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

Ian M. S. Ishikawa, Pedro H. F. Neves
ianishikawa@alunos.utfpr.edu.br, pneves.2022@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 / Sistemas de Informação
Universidade Tecnológica Federal do Paraná - UTFPR
Avenida Sete de Setembro, 3165 - Curitiba/PR, Brasil - CEP 80230-901

Resumo – Este documento apresenta o trabalho acadêmico em formato de artigo da disciplina de Técnicas de Programação. Dessa forma, os principais objetivos proposto por esse trabalho foram aprender a aplicar conceitos de Programação Orientada a Objetos, engenharia de *software,* mas também introduzir práticas de indústria por meio da produção de um jogo de plataforma em C++. Assim, para cumprir esse objetivo, foi escolhido o jogo Hero++, no qual os jogadores devem tentar eliminar todos os inimigos para avançar de fase. Para o desenvolvimento do jogo foram considerados os requisitos textualmente propostos e elaborado modelagem (análise e projeto) via Diagrama de Classes em Linguagem de Modelagem Unificada (*Unified Modeling Language* - *UML*) usando como base um diagrama genérico e prévio proposto. Subsequentemente, em linguagem de programação C++, realizou-se o desenvolvimento que contemplou os conceitos usuais de Orientação a Objetos como Classe, Objeto e Relacionamento, bem como alguns conceitos avançados como Classe Abstrata, Polimorfismo, Gabaritos, Sobrecarga de Operadores e Biblioteca Padrão de Gabaritos (*Standard Template Library* - *STL*). Após a implementação, foram realizados testes que demonstraram a funcionalidade de grande parte dos requisitos exigidos. Por fim, pode-se concluir que o jogo permitiu aplicar os conhecimentos adquiridos em sala e o desenvolvimento da *soft skill* tanto do trabalho em equipe, quanto da comunicação e escrita.

Palavras-chave ou Expressões-chave: Artigo-Relatório Modelo para o Trabalho em Técnicas de Programação, Trabalho Acadêmico Voltado a Implementação em C++, Normas Internas para Elaboração de Trabalho, Programação Orientada a Objetos

Abstract - *This document presents the academic work in the format of an article of the Programming Techniques discipline. Thus, the main objectives proposed by this work were to learn to apply Object Oriented Programming concepts, software engineering, but also to introduce industry practices through producing a platform game in C++. Thus, to fulfill this objective, the Hero++ game was chosen, in which players must try to eliminate all enemies to advance to the next level. 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 generic and previously proposed diagram as a base. Subsequently, in C++ programming language, development was carried out that included the usual Object Orientation concepts such as Class, Object and Relationship, as well as some advanced concepts such as Abstract Class, Polymorphism, Templates, Operator Overloading and Standard Template Library (Standard Template Library - STL). After implementation, tests were carried out that demonstrated the functionality of most of the required requirements. Finally, it can be concluded that the game allowed applying the knowledge acquired in the classroom and the development of the soft skill of both teamwork, communication and writing.*

• • •

Key-words or Key-expressions : *Article-Report Model for Work on Programming Techniques, Academic Work Focused on Implementation in C++, Internal Norms for Writing Work, Object-Oriented Programming*

INTRODUÇÃO

Nesse documento será apresentado o trabalho realizado para a matéria de Técnicas de Programação no período letivo de 2022/2. Dessa maneira, o desenvolvimento deste trabalho visa o aprofundamento da programação orientada a objetos, mas também introduzir aos alunos as práticas de indústria e a engenharia de *software.*

Dessa forma, para cumprir os objetivos propostos, foi desenvolvido um jogo de plataforma em C++, com auxílio da SFML (*Simple and Fast Multimedia Library*), como objeto de estudo. Por meio deste e dos requisitos funcionais apresentados na tabela 1, foi possível aplicar diversos dos conceitos apresentados na tabela 2.

