TRABALHO PARA A DISCIPLINA DE TÉCNICAS DE PROGRAMAÇÃO DO CURSO DE ENGENHARIA DE COMPUTAÇÃO DA UTFPR: EXPLORADORES ESPACIAIS

Letícia Walger Amaro, Bruno Kunnen Ledesma
leticiaamaro@alunos.utfpr.edu.br, brunokunnen@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 - A disciplina de Técnicas de Programação exige o desenvolvimento de um projeto de software, no formato de um jogo de plataforma, com o objetivo de potencializar o aprendizado de técnicas de engenharia de software, particularmente de programação orientada a objetos em C++. Neste trabalho, escolheu-se desenvolver o jogo Exploradores Espaciais, no qual onde o jogador é um astronauta enfrentando uma série de obstáculos e inimigos. O jogo possui duas fases, que diferem no nível de dificuldade, cada uma constando com diferentes tipos de inimigos e obstáculos. Para a realização do jogo foram considerados os requisitos e modelos propostos pelo orientador da matéria como base para elaborar a modelagem do software, utilizando como recurso o Diagrama de Classes em Linguagem de Modelagem Unificada a partir do software StarUML. Posteriormente, utilizando C++, realizou-se o desenvolvimento de acordo com os conceitos de orientação a objetos como Classe, Objeto, Relacionamento, em conjunto com conceitos avançados como Classe Abstrata, Polimorfismo, Gabaritos, Classes Aninhadas, Permanência de Dados através de Arquivos, Sobrecarga de Operadores, Biblioteca Padrão de Gabaritos (Standard Template Library - STL), e uso da biblioteca SFML (Simple and Fast Multimedia Library). Com a realização deste jogo foi possível cumprir o objetivo de aprendizagem pretendido.

Palavras-chave ou Expressões-chave: Projeto orientado a objetos em C++, Desenvolvimento de acordo a orientação a objetos, Diagrama de classes em linguagem de modelagem unificada.

Abstract - *The Programming Techniques subject requires the development of a software project, in the form of a platform game, with the objective of enhancing the learning of software engineering techniques, particularly object-oriented programming in C++. In this work, we chose to develop the game Exploradores Espaciais, in which the player is an astronaut facing a series of obstacles and enemies. The game has two stages, which differ in difficulty and each one has different types of enemies and obstacles. For the realization of the game, the requirements and models proposed by the subject's advisor were considered as a basis for elaborating the software modeling, using as a resource the Class Diagram in Unified Modeling Language from the StarUML software. Subsequently, using the C++ programming language, development was carried out according to object-oriented concepts such as Class, Object, Relationship, together with advanced concepts such as Abstract Class, Polymorphism, Templates, Nested Classes, Data Permanence through Files, Operator Overloading, Standard Template Library (Standard Template Library - STL), and use of SFML (Simple and Fast Multimedia Library). With the realization of this game it was possible to fulfill the intended learning objective.*

Key-words or Key-expressions Object oriented project in C++, Development according to object orientation, Class diagram in unified modeling language.

INTRODUÇÃO

Este projeto foi desenvolvido no contexto da disciplina de Técnicas de Programação da Universidade Tecnológica Federal do Paraná (UTFPR) como parte do conteúdo programático. A motivação deste projeto surge da necessidade de avaliar de forma prática os conceitos relativos à programação orientada a objetos que foram introduzidos em aula.

O objeto de estudo e da implementação do trabalho é um jogo de plataforma vertical e 2D que possa ser jogado por um jogador ou por dois jogadores concomitantemente. É importante ressaltar que o jogo deve ter um menu com representação gráfica e um sistema de salvar e recuperar dados, funcionalidades que também podem ser consideradas parte do objeto de estudo.

O método utilizado foi o ciclo de Engenharia de *Software*, porém de maneira mais simplificada. Uma vez informados e esclarecidos os requisitos, iniciou-se o desenho de um diagrama de classes (em linguagem UML) e, paralelamente, iniciou-se o desenvolvimento do código (linguagem C++) na plataforma Visual Studio Community Edition. Os testes do projeto foram feitos pelos desenvolvedores durante o desenvolvimento do código.

EXPLICAÇÃO DO JOGO EM SI

O jogo se chama “Exploradores Espaciais” pois o contexto do jogo se passa na lua onde o personagem é um astronauta que se depara com uma vida hostil e tem como objetivo eliminar inimigos evitando obstáculos em seu caminho, isso com a ajuda de esferas de poder que lhe oferecem invulnerabilidade e a capacidade de eliminar inimigos em seu caminho.

