i>ranking</i>). E (2) Pausar e Salvar Jogada. (ok)	Requisito previsto inicialmente e realizado.	Cumprido via classes dentro do <i>namespace Menus</i> , com o auxílio dos arquivos de texto dentro da pasta <i>Data</i> .
Total de requisitos funcionais apropriadamente realizados.			100% (cem por cento).

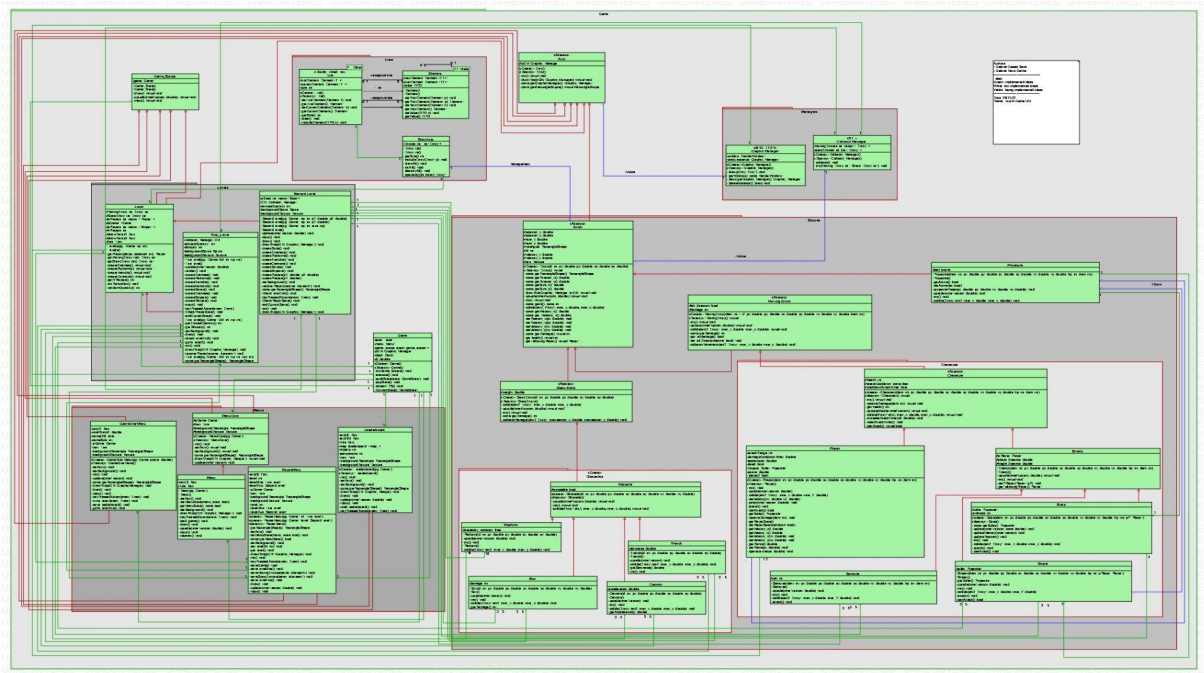


Figura 6. Diagrama de Classes em UML.

A classe principal do jogo é a *Game*, sendo que ela atua utilizando uma pilha de estados de jogo, ou seja, cada instância do jogo que é criada (“menu”, “second level”...) é adicionada na pilha e, por consequência do comportamento “last in, first out”, ela que será executada prioritariamente. Nota-se que a classe principal atua em consonância com a classe *GameState*, que indica quais classes são consideradas ou não estados de jogo e articula o funcionamento do código utilizando polimorfismo. Ademais, a classe *Game* sempre terá como “fundo” da pilha uma instância do *menu*, que por meio de suas funções poderá adicionar outros estados de jogo à estrutura. Em seguida, serão apresentadas as diversas classes que atuam auxiliando no funcionamento dessa máquina de estados.

A classe *Graphic_Manager* atua diretamente com a biblioteca gráfica *SFML*, sendo a responsável pela impressão de imagens, desenhos, retângulos e outros componentes gráficos na tela.