A fim de desenvolver o jogo proposto, foi adotado um ciclo de desenvolvimento formado por , basicamente, 4 etapas: compreensão dos requisitos, modelagem, implementação em C++ e testes. Assim, esse ciclo foi repetido durante todo o prazo de desenvolvimento e permitiu alcançar resultados mais próximos do esperado.

Nas próximas seções serão apresentadas como é o jogo em si, detalhes sobre atendimento de requisitos, além de conceitos utilizados em C++, junto das suas justificativas. Por fim, esse artigo culmina em uma comparação entre o paradigma procedural com o orientado a objetos, e em uma conclusão com agradecimentos , divisão de trabalho e referências.

EXPLICAÇÃO DO JOGO EM SI

Primeiramente, ao abrir o jogo, o indivíduo pode escolher, utilizando o mouse, se pretende jogar ou sair. A seguir, ao clicar em jogar, o jogador pode escolher entre singleplayer e multiplayer, o que implica iniciar com 1 ou 2 jogadores. Por fim, um último menu é iniciado e permite escolher entre a fase 1 ou fase 2.

Assim, após escolher a fase, ela é iniciada e o jogador 1 pode controlar o personagem por meio das teclas WASD e o jogador 2, se escolhido o modo multiplayer, por meio das setas do teclado.

Figura 1. Fase 1 - Bosque

Figura 2. Fase 2 - Castelo

Após iniciar uma das fases, são criados inimigos e obstáculos em números aleatórios. Dessa forma, todos os inimigos possuem uma maneira de causar dano no jogador, sendo que o mago realiza isso por meio de um projétil, e, tanto o esqueleto quanto o cavaleiro, realizam esse ato ao encostar no jogador. Além disso, o obstáculo Lava causa dano no jogador caso pule sobre ela.

O objetivo final do jogo é matar todos os inimigos. Desse modo, caso o jogador conclua o objetivo na fase 1, a fase 2 é automaticamente iniciada.

Por fim, ao finalizar uma fase ou morrer, é aberto um menu que permite o jogador salvar sua pontuação.

DESENVOLVIMENTO DO JOGO NA VERSÃO ORIENTADA A OBJETOS

Desde a concepção do projeto a orientação a objetos norteou a construção do jogo, sendo assim, a construção começou pelo diagrama UML base fornecido pelo professor e evoluiu até o diagrama da figura 3. Dessa maneira, todo o jogo está repartido em classes, seguindo a orientação a objetos, fato que permite grande desacoplamento. Assim, a tabela 1 apresenta todos os requisitos funcionais cumpridos durante a execução do projeto.

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.	Requisito previsto inicialmente e cumprido	Requisito cumprido via classe Menu e seu respectivo objeto, o qual é agregado pela classe Jogo. Requisito cumprido com auxílio da SFML.
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 cumprido na totalidade	Requisito cumprido inclusive via classe Jogador cujos objetos são agregados em jogo, podendo ser apenas um jogador.

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 na totalidade	Requisito realizado nas classes Bosque, Castelo e Fase, sendo que a classe Fase é herdada por Bosque e Castelo.
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 projetil contra o(s) jogador(es) e um dos inimigos dever ser um 'Chefão'.	Requisito previsto inicialmente e cumprido na totalidade.	Requisito cumprido por meio da classe Inimigo, Mago, Esqueleto e Boss, sendo que Mago, Esqueleto e Boss herdam a classe base Inimigo e o Mago consegue lançar um projétil.
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 cumprido na totalidade.	Requisito cumprido na classe Bosque e Castelo, que criam quantidades aleatórias de inimigos, sendo no mínimo 3.
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 cumprido na totalidade.	Requisito cumprido por meio das classes Obstaculo, Plataforma, Caixa e Lava, sendo que Plataforma, Caixa e Lava herdam obstáculo e a Lava pode causar dano no jogador.
7	Ter em cada fase ao menos dois tipos de obstáculos com número aleatório de instâncias (*i.e.*, objetos), sendo pelo menos 3 instâncias por tipo.	Requisito previsto inicialmente e realizado.	Requisito cumprido na classe Bosque e Castelo, que instanciam quantidades aleatórias de Caixas e Lavas, sendo no mínimo 3 instancias de cada.
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 cumprido na totalidade.	Requisito cumprido na classe Plataforma, que permite Inimigos e Jogadores subirem sobre a mesma.
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 da gravidade no âmbito deste jogo de plataforma vertical e 2D.	Requisito previsto inicialmente e cumprido na totalidade	Requisito cumprido por meio da classe Gerenciador de Colisões e de métodos próprios das classes Personagens e Obstaculos.
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** Jogada.	Requisito previsto e **NÃO** realizado.	Requisito NÃO realizado.