Na primeira fase da lua, o jogador encontra inimigos do tipo *Robô* e *Vilgax*, este último sendo um chefão, ambos têm a oportunidade de causar dano a curtas distâncias, com o tipo *Vilgax* causando mais dano que o tipo *Robô*; assim também temos os obstáculos do tipo *Espinhos* cujo dano é proporcional ao seu comprimento e o tipo *Rocha* que serve como um limitante ao movimento do jogador.

A segunda fase da lua apresenta novos inimigos do tipo *Alienígena* que podem atirar e, portanto, torná-los uma ameaça de longa distância, junto com obstáculos flutuantes do tipo *plataforma* onde o jogador pode encontrar *Esferas de Poder* que permitirão a vitória sobre os inimigos. Além dos novos inimigos e obstáculos, ainda estão presentes o tipo de inimigo *Robô* e o tipo de obstáculo *Rocha* da fase anterior.

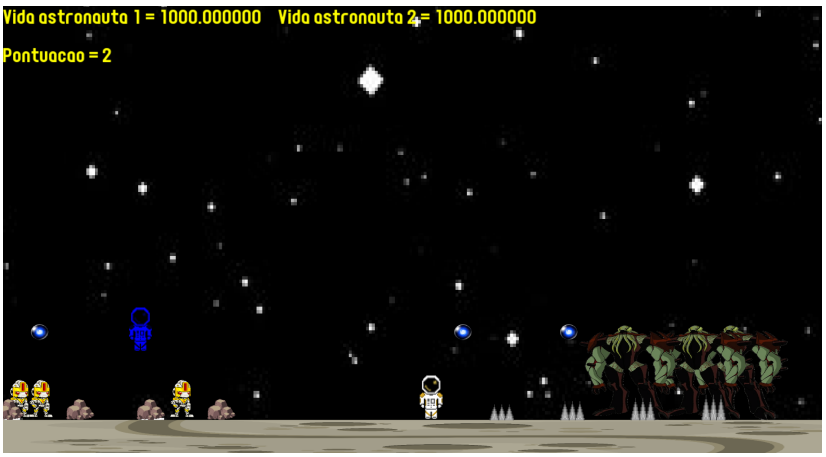

Figura 1. Primeira fase do jogo.

A Figura 1 mostra um jogador empoderado (em azul) após coletar uma *Esfera de Poder*. Os tipos *Robô* e *Rocha* estão à esquerda, enquanto *Vilgax* e Espinhos estão à direita.

Figura 2. Segunda fase do jogo.

A Figura 2 mostra um jogador (em vermelho) levando dano de projéteis disparados pelo tipo de inimigo *Alienígena*.

DESENVOLVIMENTO DO JOGO NA VERSÃO ORIENTADA A OBJETOS

A tabela abaixo apresenta os requisitos funcionais previstos inicialmente e onde foram implementados.

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 realizado.	Requisito cumprido via classe Menu e seu respectivo objeto, com suporte 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 realizado.	Requisito cumprido inclusive via classes Jogador1 e Jogador2, derivadas da classe abstrata Jogador, cujos objetos são inseridos em uma lista de entidades e uma lista de jogadores, podendo ser apenas um jogador, entretanto.

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 o jogo permite a escolha de fases através do menu, cada uma contendo seus próprios inimigos e obstáculos. Além disso, a fase 2 é carregada se a fase 1 for vencida, assim permitindo que elas sejam jogadas sequencialmente.
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 realizado.	Requisito totalmente realizado tendo os inimigos Robô, Alienígena (Inimigo que lança projétil) e Vilgax (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.	Requisito totalmente realizado, ambas as fases contêm a quantidade necessária de tipos e instâncias de inimigos.
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, sendo que os três tipos de obstáculos são Rocha, Plataforma e Espinho, este último causando dano em jogador se colidirem.
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 totalmente realizado, ambas as fases contêm a quantidade necessária de tipos e instâncias de obstáculos.
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.	Requisito cumprido com as fases possuindo os tipos de obstáculos necessários.
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.	Requisito cumprido tendo que os jogadores, inimigos e obstáculos sofrem o efeito da gravidade ao mesmo tempo que também podem sofrer colisão entre eles.
10	Permitir: (1) salvar nome do usuário, manter/salvar pontuação do jogador (incrementada via neutralização de inimigos) controlado pelo usuário e gerar	Requisito previsto inicialmente e realizado.	Requisito realizado inclusive via métodos que salvam a pontuação obtida durante o jogo e associam um nome ao final do jogo que depois mantém salvo em um arquivo

	lista de pontuação (*ranking*). **E** (2) Pausar e **Salvar** Jogada.		.txt. Salvar e pausar a jogada foi implementado nas fases.
Total de requisitos funcionais apropriadamente realizados.			**100%** (cem por cento).