Collision_Manager é a classe responsável por administrar todas as colisões que ocorrem no jogo. Ela possui como atributos duas listas, uma contendo todas as entidades não-móveis do jogo e outra contendo todas as entidades móveis. Por meio delas que ocorrem as checagens dos diferentes tipos de colisões que relacionam-se com as entidades.

O pacote *Menus* é responsável por fornecer ao usuário diferentes opções para a utilização do jogo. Por meio da classe *GameOverMenu* grava-se a pontuação, por meio da classe *PauseMenu* pode-se salvar o atual progresso e por meio da classe *Leaderboard* se fornece a classificação dos jogadores de acordo com a pontuação.

Lists é o pacote responsável por criar a base das diversas listas existentes no jogo, sendo tal processo feito através de um template presente na classe *List*, juntamente com a classe aninhada *Element*.

O pacote *Entities* comporta todos os objetos que fazem parte dos níveis e interagem entre si. Isso inclui os jogadores, projéteis, inimigos e obstáculos. Ademais, todos eles sofrem o efeito da gravidade, sendo que o algoritmo que controla como a gravidade atua sobre os objetos é baseado no método de euler semi implícito, em que a velocidade no eixo y é atualizada somando-se a velocidade anterior com a velocidade atual vezes a variação do

tempo e a posição no eixo y é atualizada somando-se a posição em y anterior mais a velocidade em y,

Por fim, há o pacote *Levels*. Em síntese, os objetos dentro desse pacote gerenciam as entidades que compõem uma fase. Dentro do pacote estão as classes *FirstLevel* e *SecondLevel*, sendo que a principal diferença entre elas está no fato de que o nível secundário possui objetos da classe *Boss*, enquanto que a primeira fase pode ser executada com duas dificuldades diferentes.

TABELA DE CONCEITOS UTILIZADOS E NÃO UTILIZADOS

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

N. .	Conceitos	Uso	Onde / O quê
1	Elementares:		
	- Classes, objetos. & - Atributos (privados), variáveis e constantes. & - Métodos (com e sem retorno).	Sim	Todos .hpp e .cpp
	- Métodos (com retorno <i>const</i> e parâmetro <i>const</i>). & - Construtores (sem/com parâmetros) e destrutores	Sim	Todos .hpp e .cpp
	- Classe Principal.	Sim	main.cpp & Game.hpp/.cpp
	- Divisão em .hpp e .cpp.	Sim	No desenvolvimento como um todo, como, por exemplo, nas classes nos <i>namespaces</i> <i>Menus</i> , <i>Managers</i> , <i>Levels</i> e <i>Entities</i> .
2	Relações de:		
	- Associação direcional. & - Associação bidirecional.	Sim	Agregação bidirecional entre as classes <i>List</i> e <i>Element</i> . Agregação direcional entre as classes <i>Entity</i> e <i>EntityList</i> .
	- Agregação via associação. & - Agregação propriamente dita.	Sim	Agregação via associação na classe <i>Game</i> , por exemplo. Agregação propriamente dita na classe <i>Level</i> , por exemplo.
	- Herança elementar. & - Herança em diversos níveis.	Sim	Presente nas classes dentro dos <i>namespaces</i> <i>Entities</i> , <i>Characters</i> e <i>Obstacles</i> .
	- Herança múltipla.	Sim	Utilizado na classe <i>MenuCore</i> e <i>Level</i> .
3	Ponteiros, generalizações e exceções		
	- Operador <i>this</i> .	Sim	Utiliza-se nas classes <i>Game</i> , <i>Player</i> , <i>FirstLevel</i> e <i>SecondLevel</i> .

	- Alocação de memória (<i>new & delete</i>).	Sim	Principalmente nas classes presentes nos <i>namespaces Lists</i> e <i>Levels</i> .
	- Gabaritos/ <i>Templates</i> criada/adaptados pelos autores (e.g. Listas Encadeadas via <i>Templates</i>).	Sim	Implementado nas classes <i>List</i> e <i>Element</i> (aninhada em <i>List</i>).
	- Uso de Tratamento de Exceções (<i>try catch</i>).	Sim	Utilizado no método <i>recover</i> da classe <i>Menu</i> .
4	Sobrecarga de:		
	- Construtoras e Métodos.	Sim	Utilizado principalmente nas classes <i>FirstLevel</i> e <i>SecondLevel</i> .
	- Operadores (2 tipos de operadores pelo menos).	Sim	Implementado na classe <i>EntityList</i> (operador []) e na classe <i>Player</i> (operadores + e -)