Total de requisitos funcionais apropriadamente realizados. *(Cada tópico vale 10%, sendo que para ser contabilizado deve estar realizado efetivamente e não parcialmente)*	**90%** (setenta por cento).

A seguir, podem ser visualizadas todas as dependências entre classes presentes no jogo, a partir do diagrama UML final.

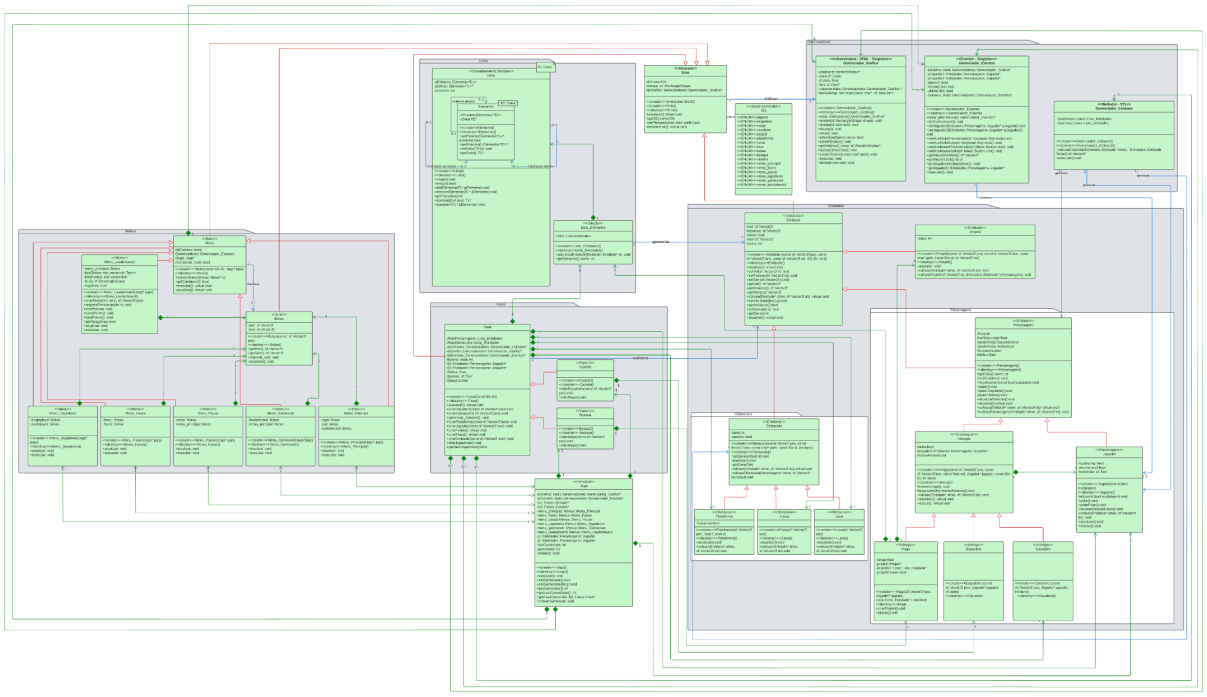

Figura 3. Diagrama Completo

Dessa forma, após analisar o diagrama, pode-se perceber as principais dependências entre classes no jogo.