O arquivo JogoTecProg.cpp, que contém a main, cria um objeto Jogo e chama seu método exec (executar), esse método habilita a tela principal do jogo e o loop através de um ponteiro para o *Gerente Gráfico* o qual está implementado como singleton, a partir daqui é chamada a máquina de estados que se encarrega de mostrar as opções do *Menu* e a partir daí o usuário pode escolher em qual fase jogar ou entrar na lista de ranking.

Ao longo da execução do jogo, os estados vão sendo gerenciados por uma *Máquina de Estados* que é implementada como um singleton.

Foi implementada uma interface gráfica simples a partir dos componentes da biblioteca SFML que também foi usado para as texturas de todo o projeto em geral.

Para generalizar a maioria dos objetos, uma classe abstrata chamada *Entidade* foi criada que determina atributos e métodos a serem utilizados pelas derivadas imediatas, que são as classes *Obstáculos* e *Personagem* sendo essas classes também abstratas.

A classe abstrata *Personagem* define os atributos e métodos utilizados por qualquer personagem do jogo, uma de suas derivadas é a classe *Jogador*, que é a classe que determina o comportamento dos jogadores que são classes próprias, sendo assim *Jogador1* e *Jogador2* cuja única diferença é o método de movimento. A outra derivada é a classe *Inimigo* que fornece os métodos de qualquer inimigo, principalmente determina os atributos referentes ao dano de cada inimigo e onde define seu movimento aleatório e suas derivadas são as classes *Robô*, *Alienígena* e *Vilgax*.

A classe abstrata *Obstáculo* define o comportamento base de um obstáculo, onde define se a colisão com os jogadores causa dano ou não, neste último caso apenas bloqueia o movimento. Suas derivadas são as classes *Espinho*, *Rocha* e *Plataforma*.

Uma vez selecionada a fase a ser jogada, a *Fase* é a classe abstrata base que se encarrega de inicializar os elementos comuns entre as fases. Tanto a primeira como a segunda fase são classes derivadas que tem agregado fortemente a *Lista de Entidades* para fazer a chamada do método executar de cada entidade, assim também como cada fase possui uma instância do *Gerente Colisões* que recebe os jogadores, inimigos, obstáculos e esferas de poder para então se encarregar de gerenciar as colisões entre eles.

