

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

Bruno Lapa, Matheus Cruz da Silva

brunolapa@aluno.utfpr.edu.br, silvam.2020@aluno.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/Sistemas de Informação

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 tal, desenvolveu-se o jogo *CastleVania++*, no qual o jogador principal controla um explorador que busca escapar de um Castelo antigo repleto de monstros. Para o desenvolvimento do jogo foram considerados os requisitos propostos pelo professor e elaborada a modelagem (análise e projeto) usando como recurso o Diagrama de Classes em Linguagem de Modelagem Unificada (*Unified Modeling Language – UML*). Subsequentemente, o projeto foi desenvolvido em linguagem de programação C++, que contemplou 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 e partidas do jogo demonstraram sua funcionalidade conforme os requisitos e a modelagem. Por fim, destaca-se que o projeto em questão permitiu cumprir o objetivo de aumentar a experiência e o conhecimento dos discentes.

Palavras-chave ou Expressões-chave: Paradigma Orientado a Objetos; C ++; Desenvolvimento de jogos digitais.

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++. For that reason, the game CastleVania++ was developed. In this game, the main player controls an explorer trying to escape from an ancient temple full of monsters. All proposed requirements were considered to develop the game, which was modeled (analysis and project) using the Unified Modeling Language Class Diagram (UML). Subsequently, the software was developed using C++ programming language and 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 implementing, testing and playing the game, the developers showed its functionality according to requirements and modeling. At last, the project met the desired goals of improving the students' knowledge and experience.

Keywords or Key-expressions: Object Oriented Paradigm; C++; Video Game development.

INTRODUÇÃO

O presente documento busca relatar o desenvolvimento de um projeto para a disciplina de Técnicas de Programação, matéria obrigatória do curso de Engenharia da Computação na UTFPR. O objetivo é aplicar, avaliar e ampliar os conhecimentos adquiridos durante as aulas dessa disciplina, especificamente, no que diz respeito ao Paradigma de Programação Orientado a Objetos, à linguagem de programação C++ e aos princípios da Engenharia de Software. Esse documento, o artefato de software que ele descreve e o diagrama de classes UML produzido são baseados em modelos disponíveis no site do professor da disciplina [1].

Com esses objetivos em mente, foi criado um jogo de plataforma com temática e mecânicas escolhidas pelos discentes e aprovadas pelo docente, visando seguir os requisitos previamente determinados, que serão detalhados no presente documento.

Para atingir esse objetivo, os alunos buscaram seguir o ciclo tradicional de engenharia de software, ou seja, com a orientação do professor e do monitor da disciplina, estudaram os requisitos necessários, modelaram o software através de um diagrama de classes UML, implementaram o programa na linguagem C++ (com uso da biblioteca SFML) e testaram o programa tão extensivamente quanto possível.

As próximas seções irão explicar o jogo, sua implementação e os conceitos utilizados.

O JOGO

O jogo *CastleVania++* é um jogo de plataforma em terceira pessoa para um ou dois jogadores. Quando o programa é executado, o usuário visualiza o menu principal. Existe também o menu de pausa, que pode ser acessado apertando a tecla ESC dentro de uma fase.



Figura 1: Menu principal.

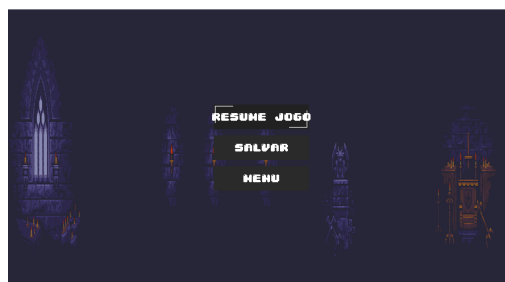


Figura 2: Menu de pausa.

O jogador deve chegar ao final de cada fase de modo a não perder totalmente sua pontuação de vidas, ou seja, de modo a colidir o menor número de vezes possível com os inimigos, projéteis e obstáculos danosos, caso o jogador venha morrer, então ele será levado para o menu principal. A classificação se dá ao número de inimigos que o jogador conseguiu destruir dentro da fase pertinente.

Cada vez que o jogador joga uma nova fase, a posição e número dos inimigos mudam aleatoriamente, de forma que cada jogada é diferente.



Figura 3: Seção da fase 1 (bosque)



Figuras 4: Seção da fase 2 (Castelo).