Assim, percebe-se que uma das principais classes é a “Jogo”, que agrega todas as fases, gerenciadores, menus e jogador. Ademais, a classe Ente também possui grande importância, haja vista que praticamente todas as outras classes derivam dela.

Além disso, a classe Entidade também é muito relevante, pois todos os inimigos, obstáculos e jogador derivam dela. Sendo assim, ela reflete bem a programação orientada a objetos, haja vista que permite um desacoplamento fundamental para o projeto. Desse modo, o desacoplamento gerado por essa classe possibilita a criação de inimigos e obstáculos de forma simples e rápida.

Por fim, a classe “Gerenciador_Grafico” também demonstra a importância do desacoplamento na programação orientada a objetos, posto que possibilita separar a biblioteca gráfica do restante do jogo, de modo que, se for necessário trocar essa biblioteca, é preciso realizar modificações somente na classe “Gerenciador_Grafico” e não no código todo.

TABELA DE CONCEITOS UTILIZADOS E NÃO UTILIZADOS

Nessa seção, serão apresentados os conceitos utilizados durante a confecção do projeto. Desse modo, são apresentados os conceitos, o uso e onde tal conceito foi utilizado.

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 .h e .cpp. Como na classe Mago.
1.2	- Métodos (com retorno *const* e parâmetro *const*). & - Construtores (sem/com parâmetros) e destrutores	Sim	Na maioria dos .h e .cpp, como na classe Personagem .
1.3	- Classe Principal.	Sim	Main.cpp & Jogo.h/cpp.
1.4	- Divisão em .h e .cpp.	Sim	Em todas as classes, com exceção da classe Lista, que possui somente .h.
2	**Relações de:**		
2.1	- Associação direcional. & - Associação bidirecional.	Sim	Nas classes Menu e Jogo.
2.2	- Agregação via associação. & - Agregação propriamente dita.	Sim	Em alguns dos .h e .cpp, como na classe Mago, nos atributos Projetil e ListaObst.
2.3	- Herança elementar. & - Herança em diversos níveis.	Sim	Em alguns dos .h e .cpp, como nas classes Entidades, que herda Ente e na classe Mago, que herda classes em diversos niveis.
2.4	- Herança múltipla.	Não	Precisamente nos .h e .cpp, das classes C, D e F.
3	**Ponteiros, generalizações e exceções**		
3.1	- Operador *this* para fins de relacionamento bidirecional.	Sim	Precisamente nos .h e .cpp, da classe Jogo, em sua construtora.
3.2	- Alocação de memória (*new* & *delete*).	Sim	Em diversos .h e .cpp, como na destrutora da classe Gerenciador de Colisões e no método Criar Mago da classe Fase.
3.3	- Gabaritos/*Templates* criada/adaptados pelos autores (*e.g.*, Listas Encadeadas via *Templates*).	Sim	Especificamente no arquivo Lista.h.
3.4	- Uso de Tratamento de Exceções (*try catch*).	Sim	Especificamente nas classes Bosque e Castelo.
4	**Sobrecarga de:**		
4.1	- Construtoras e Métodos.	Sim	Especificamente no método setGameState da classe Jogo e na contrutora da classe Mago.
4.2	- Operadores (2 tipos de operadores pelo menos – Quais?).	Sim	Foi usado o operator[] e o operador= na ListaEntidades.cpp.
---	**Persistência de Objetos (via arquivo de texto ou binário)**		
4.3	- Persistência de Objetos.	Não	. . .
4.4	- Persistência de Relacionamento de Objetos.	Não	. . .
5	**Virtualidade:**		. . .
5.1	- Métodos Virtuais Usuais.	Sim	Especificamente no Inimigo.cpp
5.2	- Polimorfismo.	Sim	Nas classes Jogador e Inimigo especificamente no método atualizar.