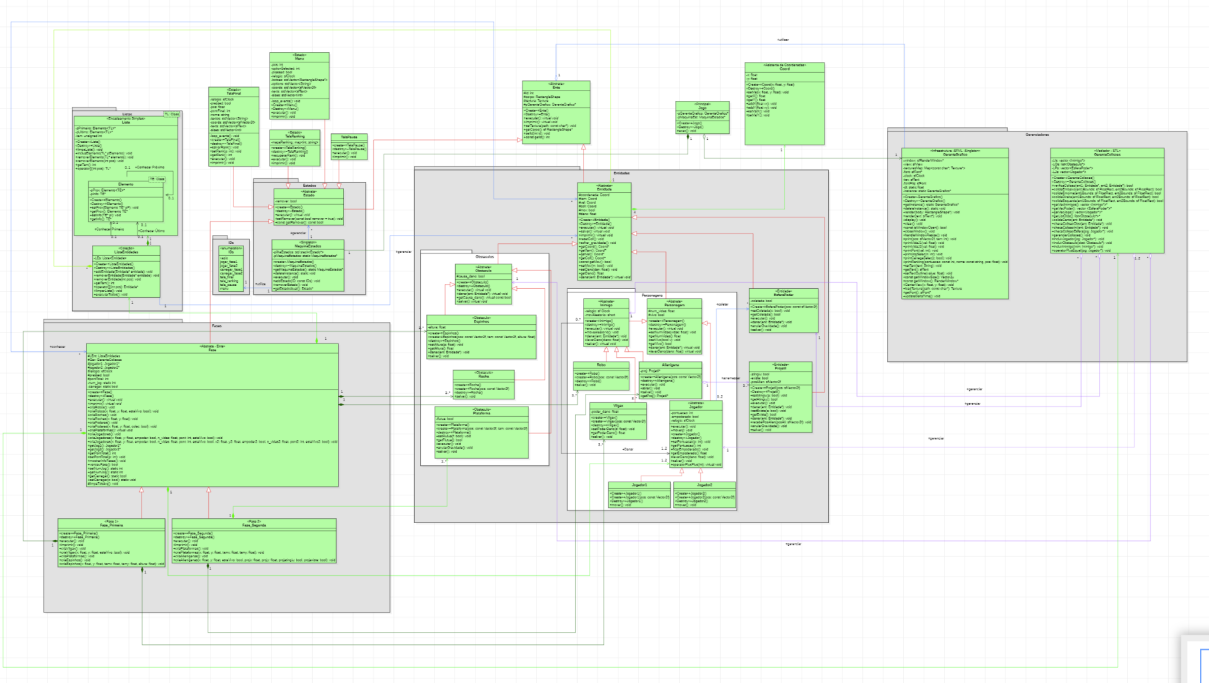

Figura 3. Diagrama de Classes UML

A Figura 3 mostra o diagrama de classes do projeto, sendo as classes em verde as que foram totalmente completas.

TABELA DE CONCEITOS UTILIZADOS E NÃO UTILIZADOS

A Tabela 2, apresentada a seguir, informa quais conceitos foram utilizados no projeto e onde foram implementados.

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 nas classes nos *namespaces* Gerenciadores e Entidades.
1.2	- Métodos (com retorno const e parâmetro const). & - Construtores (sem/com parâmetros) e destrutores	Sim	Na maioria dos .h e .cpp, como nas classes nos *namespaces* dos Gerenciadores (Gerenciador de Colisões e Gerenciador Grafico).
1.3	- Classe Principal.	Sim	Jogo.h/.cpp
1.4	- Divisão em .h e .cpp.	Sim	No desenvolvimento como um todo, como nas classes nos *namespaces* Entidades, Fases, Estados e Gerenciadores.
2	**Relações de:**		
2.1	- Associação direcional. & - Associação bidirecional.	Não	Somente usada associação direcional em vários dos .h e .cpp, como nas classes nos *namespaces* Personagens e Entidades.
2.2	- Agregação via associação. & - Agregação propriamente dita.	Sim	Em vários dos .h e .cpp, como nas classes nos *namespaces* Gerenciadores, Fases e Entidades.

	2.3	- Herança elementar. & - Herança em diversos níveis.	Sim	Em alguns dos .h e .cpp, como nas classes no *namespace* Entidades.
	2.4	- Herança múltipla.	Não	
	3	**Ponteiros, generalizações e exceções**		
	3.1	- Operador *this* para fins de relacionamento bidirecional.	Não	
	3.2	- Alocação de memória (*new* & *delete*).	Sim	Precisamente nas fases, ao criar jogadores, inimigos, obstáculos e poderes.
	3.3	- Gabaritos/*Templates* criada/adaptados pelos autores (*e.g.*, Listas Encadeadas via *Templates*).	Sim	Precisamente em Lista.h, nas classes Lista e Elemento. O código foi adaptado de "Elemento.h" e "Lista.h" do Grupo de Slides 10 - Parte A/B do sistema acadêmico.
	3.4	- Uso de Tratamento de Exceções (*try catch*).	Sim	Usado no Gerenciador Gráfico para lidar com falhas em carregar textura.
	4	**Sobrecarga de:**		
	4.1	- Construtoras e Métodos.	Sim	Diferentes construtoras como nos .h e .cpp de EsferaPoder, Espinhos, Plataforma, Robo, Rocha, Vilgax e Jogadores (1 e 2) que são utilizados em diferentes situações.
	4.2	- Operadores (2 tipos de operadores pelo menos – Quais?).	Sim	O operador ++ foi utilizado no jogador para aumentar a pontuação e o operador += foi utilizado no gerente de colisões para a inclusão de um novo jogador no vetor<jogador*>
	---	**Persistência de Objetos (via arquivo de texto ou binário)**		
	4.3	- Persistência de Objetos.	Sim	Persistência das entidades Inimigos, Obstáculos e Jogadores por meio de arquivos em formato .txt.
	4.4	- Persistência de Relacionamento de Objetos.	Sim	Persistência dos elementos das fases com seus respectivos relacionamentos a partir de uma recriação dos objetos a partir de arquivos em formato .txt.
	5	**Virtualidade:**		
	5.1	- Métodos Virtuais Usuais.	Sim	Precisamente o método imprimir() da classe Entidade, o método executar() da classe Inimigo, os métodos executar() e danar() das classes Obstaculo e Personagem.
	5.2	- Polimorfismo.	Sim	Se observa polimorfismo nas classes derivadas de Ente e Entidade.
	5.3	- Métodos Virtuais Puros / Classes Abstratas.	Sim	Precisamente os métodos executar() e imprimir() da classe Ente, os métodos executar() e danar() da classe Entidade, o método executar() da classe Fase, o método levarDano() da classe Personagem.
	5.4	- Coesão/Desacoplamento efetiva e intensa com o apoio de padrões de projeto.	Sim	Gerenciador Gráfico e Máquina de Estados implementados como Singleton.
	6	**Organizadores e Estáticos**		
	6.1	- Espaço de Nomes (*Namespace*) criada pelos autores.	Sim	Os seguintes *namespaces* foram criados: Listas, Fases, Estados, IDs, Entidades, Obstaculos, Personagens e Gerenciadores.
	6.2	- Classes aninhadas (*Nested*) criada pelos autores.	Sim	Precisamente em Lista.h, nas classes Lista e Elemento.