O primeiro e o segundo jogador podem mover-se para os lados, pular e atacar. Assim como, um pode auxiliar o outro a avançar na fase.



Figura 5: Dois jogadores com movimentos similares, na fase 1(bosque).

Para vencer o chefe, que se encontra no final da segunda fase(Castelo) e que lança raios danosos, é necessário que o jogador tenha preservado sua pontuação de vida, ou no caso de dois jogadores, que ao menos um deles esteja com um alto nível de saúde para derrotar o chefe.



Figura 6: Chefão atacando o jogador enquanto ele desvia dos raios e ataca o chefão.

Ao terminar a última fase e vencer o chefe, o jogador entra no menu de classificação, onde pode escrever seu nome e deixar salvo sua pontuação.



Figura 7: Menu de classificação ao finalizar segunda fase.

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.	Requisito cumprido via classe Menu e Botão.
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 via classe Jogador cujos objetos são agregados em jogo, podendo instanciar os dois jogadores.
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 pacote Nível e pelo objeto da classe Menu, onde as fases podem ser escolhidas no menu 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 dever ser um 'Chefão'.	Requisito previsto inicialmente e realizado.	Requisito cumprido via classe Inimigo, e dos objetos derivados da classe nível e da classe Projéteis.
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.	Inimigos gerenciados e criados de maneira aleatória pelo ConstrutorNível, em cada fase instancia uma quantidade aleatória.
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 cumprido via classe Obstáculos e seus objetos derivados.
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 cumprido via Gerenciador de Nível e Construtor Nível e seus derivados.
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 por meio da classe Nível e objetos de classes derivados de ConstrutorNível.
9	Gerenciar colisões entre jogador para com inimigos e seus projeteis, 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.	Cumprido por meio do uso das classes ListaEntidade e GerenciadorColisao e seus derivados, que itera as entidades e calcula suas colisões.
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.	Requisito previsto inicialmente e realizado.	Cumprido pelas classes GerenciadorEventos, PauseMenuState, CarregarJogoState e funções de salvar incluídas em cada Entidade.

Total de requisitos funcionais apropriadamente realizados.

(Cada tópico vale 10%, sendo que para ser contabilizado deve estar realizado efetivamente e não parcialmente)

100% (cem por cento).

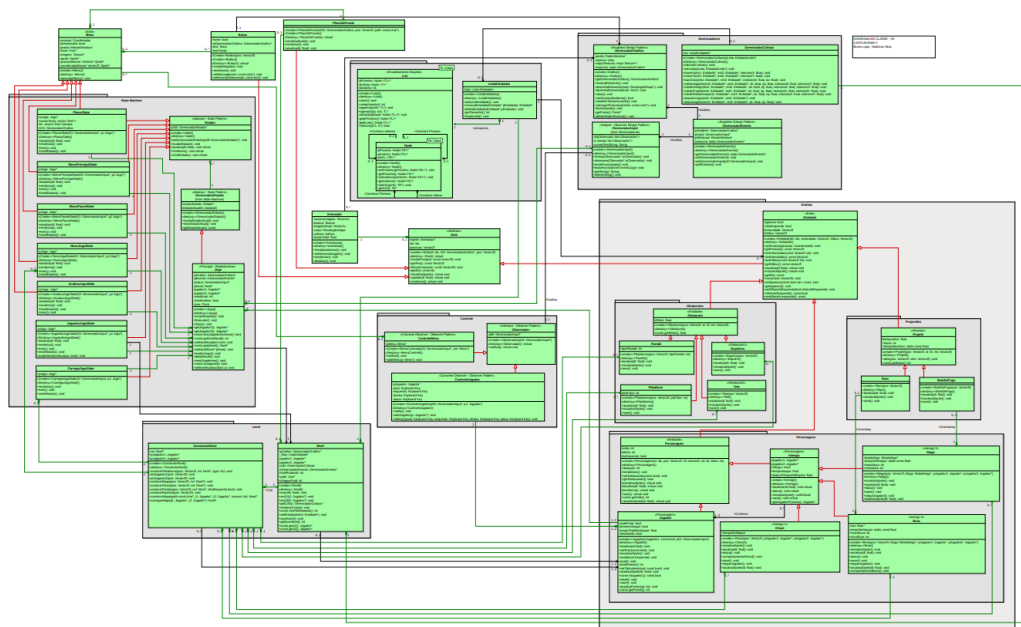


Figura 8: Diagrama de Classes UML.¹

¹ Devido a dimensão do diagrama, é impossível colocar o diagrama UML em tamanho legível dentro desse documento, porém, o arquivo encontra-se em posse do professor e no github dos desenvolvedores, (vide documentação).