	5.3	- Métodos Virtuais Puros / Classes Abstratas.	Sim	Na classe Entidade, especificamente no método atualizar.
	5.4	- Coesão/Desacoplamento efetiva e intensa com o apoio de padrões de projeto.	Não	Foi feito o uso de coesão e desacoplamento no gerenciador gráfico, porém sem apoio de padrões de projeto.
	6	**Organizadores e Estáticos**		
	6.1	- Espaço de Nomes (*Namespace*) criada pelos autores.	Sim	Especificamente na classe Mago.
	6.2	- Classes aninhadas (*Nested*) criada pelos autores.	Sim	Na Lista.h foi feito o aninhamento da classe Elemento.
	6.3	- Atributos estáticos e métodos estáticos.	Sim	Especificamente no Gerenciador_Gráfico.
	6.4	- Uso extensivo de constante (*const*) parâmetro, retorno, método...	Sim	Especificamente na Personagem.h.
	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	Para salvar o caminho das texturas foi utilizado const char* na classe Gerenciador_Gráfico. Vector foi utilizado para salvar o ranking dos jogadores, na classe Menu_Leaderboard.
	7.2	- Pilha, Fila, Bifila, Fila de Prioridade, Conjunto, Multi-Conjunto, Mapa **OU** Multi-Mapa.	Sim	Foi usado o Mapa na Classe Gerenciador Grafico.
	---	**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	Tratamento da movimentação e de Colisão na Gerenciador_Colisoes e na Gerenciador_Eventos.
	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).	Sim	Na classe Gerenciador_Eventos.
	---	**Interdisciplinaridades via utilização de Conceitos de Matemática Contínua e/ou Física.**		
	8.3	- Ensino Médio Efetivamente.	*Sim*	Torricelli para tratar a gravidade em diversos .cpp, como o Caixa.cpp.
	8.4	- Ensino Superior Efetivamente.	*Sim*	Efeito magnus para fazer o projétil no Projetil.cpp.
	9	**Engenharia de Software**		
	9.1	- Compreensão, melhoria e rastreabilidade de cumprimento de requisitos. &	Sim	Realizado no início e fim do projeto.
	9.2	- Diagrama de Classes em *UML*.	Sim	Atualizado durante todo o andamento do projeto.

	9.3	- Uso efetivo e intensivo de padrões de projeto *GOF*, *i.e.*, mais de 5 padrões.	Parcial	Foi usado somente o padrão de projeto Singleton.
	9.4	- Testes à luz da Tabela de Requisitos e do Diagrama de Classes.	Sim	Durante todo o projeto.
	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*	Por meio do github.
	10.2	- Reuniões com o professor para acompanhamento do andamento do projeto.	*Sim*	4 reuniões -3/11 -7/11 -17/11 -24/11
	10.3	- Reuniões com monitor da disciplina para acompanhamento do andamento do projeto.	*parcial*	*5 reuniões* *-28/09* *-28/10* *-28/10* *-4/11* *-16/11*
	10.4	- Revisão do trabalho escrito de outra equipe e vice-versa.	*Sim*	Vitor Errera e Thiago Seiji
	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%.)*			**80%** (oitenta porcento).

A seguir, está a tabela que explica a motivação de cada conceito utilizado na tabela 2.

Tabela 3. Lista de Justificativas para Conceitos Utilizados.