	6.3	- Atributos estáticos e métodos estáticos.	Sim	O método "getInstance" é estático e retorna "instance" que é um atributo estático (Gerenciador Grafico). O método "getMaquinaEstados" é estático e retorna "pMaquinaEstados" que é um atributo estático (MaquinaEstados). O método deleteInstance também é estático (Gerenciador Grafico e MaquinaEstados) Os atributos num_jog e carregar, também como seus respectivos métodos set e get na classe Fase são estáticos.
	6.4	- Uso extensivo de constante (*const*) parâmetro, retorno, método...	Sim	Nas classes Coord, Ente, Entidade, Espinhos, Estado, GerenteGrafico, MaquinaEstados, Menu, Robo, Rocha, Vilgax.
	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	Classe *String* usada no Gerenciador Gráfico para mostrar pontuação e número de vidas. *String* foi também usada nas classes Menu e TelaFinal para guardar o texto que é mostrado na tela. *Vector* e *List* utilizados em GerenteColisoes, Fase, Fase_Primeira, Fase_Segunda para guardar ponteiros de entidades, inimigos, obstáculos, esferas de poder e jogadores. *Vector* também foi utilizado em Menu e TelaFinal para guardar tamanho, coordenadas e conteúdo do texto que é impresso nessas telas.
	7.2	- Pilha, Fila, Bifila, Fila de Prioridade, Conjunto, Multi-Conjunto, Mapa **OU** Multi-Mapa.	Sim	Mapa está sendo utilizado no Gerenciador Gráfico para salvar as texturas que já foram renderizadas. Mapa também foi utilizado em TelaRanking para guardar e organizar a pontuação e o nome dos jogadores.
	---	**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	Funcionalidades elementares como a aplicação de texturas no projeto e avançadas como o tratamento de colisões no *GerenteColisões*.
	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 Matemática Contínua e/ou Física.**		

	8.3	- Ensino Médio Efetivamente.	Sim	Uso de vetores e cálculo da distância mínima mínima entre eles.
	8.4	- Ensino Superior Efetivamente.	Sim	Implementação de uma lista simplesmente encadeada. (Estrutura de Dados 1 - ED1)
	9	**Engenharia de Software**		
	9.1	- Compreensão, melhoria e rastreabilidade de cumprimento de requisitos. &	Sim	Mediante reuniões com o professor orientador e monitores.
	9.2	- Diagrama de Classes em *UML*.	Sim	Uso do software StarUML para fazer o Diagrama de Classes.
	9.3	- Uso efetivo e intensivo de padrões de projeto *GOF*, *i.e.*, mais de 5 padrões.	Não	-
	9.4	- Testes à luz da Tabela de Requisitos e do Diagrama de Classes.	Sim	Realizado pelos desenvolvedores do 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	Compartilhamento do código através do GitHub. Backups de versões antigas do código no computador.
	10.2	- Reuniões com o professor para acompanhamento do andamento do projeto.	Sim	1) 27/10/2022 2) 03/11/2022 3) 10/11/2022 4) 17/11/2022
	10.3	- Reuniões com monitor da disciplina para acompanhamento do andamento do projeto.	Sim	1) 21/10/2022 2) 31/10/2022 3) 04/11/2022 4) 07/11/2022 5) 10/11/2022 6) 17/11/2022 7) 21/11/2022 8) 24/11/2022 9) 25/11/2022 10) 26/11/2022
	10.4	- Revisão do trabalho escrito de outra equipe e vice-versa.	Não	-
	Total de conceitos apropriadamente utilizados.			**75%** (setenta e cinco por cento).