A classe principal do jogo é a *StateMachine*. O objeto dessa classe agrega as fases, menus e jogadores, além dos objetos das classes *Level*, *Controle* e *Botão*. Ela funciona, em resumo, como uma *State Machine*, seguindo o Padrão de Projeto GOF Patterns, executando uma fase ou menu de acordo com o valor da variável *current*. Para manter a coerência da explicação com a organização real do programa, a explicação das demais classes se dará em blocos dedicados a cada pacote.

A classe *GerenciadorGrafico*, é a classe responsável por todos os elementos gráficos do jogo, interagindo diretamente com a biblioteca gráfica utilizada SFML. Ela tem métodos para carregar e imprimir imagens, retângulos e texto, de forma que nenhum objeto precisa interagir realmente com a SFML para ser desenhado.

A classe *GerenciadorEvento* é responsável por lidar com todos os eventos relacionados ao teclado, chamando as funções “inscrites” pelos objetos sempre que um evento ocorre. Vale notar que essa classe e a descrita no parágrafo anterior são as únicas que tem alguma relação direta com a biblioteca gráfica escolhida, o que significa que seria fácil transformá-las em classes abstratas e trocar a biblioteca gráfica.

A classe *GerenciadorColisao* gerencia as colisões das entidades físicas entre si e com o mapa. Quando seu método *calculaColisao* é chamado, ela verifica quais entidades colidiram com quais outras comparando suas posições e tamanhos e usando o método ID de cada entidade, sabendo se trata-se de um inimigo ou obstáculo, por exemplo.

O pacote Entities comporta todos os objetos que fazem parte dos níveis, interagem entre si e precisam ser desenhados. Isso inclui os jogadores, projéteis, inimigos e os obstáculos. Todos dentro de Entities possuem uma posição e uma texture, ou seja, possuem um caminho para uma imagem para se desenharem em tela.

A classe Obstáculo, também é abstrata, e deriva-se dela outras classes, como Plataforma, Parede, Espinhos e Teia, cada qual com sua respectiva posição, hitbox(Tamanho) e ID, que são tratados para verificar se houve ou não colisão.

Em suma, os objetos das classes Level gerenciam as entidades que compõem uma fase. Eles são responsáveis por criar, inicializar e chamar os métodos de desenho e atualização de suas entidades, além de salvá-las/carregá-las por uso de arquivos.txt.

O pacote State Machine funciona como uma Máquina de Estado seguindo o padrão de projeto GOF Patterns, onde é gerenciado todos os estados do jogo, entre eles o Menu, Botão e menu principal.

O pacote Controle, basicamente segue o padrão de projeto Observer, segundo GOF Patterns, onde é responsável por observar o Menu e notificar os seus dependentes, dessa maneira, otimizando o processamento e memória gastos na operação.

A classe Animação, é responsável pela atualização das sprites e dos movimentos das entidades, dessa forma ela agrega a classe Ente e conhece a classe GerenciadorGrafico.

TABELA DE CONCEITOS UTILIZADOS E NÃO UTILIZADOS

Na presente seção será realizado um estudo dos conceitos trabalhados e utilizados durante o desenvolvimento do projeto. Na tabela a seguir, é possível observar uma lista com detalhes dos conceitos utilizados ou não.

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

N.	Conceitos	Uso	Onde / O quê
1	Elementares:		
1.1	- Classes, objetos. & - Atributos (privados), variáveis e constantes. & - Métodos (com e sem retorno).	Sim	Todos arquivos .hpp e .cpp.
1.2	- Métodos (com retorno <i>const</i> e parâmetro <i>const</i>). & - Construtores (sem/com parâmetros) e destrutores	Sim	Basicamente em todos arquivos .hpp e .cpp, facilitando a manutenção do código.
1.3	- Classe Principal.	Sim	Main.cpp & Jogo.h/.cpp
1.4	- Divisão em .h e .cpp.	Sim	Em todo o projeto, seguindo boas práticas.
2	Relações de:		
2.1	- Associação direcional. & - Associação bidirecional.	Sim	Direcional entre Nível come GerenciadorGrafico; Bidirecional entre GerenciadorEstado com GerenciadorInput e GerenciadorGrafico.
2.2	- Agregação via associação. & - Agregação propriamente dita.	Sim	Agregação entre Menu (e derivadas) e MenuControle, propriamente dita entre Entidade e ListaEntidades.
2.3	- Herança elementar. & - Herança em diversos níveis.	Sim	Elementar entre Entidade e Obstaculos, diversos níveis entre Entidade e Personagens.
2.4	- Herança múltipla.	Sim	Classe de MenuPrincipalState, herança de Estado e Menu.
3	Ponteiros, generalizações e exceções		