	Persistência de Objetos (via arquivo de texto ou binário)		
	- Persistência de Objetos.	Sim	Por meio das classes dentro dos <i>namespaces Menus</i> e <i>Levels</i> .
	- Persistência de Relacionamento de Objetos.	Sim	Idem item anterior.
5	Virtualidade:		
	- Métodos Virtuais.	Sim	Utilizam-se métodos virtuais principalmente nas classes dentro do <i>namespace Entities</i> .
	- Polimorfismo.	Sim	Observa-se o polimorfismo nas classes presentes no <i>namespace Entities</i> e nas classes derivadas de <i>GameState</i> .
	- Métodos Virtuais Puros / Classes Abstratas.	Sim	Na classe <i>GameState</i> .
	- Coesão e Desacoplamento efetiva e intensa com o apoio de padrões de projeto.	Não	-
6	Organizadores e Estáticos		
	- Espaço de Nomes (<i>Namespace</i>) criados pelos autores.	Sim	Foram criados os <i>namespaces Menus, Managers, Lists, Levels, Entities, Characters</i> e <i>Obstacles</i> .
	- Classes aninhadas (<i>Nested</i>) criada pelos autores.	Sim	Utilizado nas classes <i>List</i> e <i>Element</i> (aninhada em <i>List</i>).
	- Atributos estáticos e métodos estáticos.	Sim	Utilizados na classe <i>Graphic Manager</i> .
	- Uso extensivo de constante (<i>const</i>) parâmetro, retorno, método...	Não	-
7	Standard Template Library (STL) e String OO		

	- A classe Pré-definida String ou equivalente. & - <i>Vector</i> e/ou <i>List</i> da <i>STL</i> (p/ objetos ou ponteiros de objetos de classes definidos pelos autores)	Sim	A classe <i>String</i> foi utilizada em diversos momentos nas classes dentro dos <i>namespaces</i> <i>Menus</i> e <i>Levels</i> . <i>Vector</i> e <i>List</i> foram utilizados nas classes <i>Collision_Manager</i> e <i>Levels</i> .
	- Pilha, Fila, Bifila, Fila de Prioridade, Conjunto, Multi-Conjunto, Mapa OU Multi-Mapa.	Sim	Utiliza-se uma Pilha na classe <i>Game</i> e um Multi-Mapa na classe <i>Leaderboard</i> .
Programação concorrente			
	- <i>Threads</i> (Linhas de Execução) no âmbito da Orientação a Objetos, utilizando Posix, C-Run-Time OU Win32API ou afins.	Não	-
	- <i>Threads</i> (Linhas de Execução) no âmbito da Orientação a Objetos com uso de Mutex, Semáforos, OU Troca de mensagens.	Não	-

8	Biblioteca Gráfica / Visual		
	- Funcionalidades Elementares. & - Funcionalidades Avançadas como: <ul style="list-style-type: none"> • tratamento de colisões • duplo <i>buffer</i> 	Sim	Carregar e desenhar imagens na tela, escrever texto e duplo buffer por meio da biblioteca SFML e tratamento de colisões através da classe <i>Collision Manager</i> .
	- Programação orientada a evento em algum ambiente gráfico. OU - <i>RAD – Rapid Application Development</i> (Objetos gráficos como formulários, botões etc).	Sim	Por meio da utilização dos eventos da biblioteca SFML na implementação das classes derivadas de <i>MenuCore</i> .
	Interdisciplinaridades por meio da utilização de Conceitos de Matemática e/ou Física.		
	- Ensino Médio.	Sim	Aceleração, velocidade e uso de coordenadas cartesianas.
9	- Ensino Superior.	Sim	Método de euler semi-implícito para a implementação do efeito da gravidade sobre as classes presentes no <i>namespace Entities</i> .
	Engenharia de Software		
	- Compreensão, melhoria e rastreabilidade de cumprimento de requisitos. &	Sim	Por meio das reuniões com os monitores.
	- Diagrama de Classes em <i>UML</i> .	Sim	No projeto como um todo utilizando o software StarUml.
	- Uso efetivo e intensivo de padrões de projeto GOF, i.e, mais de 5 padrões.	Não	-