No.	Conceitos	***Listar apenas os utilizados*** **Situação**
1	**Elementares**	
1.1	- Classes, objetos. & - Atributos (privados), variáveis e constantes. & - Métodos (com e sem retorno).	Classe, Objetos, Atributos e Métodos foram utilizados porque são conceitos elementares na orientação a objetos.
1.2	- Métodos (com retorno *const* e parâmetro *const*). & - Construtores (sem/com parâmetros) e destrutores	A constância de atributos, métodos e afins, quando pertinente, evite mudanças equivocadas. Construtores são finalmente mandatórios para bem inicializar os atributos, enquanto que destrutores são usados para desalocar memória que não está mais em uso.
1.3	- Classe Principal.	Ter uma classe Principal permite se alcançar um melhor 'purismo' em termos de OO.
1.4	- Divisão em .h e .cpp.	Permite melhor organização das classes e afins que compõem o sistema.
2	**Relações**	
2.1	- Associação direcional. & - Associação bidirecional.	A associação foi utilizada porque determinadas classes necessitam de outras classes para funcionar da melhor forma.
2.2	- Agregação via associação. & - Agregação propriamente dita.	São casos diferentes de agregação, no primeiro um objeto pode existir sem a necessidade da existência do outro. No segundo caso, para que um objeto exista, outro é necessário.

2.3	- Herança elementar. & - Herança em diversos níveis.	Permite a existência de diversos tipos de uma determinada classe base.
3	**Ponteiros, generalizações e exceções**	
3.1	- Operador this para fins de relacionamento bidirecional	Ele aponta para o objeto para o qual a função de membro é chamada, permitindo a alteração desse objeto.
3.2	- Alocação de memória(new e delete)	O new permite criar um objeto e usá-lo fora da função criadora. Assim, junto com o new, é necessário usar o delete após o uso do objeto para liberar a memória alocada no new.
3.3	-Gabaritos/Templates criada/adaptados pelos autores (e.g., Listas Encadeadas via Templates)	Usado na lista encadeada criada, permite criar uma lista de diversos tipos.
3.4	-Uso e Tratamento de exceções (try catch)	Usado para tratar possíveis erros na abertura de arquivo das Fases.
4	**Sobrecarga de:**	
4.1	-Construtoras e Métodos	Permite que classes moldem seu funcionamento conforme a necessidade.
4.2	-Operadores (2 tipos de operadores pelo menos - Quais ?)	O operador [] permite navegar de forma mais simples na lista criada e o operador = permite adicionar elementos de forma mais simples.
5	**Virtualidade**	
5.1	-Métodos Virtuais Usuais	Permite que um método seja usado de diferentes formas em classes derivadas.
5.2	-Polimorfismo	Permite classes diferentes agirem de maneira diferente sobre determinadas circunstâncias. A aplicação mais clara é a sobrecarga.
5.3	-Métodos Virtuais Puros / Classes Abstratas	Garante que somente as classes derivadas instanciem a classe base.
6	**Organizadores Estáticos**	
6.1	-Espaço de nomes (namespace) criada pelos autores.	Facilita a organização de classes .
6.2	-Classes aninhadas (Nested) criada pelos autores	Permite o encapsulamento de classes, como da Lista e Elemento.
6.3	-Atributos Estáticos e métodos estáticos	Usado pelo padrão de projeto Singleton, permite que exista somente uma instância de determinada classe.
6.4	-Uso extensivo de constante (const) parâmetro, retorno, método…	Usado principalmente nos métodos Get…
7	**Standard Template Library (STL) e String))**	
7.2	-Pilha, Fila, Bifila, Fila de Prioridades, Conjunto, Multi-Conjunto, Mapa **OU** Multi-Mapa	Foi usado o mapa para organizar as texturas e ,assim, facilitar seu uso.
8	**Biblioteca Gráfica / Visual**	
8.1	-Funcionalidades Elementares . -Funcionalidades Avançadas como: -Tratamento de Colisões - duplo buffer	Necessário para renderizar, tratar movimento e colisão entre os objetos.