3.1	- Operador <i>this</i> para fins de relacionamento bidirecional.	Sim	Múltiplas classes como <i>Animacao</i> e <i>ControleJogador</i> .
3.2	- Alocação de memória (<i>new & delete</i>).	Sim	Todas as classes que instanciam uma nova Entidade.
3.3	- Gabaritos/ <i>Templates</i> criada/adaptados pelos autores (e.g., Listas Encadeadas via <i>Templates</i>).	Sim	List e ListaEntidade.
3.4	- Uso de Tratamento de Exceções (<i>try catch</i>).	Sim	No desenvolvimento do projeto para testar componentes, prontamente retirado depois.
4	Sobrecarga de:		
4.1	- Construtoras e Métodos.	Sim	Classe <i>ConstrutorNivel</i> (Construtora) e <i>Animacao</i> (método <i>inicializaTextura</i>)
4.2	- Operadores (2 tipos de operadores pelo menos – Quais?).	Sim	Foi usado o <i>operator==</i> e o <i>operator++</i> . [Em <i>animacao: imagemAtual.x++ e textura ==</i>]
---	Persistência de Objetos (via arquivo de texto ou binário)		
4.3	- Persistência de Objetos.	Sim	Pacote <i>Nivel</i> .
4.4	- Persistência de Relacionamento de Objetos.	Sim	Idem anterior (relação entre diferentes Entidades)
5	Virtualidade:		
5.1	- Métodos Virtuais Usuais.	Sim	Classe <i>Obstaculo</i> , por exemplo.
5.2	- Polimorfismo.	Sim	Chamadas como <i>atualiza</i> e <i>renderiza</i> de Ente em animação.
5.3	- Métodos Virtuais Puros / Classes Abstratas.	Sim	Classe Ente.
5.4	- Coesão/Desacoplamento efetiva e intensa com o apoio de padrões de projeto.	Sim	Em todo o projeto.
6	Organizadores e Estáticos		
6.1	- Espaço de Nomes (<i>Namespace</i>) criada pelos autores.	Sim	Gerenciadores, Entidades, Controle etc..
6.2	- Classes aninhadas (<i>Nested</i>) criada pelos autores.	Sim	Classe <i>Lista</i> , por exemplo.
6.3	- Atributos estáticos e métodos estáticos.	Sim	Classe <i>GerenciadorEstado</i> , método <i>getEstadoAtual</i> . Classe <i>ID</i> , com atributos estáticos.
6.4	- Uso extensivo de constante (<i>const</i>) parâmetro, retorno, método...	Sim	Em todo o Projeto
7	Standard Template Library (STL) e String OO		
7.1	- A classe Pré-definida <i>String</i> 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	Classe Entidade, por exemplo.
7.2	- Pilha, Fila, Bifila, Fila de Prioridade, Conjunto, Multi-Conjunto, Mapa OU Multi-Mapa.	Sim	<i>GerenciadorEventos</i> (Múltiplos eventos), <i>Nivel</i> (múltiplos níveis).
---	Programação concorrente		
7.3	- <i>Threads</i> (Linhas de Execução) no âmbito da Orientação a Objetos, utilizando <i>Posix</i> , <i>C-Run-Time</i> OU <i>Win32API</i> ou afins.	Não	...
7.4	- <i>Threads</i> (Linhas de Execução) no âmbito da Orientação a Objetos com uso de <i>Mutex</i> , <i>Semáforos</i> , OU <i>Troca de mensagens</i> .	Não	...
8	Biblioteca Gráfica / Visual		