	- Testes à luz da Tabela de Requisitos e do Diagrama de Classes.	Sim	Realizado pelos desenvolvedores e em conjunto com os monitores.
10	Execução de Projeto		
	- Controle de versão de modelos e códigos automatizado (via SVN e/ou afins) OU manual (via cópias manuais). & - Uso de alguma forma de cópia de segurança (backup).	Sim	Uso da plataforma <i>Github</i> para controle de versão e backup.
	- Reuniões com o professor para acompanhamento do andamento do projeto.	Sim	Total de reuniões: 4 - 27/10/2022; - 03/11/2022; - 10/11/2022; - 17/11/2022.
	- Reuniões com monitor da disciplina para acompanhamento do andamento do projeto.	Sim	Total de reuniões: 12 - 11/10/2022; - 18/10/2022; - 20/10/2022; - 21/10/2022; - 26/10/2022; - 28/10/2022; - 08/11/2022; - 09/11/2022; - 11/11/2022; - 16/11/2022; - 18/11/2022; - 22/11/2022.
	- Revisão do trabalho escrito de outra equipe e vice-versa.	Sim	Dupla Enzo Gaio e Felipe Stillner.
Total de conceitos apropriadamente utilizados.			87,5% (oitenta e sete vírgula cinco por cento)

Tabela 3. Lista de Justificativas para Conceitos Utilizados.

No.	Conceitos	Situação
1	Elementares	Classes, objetos, atributos, métodos e classe Principal foram utilizados por serem partes fundamentais da programação orientada a objetos.
2	Relações	Relações de associação, agregação e herança foram utilizadas por serem partes fundamentais da programação orientada a objetos.
3	Ponteiros, generalizações e exceções	O operador <i>this</i> foi utilizado para simplificar algumas partes do código. O uso de memória dinâmica permitiu a variação do número de objetos existentes no programa em tempo de execução. Templates permitiram simplificar e reutilizar várias partes do código. Exceções foram utilizadas para notificar erros de execução junto com try-catch.
4	Sobrecarga de métodos / Persistência de Objetos	A sobrecarga de métodos, construtoras e operadores foram utilizadas a fim de simplificar várias operações e aumentar a reusabilidade do código. A persistência de

		objetos foi implementada a fim de cumprir com os requisitos funcionais propostos.
5	Virtualidade	Classes abstratas, funções virtuais e polimorfismo foram utilizadas por serem partes integrais da programação orientada a objetos.
6	Organizadores e Estáticos	Os espaços de nomes foram utilizados a fim de organizar o código, identificando a qual parte do programa cada classe pertence. Atributos e métodos estáticos foram utilizados a fim de que todas as classes possuíssem acesso a eles.
7	STL e String OO Programação Concorrente	A classe String e os gabaritos e classes da STL foram utilizados a fim de simplificar o trabalho dos desenvolvedores e acelerar o desenvolvimento do projeto.
8	Biblioteca Gráfica	Utilizou-se a biblioteca gráfica SFML pela facilidade de implementação da parte gráfica do jogo e por ser possível integrá-la com sistemas operacionais diferentes.
9	Engenharia de Software	Requisitos, diagramas e testes foram contemplados por serem intrínsecos ao ciclo clássico de engenharia de software.
10	Execução de projeto	As reuniões com os monitores/professor foram realizadas a fim de assegurar a conformidade com os requisitos e serviram de direcionamento durante a execução do projeto. O controle de versão foi utilizado para integrar o trabalho entre os discentes, versionar o projeto e gerar cópias de segurança.

REFLEXÃO COMPARATIVA ENTRE DESENVOLVIMENTOS

Nota-se, primeiramente, que o paradigma orientado a objetos permite uma maior reutilização de código do que o procedimental. Isto pode ser exemplificado pelo conceito de polimorfismo, em que a chamada de um método e a execução de uma de suas especializações permite, com poucas linhas de código, gerar padrões de software complexos e refinados. Além disso, é importante ressaltar que a utilização da orientação a objetos não necessariamente é melhor do que a procedural, ambas possuem suas vantagens e o uso de cada uma depende do cenário em questão.