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)	O uso de gerenciador de eventos facilita o tratamento de entradas pelo teclado e , assim, torna mais fácil controlar a movimentação do jogadores.
–	**Interdisciplinaridades via utilização de conceitos de Matemática Contínua e/ou Física**	
8.3	-Ensino Médio Efetivamente	Necessário para aplicar a gravidade a todas as entidades
8.4	-Ensino Superior Efetivamente	Necessário para simular o efeito magnus no projétil e, assim, permitir que o mesmo seja lançado em linha reta.
9	**Engenharia de Software**	
9.1	-Compreensão, melhoria e rastreabilidade de cumprimento de requisitos	Realizado por meio de reuniões com professor e monitores, compõe toda a necessidade de entrega.
9.2	-Diagrama de Classes em UML	Permite a visualização do projeto como um todo.
9.4	-Testes à luz de Tabela de Requisitos e do diagrama de Classes	Necessário para corrigir os bugs.
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	Permite manter um histórico de modificações e facilita a colaboração de codigo.
10.2	Reuniões com o Professor para acompanhamento do projeto.	Essenciais para melhor entendimento dos requisitos.
10.4	-Revisão do trabalho escrito de outra equipe e vice-versa	Permite ajuda mútua para encontrar possíveis erros.

REFLEXÃO COMPARATIVA ENTRE DESENVOLVIMENTOS

Com base na experiência em programação procedural e orientada a objetos, é possível analisar que a essa possibilita expandir o código de forma mais simples, devido ao grande desacoplamento gerado. Dessa forma, em projetos que necessitam de constante atualização e que possuem conteúdo novo adicionado frequentemente, a utilização da programação orientada a objetos possui grande valor, pois permite realizar manutenção de forma rápida e simples, sem grandes alterações na base do código.

DISCUSSÃO E CONCLUSÕES

Após concluir o trabalho, foi notável o avanço do conhecimento em programação orientada a objetos, bem como a melhoria da comunicação e organização entre a dupla. Além disso, foi possível avançar nos conhecimentos sobre engenharia de software. Sendo assim,

pode-se afirmar que 90% dos requisitos funcionais foram cumpridos, bem como 70% dos conceituais.

CONSIDERAÇÕES PESSOAIS

No início do desenvolvimento o projeto e os requisitos assustaram os integrantes do grupo, entretanto, após entender as bases e o básico da biblioteca SFML, o projeto seguiu de forma mais tranquila.

DIVISÃO DO TRABALHO

Segue abaixo tabela com a divisão do trabalho em cada uma das etapas do desenvolvimento, na qual pode-se concluir que a divisão do trabalho foi realizada de forma equilibrada, não havendo nenhum tópico em que alguém da dupla ficou responsável por 100% do desenvolvimento do código.

Tabela 4. Lista de Atividades e Responsáveis.

Atividades.	**Responsáveis**
Compreensão de Requisitos	50% Ian e 50% Pedro
Diagramas de Classes	50% Ian e 50% Pedro
Programação em C++	50% Ian e 50% Pedro
Implementação de *Template*	30% Ian e 70% Pedro
Implementação de Colisão	70% Ian e 30% Pedro
Implementação de Obstáculos	50% Ian e 50% Pedro
Implementação de Entidades	50% Ian e 50% Pedro
Gerenciador Gráfico	30% Ian e 70% Pedro
Gerenciador de Eventos	30% Ian e 70% Pedro
Menus	30% Ian e 70% Pedro
Escrita do Trabalho	70% Ian e 30% Pedro
Revisão do Trabalho	70% Ian e 30% Pedro

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

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

AGRADECIMENTOS

Queremos agradecer especialmente os monitores, principalmente o Giovane pelos vídeos desenvolvidos.

REFERÊNCIAS UTILIZADAS NO DESENVOLVIMENTO

[A] SIMÃO, J. M. Site das Disciplina de Fundamentos de Programação 2, Curitiba – PR, Brasil, Acessado em 25/11/2022, às 23:32 - http://www.dainf.ct.utfpr.edu.br/~jeansimao/Fundamentos2/Fundamentos2.htm.

[B] SALVI, G. L. Canal do Youtube do Monitor Giovane, Curitiba, PR, Brasil, Acessado em 25/11/2022, às 23:30- https://www.youtube.com/@gege171