8.1	- Funcionalidades Elementares. & - Funcionalidades Avançadas como: • tratamento de colisões • duplo <i>buffer</i>	Sim	Carregar e desenhar imagens, texto e duplo buffer utilizando a biblioteca SFML.
8.2	- Programação orientada e evento efetiva (com gerenciador apropriado de eventos inclusive) em algum ambiente gráfico. OU - RAD – <i>Rapid Application Development</i> (Objetos gráficos como formulários, botões etc).	Sim	Classes GerenciadorEvento e Botao, além do uso das funcionalidades de evento do SFML.
---	Interdisciplinaridades via utilização de Conceitos de Matemática Contínua e/ou Física.		
8.3	- Ensino Médio Efetivamente.	Sim	Cálculo de velocidade e aceleração, além de coordenadas.
8.4	- Ensino Superior Efetivamente.	Sim	Cálculo de arco de projéteis e vetores.
9	Engenharia de Software		
9.1	- Compreensão, melhoria e rastreabilidade de cumprimento de requisitos. &	Sim	Progresso com reuniões entre os monitores e professor, além de registros no GitHub e versões do Diagrama de classes.
9.2	- Diagrama de Classes em <i>UML</i> .	Sim	Em todo o desenvolvimento.
9.3	- Uso efetivo e intensivo de padrões de projeto <i>GOF</i> , i.e., mais de 5 padrões.	Sim	Padrões utilizados: Singleton, State, Observer, Builder e Iterator.
9.4	- Testes à luz da Tabela de Requisitos e do Diagrama de Classes.	Sim	Reuniões com monitores e professor.
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., <i>backup</i>).	Sim	Repositório no GitHub. https://github.com/Matheussilva05/CastleVania.git
10.2	- Reuniões com o professor para acompanhamento do andamento do projeto. [ITEM OBRIGATÓRIO PARA A ENTREGA DO TRABALHO]	Sim	1ª: 02/06 2ª: 06/06.
10.3	- Reuniões com monitor da disciplina para acompanhamento do andamento do projeto. [ITEM OBRIGATÓRIO PARA A ENTREGA DO TRABALHO]	Sim	1ª: 20/05 2ª: 24/05 3ª: 29/05 4ª: 30/05
10.4	- Revisão do trabalho escrito de outra equipe e vice-versa.	Sim	Rafael Bergo e Ariel Wilson
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%.)			95% (noventa por cento).

A seguir, é apresentado a tabela 3 baseada na tabela 2, porém agora é discutido de forma sucinta uma justificativa do uso de cada conceito durante o desenvolvimento do projeto, visando assim tornar evidente a primordialidade do uso de tais conceitos quando se trata de POO.

Tabela 3. Lista de Justificativas para Conceitos Utilizados.

No.	Conceitos	Listar apenas os utilizados Situação
1	Elementares	
1.1	- Classes, objetos. &	Classe, Objetos, Atributos e Métodos foram utilizados porque são conceitos elementares na orientação a objetos.

	<ul style="list-style-type: none"> - Atributos (privados), variáveis e constantes. & - Métodos (com e sem retorno). 	Proporcionando organização e reutilização subsequente do código.
1.2	<ul style="list-style-type: none"> - Métodos (com retorno <i>const</i> e parâmetro <i>const</i>). & - Construtores (sem/com parâmetros) e destrutores 	Métodos com retorno e parâmetros <i>const</i> garantem integridade dos dados e construtores/destrutores permitem inicialização e liberação adequada de recursos.
1.3	- Classe Principal.	A classe principal é a classe que contém o ponto de entrada do programa em C++. Ela contém métodos e atributos que coordenam a execução do programa e interagem com outras classes e objetos seguindo o paradigma POO.
1.4	- Divisão em .h e .cpp.	Permite melhor organização das classes e afins que compõem o sistema com um todo.
2	Relações	
2.1	<ul style="list-style-type: none"> - Associação direcional. & - Associação bidirecional. 	Associação foi utilizada porque a assoc. direcional permite relação unidirecional enquanto a associação bidirecional é uma relação de duas vias, permitindo a comunicação e interação entre as classes.
2.2	<ul style="list-style-type: none"> - Agregação via associação. & - Agregação propriamente dita. 	Agregação via associação porque permite que uma classe seja composta por outras classes como parte de suas estruturas, enquanto que a agregação propriamente dita permite que a classe principal seja responsável pela criação e destruição das classes agregadas.
2.3	<ul style="list-style-type: none"> - Herança elementar. & - Herança em diversos níveis. 	Herança elementar permite que uma classe herde diretamente de uma classe base, enquanto que Heranças múltiplas permite formar uma cadeia de heranças formando um sistema complexo de hierarquia.
2.4	- Herança múltipla	Permite que uma classe herde atributos e comportamentos de várias classes.
3	Ponteiros, generalizações e exceções	