Ademais, ressalta-se que, na percepção dos discentes, é mais difícil pensar nos relacionamentos entre os objetos que resultam na execução desejada do programa do que em uma sequência de passos para realizar uma ação. Contudo, a primeira é mais fácil de modelar e analisar. Desse modo, nota-se que o paradigma orientado a objetos é extremamente útil para uma quantia muito grande de aplicações, seja dentro do meio acadêmico ou empresarial.

DISCUSSÃO E CONCLUSÕES

Durante o desenvolvimento do software, os alunos notaram que é fundamental planejar e avaliar o planejamento antes e durante a implementação do projeto. Isso porque, sem um direcionamento prévio, é fácil criar um código confuso que precisa sofrer

manutenções constantes, o que consome bastante tempo. Também, é sempre importante avaliar as soluções para os diversos problemas do código, dado que, às vezes, existem caminhos complexos e simples para resolver o mesmo problema, cuja dificuldade só é conhecida por quem já passou pela mesma situação. Isto posto, percebe-se que o aprendizado e a experiência adquirida pelos discentes durante a execução do trabalho os prepararam para adentrar no meio acadêmico ou empresarial muito mais conscientes de diversos desafios que geralmente não são vistos em sala de aula.

DIVISÃO DO TRABALHO

Tabela 4. Lista de Atividades e Responsáveis.

Atividades.	Responsáveis
Levantamento de Requisitos	Cassol e Conke
Diagramas de Classes	Cassol e Conke
Programação em C++	Cassol e Conke
Implementação de <i>Template</i>	Conke
Implementação da Persistência dos Objetos	Mais Conke que Cassol
Implementação de State	Cassol
Implementação de Menus	Cassol e Conke
Implementação de Entidades	Cassol e Conke
Implementação de Gerenciadores	Cassol e Conke
Implementação de Fases	Cassol e Conke
Implementação de Listas	Conke
Implementação de Interface Gráfica	Cassol e Conke
Implementação de Ranking	Cassol e Conke
Escrita do Trabalho	Mais Cassol que Conke
Revisão do Trabalho	Mais Cassol que Conke
Preparação da apresentação	Conke

- O aluno Gabriel Cassol Bach trabalhou em 95% das atividades ou as realizando ou colaborando nelas efetivamente;
- O aluno Gabriel Moro Conke trabalhou em 95% das atividades ou as realizando ou colaborando nelas efetivamente.
-

AGRADECIMENTOS

Agradecimentos aos monitores da disciplina e aos colegas discentes Felipe Stillner, Enzo Gaio, Gabriel Linke, Gustavo Chierici, Matheus Kunnen e Lucas Eduardo Bonancio Skora pela assistência e conhecimentos outorgados durante o desenvolvimento do projeto.

REFERÊNCIAS CITADAS NO TEXTO

[1] SIMÃO, J. M. Site das Disciplina de Técnicas de Programação, Curitiba – PR, Brasil, Acessado em 28/11/2022, às 01:17:
<http://www.dainf.ct.utfpr.edu.br/~jeansimao/Fundamentos2/Fundamentos2.htm>

REFERÊNCIAS UTILIZADAS NO DESENVOLVIMENTO

[A] SIMÃO, J. M. Site das Disciplina de Técnicas de Programação, Curitiba – PR, Brasil, Acessado em 28/11/2022, às 01:17:

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

[B] DEITEL, H M. **C++ Como Programar**: Introdução a classes e objetos com UML. 5. ed. São Paulo: Pearson Prentice Hall, 2005.

[C] Syetm, Sonar. SFML 2.1 Tutorial Series. YouTube, 6 de agosto de 2014. Disponível em: https://www.youtube.com/watch?v=FLpD54gx_5w&list=PLRtjMdoYXLf776y4K432eL_qPw4na_py3. Acesso em: 15 de outubro de 2022.

[D] Documentation for SFML 2.5.1. Sfml-dev, 2018. Disponível em <https://www.sfml-dev.org/documentation/2.5.1/annotated.php>. Acesso em: 12 de outubro de 2022.