3.1	- Operador <i>this</i> para fins de relacionamento bidirecional.	O operador "this" é usado para estabelecer um relacionamento bidirecional entre classes em programação orientada a objetos.
3.2	- Alocação de memória (<i>new</i> & <i>delete</i>).	A alocação de memória é realizada com o operador "new" para criar dinamicamente um objeto ou um array, enquanto o operador "delete" é usado para liberar a memória alocada previamente, evitando vazamentos de memória.
3.3	- Gabaritos/ <i>Templates</i> criada/adaptados pelos autores (e.g., Listas Encadeadas via <i>Templates</i>).	Possibilita a criação de estruturas genéricas, facilitando o reuso e manutenção destas.
3.4	- Uso de Tratamento de Exceções (<i>try catch</i>).	Tratamento de exceções (<i>try-catch</i>) permite lidar com erros e exceções durante a execução de um programa.
4	Sobrecarga de:	
4.1	- Construtoras e Métodos.	Permite definir diferentes versões de um construtor ou método com parâmetros distintos, permitindo maior flexibilidade.
4.2	- Operadores (2 tipos de operadores pelo menos – Quais?).	Permite definir o comportamento dos operadores em uma classe, permitindo operações personalizadas e intuitivas entre objetos dessa classe.
4.3	- Persistência de Objetos.	permite personalizar a leitura e escrita de objetos para a persistência em diferentes formatos.

4.4	- Persistência de Relacionamento de Objetos.	recuperar objetos relacionados em um sistema de persistência, mantendo a integridade e consistência dos relacionamentos entre eles.
5	Virtualidade:	
5.1	- Métodos Virtuais Usuais.	Permitem fornecer um ponto de entrada comum para comportamentos específicos de cada classe derivada.
5.2	- Polimorfismo.	Permitem escrever código genérico e flexível, que pode lidar com diferentes tipos de objetos de forma uniforme, facilitando a extensibilidade e a reutilização de código.
5.3	- Métodos Virtuais Puros / Classes Abstratas.	Permite a criação de interfaces comuns e obriga as classes derivadas a implementarem tais métodos.
5.4	- Coesão/Desacoplamento efetiva e intensa com o apoio de padrões de projeto.	auxilia na obtenção de coesão e desacoplamento efetivos, promovendo um design flexível e de fácil manutenção.
6	Organizadores e Estáticos	
6.1	- Espaço de Nomes (<i>Namespace</i>) criado pelos autores.	evitando conflitos de nomes e facilitando a modularidade e reutilização do código.
6.2	- Classes aninhadas (<i>Nested</i>) criada pelos autores.	permitem definir classes dentro de outras classes, promovendo organização e encapsulamento do código.
6.3	- Atributos estáticos e métodos estáticos.	Permite acesso sem a necessidade de criar objetos e compartilhando informações globais.
6.4	- Uso extensivo de constante (<i>const</i>) parâmetro, retorno, método...	Permite a garantia de imutabilidade e não modificação dos valores, promovendo a segurança e integridade dos dados ao longo do código.
7	Standard Template Library (STL) e String OO	
7.1	- A classe Pré-definida <i>String</i> 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).	String: oferece funcionalidades para manipulação de strings de texto de forma conveniente e eficiente. Vector e/ou List: permite armazenar objetos ou ponteiros de objetos de classes definidos pelos autores, oferecendo estruturas de dados flexíveis e eficientes para gerenciamento e manipulação.
7.2	- Pilha, Fila, Bifila, Fila de Prioridade, Conjunto, Multi-Conjunto, Mapa OU Multi-Mapa.	São estruturas que oferecem maneiras práticas de armazenar, organizar e manipular dados em um dado programa.
7.3	- <i>Threads</i> (Linhas de Execução) no âmbito da Orientação a Objetos, utilizando Posix, C-Run-Time OU Win32API ou afins.	Não implementado por falta de tempo.
7.4	- <i>Threads</i> (Linhas de Execução) no âmbito da Orientação a Objetos com uso de Mutex, Semáforos, OU Troca de mensagens.	Não implementado por falta de tempo.
8	Biblioteca Gráfica / Visual	
8.1	- Funcionalidades Elementares. & - Funcionalidades Avançadas como: • tratamento de colisões • duplo <i>buffer</i>	Funcionalidades Elementares: Permite criar um programa simples e funcional, atendendo aos requisitos mínimos do projeto. Tratamento de colisões: Permite tratar as colisões entre diferentes entidades, evitando conflitos e sobreposições. Duplo buffer: melhora a renderização e evita efeitos indesejados.

8.2	- Programação orientada e evento efetiva (com gerenciador apropriado de eventos inclusive) em algum ambiente gráfico. OU - <i>RAD – Rapid Application Development</i> (Objetos gráficos como formulários, botões etc).	Envolve a utilização de um gerenciador apropriado de eventos para lidar com interações do usuário, como cliques de mouse e pressionamentos de teclas, permitindo a resposta adequada e atualização da interface gráfica em tempo real
8.3	- Ensino Médio Efetivamente.	Permitem uma simulação de movimentos básicos de determinadas objetos do jogo
8.4	- Ensino Superior Efetivamente.	Permitem a visualização de determinados elementos gráficos visando uma estética agradável para o usuário.
9	Engenharia de Software	
9.1	- Compreensão, melhoria e rastreabilidade de cumprimento de requisitos. &	Permitem uma abordagem pragmática e objetivo do que se deve cumprir durante o desenvolvimento do projeto
9.2	- Diagrama de Classes em <i>UML</i> .	Permite um vislumbre claro e objetivo das hierarquias e dependências entre as classes, permitindo uma visão global do projeto
9.3	- Uso efetivo e intensivo de padrões de projeto <i>GOF</i> , <i>i.e.</i> , mais de 5 padrões.	Permitem uma otimização do código em si, encapsulamento e modularização, facilitando o reuso e manutenção do projeto.
9.4	- Testes à luz da Tabela de Requisitos e do Diagrama de Classes.	Garante uma verificação pragmática do que está sendo feito dentro do projeto à luz do que se é pedido ou requerido, facilitando a correção de possíveis desvios do objetivo principal
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>i.e.</i> , <i>backup</i>).	Permite acompanhar a evolução do projeto, além do trabalho mútuo e correção de possíveis erros, além de evitar perda do código, deixando sempre um backup do mesmo.
10.2	- Reuniões com o professor para acompanhamento do andamento do projeto. [ITEM OBRIGATÓRIO PARA A ENTREGA DO TRABALHO]	Permitem um vislumbre de um especialista da área, críticas valiosas de como proceder em determinadas situações e corrigir possíveis desvios da tabela de requisitos.
10.3	- Reuniões com monitor da disciplina para acompanhamento do andamento do projeto. [ITEM OBRIGATÓRIO PARA A ENTREGA DO TRABALHO]	Proporcionam um olhar mais próximo por se tratar de um aluno, e de uma visão mais comum e possíveis discussões mais abertas, visando um entendimento claro daquilo que foi solicitado
10.4	- Revisão do trabalho escrito de outra equipe e vice-versa.	Proporciona uma visão de observador externo, visto que essa experiência proporciona correção de possíveis erros e garante um trabalho mais coerente e assertivo.

REFLEXÃO COMPARATIVA ENTRE DESENVOLVIMENTOS

O desenvolvimento procedimental é uma abordagem mais tradicional, em que o programa é estruturado em torno de procedimentos ou funções. Nesse modelo, o foco principal é na lógica procedural, ou seja, nas etapas sequenciais que devem ser executadas para alcançar um resultado

desejado. O código é organizado em funções que manipulam os dados, e o fluxo de controle é controlado principalmente por meio de estruturas de controle, como loops e condicionais.

Por outro lado, a Programação orientada a objetos (POO) se baseia nos conceitos de objetos e classes. Nesse modelo, o foco principal é na representação dos objetos do mundo real e em como eles interagem entre si. Os dados e as funções relacionadas são encapsulados em objetos, que são instâncias de classes. As classes definem o comportamento dos objetos e como eles se comunicam uns com os outros. O código é organizado em métodos de classe, e o fluxo de controle é distribuído entre os objetos por meio de chamadas de método e manipulação de eventos.

A principal diferença entre essas duas abordagens está na forma como elas lidam com a complexidade e a reutilização do código. O POO favorece a modularidade e a reutilização, permitindo que os desenvolvedores criem classes e objetos que representam entidades específicas do problema e possam ser reutilizados em diferentes partes do código. Isso resulta em um código mais flexível, escalável e de fácil manutenção.

DISCUSSÃO E CONCLUSÕES

Com base no processo de aprendizado na disciplina, é evidente o amadurecimento do raciocínio lógico e a maneira de como passamos a observar as circunstâncias ao nosso redor. Quando trabalhamos com a Programação orientada a Objetos (POO) passamos a ter uma visão mais crítica e coerente em diversos aspectos, buscando sempre o princípio da coesão e desacoplamento, visando um olhar mais lógico e naturalista de como as coisas se comportam.

Após a conclusão do projeto notamos a importância do levantamento e estudo dos requisitos, assim como da modelagem e só então a implementação, foi visto na prática a importância da Engenharia de Software para o desenvolvimento de projetos de larga escala e trabalho mútuo. Visto isso, a disciplina e o desenvolvimento do projeto foi de grande valia para os discentes, visto que pode agregar conhecimento múltiplos e diversas experiências positivas para o crescimento profissional de cada indivíduo.

DIVISÃO DO TRABALHO

Tabela 4. Lista de Atividades e Responsáveis.

Atividades.	Responsáveis
Levantamento de requisitos	Bruno e Matheus
Diagramas de classes	Bruno e Matheus
Programação em C++	Bruno e Matheus
Implementação de <i>Template</i>	Bruno e Matheus
Implementação da persistência dos objetos	Bruno e Matheus
Mecânicas de jogo	Mais Bruno que Matheus
Design das fases	Bruno e Matheus
Arte	Bruno e Matheus
Tratamento de colisões	Bruno e Matheus
Tratamento de eventos	Bruno e Matheus
Gerenciador gráfico	Bruno e Matheus
Criação das sprites para animação	Bruno e Matheus
Animação	Bruno e Matheus
Criação dos IDs	Bruno e Matheus

Inimigos, jogadores e Obstáculos	Bruno e Matheus
Menus	Bruno e Matheus
Uso de programação concorrente	Bruno e Matheus

save	Mais Bruno que Matheus
Escrita do trabalho	Bruno e Matheus
Revisão do trabalho	Bruno e Matheus
Preparação da apresentação	Bruno e Matheus


AGRADECIMENTOS

Agradecimentos à equipe Rafael Bergo e Ariel Wilson, por terem colaborado com a revisão do projeto, em especial ao aluno Rafael Bergo por ter auxiliado na configuração do compilador e da sincronização com o github; Aos monitores da disciplina, Daniel, Murilo e Giovani, em especial aos monitores Giovani e Daniel que se fizeram presentes mesmo em horários fora de sua grade obrigatória, e sempre atenderam nossas necessidades com profissionalismo e expertise.

REFERÊNCIAS CITADAS NO TEXTO

- [1] SIMÃO, J. M. Site das Disciplina de Técnicas de Programação, Curitiba – PR, Brasil, Acessado em 22/11/2019, às 08:46:
<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 22/11/2019, às 08:46:
<http://www.dainf.ct.utfpr.edu.br/~jeansimao/Fundamentos2/Fundamentos2.htm>.
- [B] TIME C++ REFERENCE. Referência da linguagem de programação C++, Berkeley - California, EUA, Acessado em 22/11/2019, às 08:52:
<https://en.cppreference.com/w/>
- [C] THE C++ RESOURCES NETWORK. Referência da linguagem de programação C++, Pasadena – California, EUA, Acessado em 24/11/2019, às 20:10:
<http://www.cplusplus.com/>
- [2] SIMÃO, J. M. Site das Disciplina de Técnicas de Programação, Curitiba – PR, Brasil, Acessado em 22/11/2019, às 08:46:
<http://www.dainf.ct.utfpr.edu.br/~jeansimao/Fundamentos2/Fundamentos2.htm>.
- [3] CRIANDO UM JOGO EM C++ DO ZERO. youtube, canal Gege++, (atual monitor) Acessado em 10/06/2023,
https://www.youtube.com/watch?v=gfGE5KY1OQU&list=PLR17O9xbTbIBBoL3li44N8LdZVvg-_uZ&index=1
- [4] SFML 2.4 For Beginners - 10: Animated Movement. youtube, canal: Hilze Vonck, Acessado em 10/06/2023.  SFML 2.4 For Beginners - 10: Animated Movement.