A Tabela 3, a seguir, informa os motivos de certos conceitos terem sido utilizados.

Tabela 3. Lista de Justificativas para Conceitos Utilizados.

No.	Conceitos	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, além de ajudar em desacoplar as classes do projeto em geral.
1.2	- Métodos (com retorno *const* e parâmetro *const*). & - Construtores (sem/com parâmetros) e destrutores	A constância de atributos, métodos e afins, quando pertinente, evitam mudanças equivocadas. Construtores são finalmente mandatórios para bem inicializar os atributos, em relação às destrutoras para despejar a memória que foi utilizada durante a execução do projeto de maneira apropriada.

1.3	- Classe Principal.	Ter uma classe Principal permite se alcançar um maior "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.	Associação direcional foi utilizada porque permite associar classes de forma versátil de acordo com a necessidade.
2.2	- Agregação via associação. & - Agregação propriamente dita.	Permite que os objetos sejam manipulados de formas diferentes dependendo da necessidade.
2.3	- Herança elementar. & - Herança em diversos níveis.	Permite que os objetos tenham um relacionamento forte e desacoplado.
3	**Ponteiros, generalizações e exceções**	
3.2	- Alocação de memória (*new* & *delete*).	Faz um programa mais completo que pode lidar com seu próprio uso de memória.
3.3	- Gabaritos/*Templates* criada/adaptados pelos autores (*e.g.*, Listas Encadeadas via *Templates*).	Oferecem versatilidade nas listas do código e na manipulação de diferentes classes.
3.4	- Uso de Tratamento de Exceções (*try catch*).	Ajuda no tratamento de possíveis erros críticos no projeto.
4	**Sobrecarga de:**	
4.1	- Construtoras e Métodos.	Eles ajudam a dar versatilidade ao código, já que algumas classes são instanciadas em diferentes contextos.
4.2	- Operadores (2 tipos de operadores pelo menos – Quais?).	Sobrecarga de operadores foi utilizada para criar uma interface mais limpa com os objetos.
4.3	- Persistência de Objetos.	Persistência de objetos foi utilizado para recuperar informações para que o usuário possa continuar um jogo anterior.
5	**Virtualidade:**	
5.1	- Métodos Virtuais Usuais.	Utilizado para permitir a redefinição de métodos genéricos entre as classes.
5.2	- Polimorfismo.	Utilizado na prática de código orientado a objetos.
5.3	- Métodos Virtuais Puros / Classes Abstratas.	Utilizado para formar classes base com métodos mais gerais.
5.4	- Coesão/Desacoplamento efetiva e intensa com o apoio de padrões de projeto.	Permite um código mais limpo com manipulação de atributos mais encapsulados.
6	**Organizadores e Estáticos**	
6.1	- Espaço de Nomes (Namespace) criada pelos autores.	Dá possibilidade de um código mais limpo e organizado.
6.2	- Classes aninhadas (Nested) criada pelos autores.	Permitem agrupar logicamente classes que serão usadas apenas em um lugar, aumentando o uso de encapsulamento e criando código mais compreensível e de fácil manutenção
6.3	- Atributos estáticos e métodos estáticos.	Permite a versatilidade de métodos e atributos que são utilizados por diversas classes.
6.4	- Uso extensivo de constante (const) parâmetro, retorno, método...	Ajuda a evitar modificações indesejadas de informações, uma forma de garantir que o método não produza efeitos colaterais no objeto.
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)	Classes String, Vector e List utilizado pela facilidade que oferece para a manipulação de objetos.
7.2	- Pilha, Fila, Bifila, Fila de Prioridade, Conjunto, Multi-Conjunto, Mapa **OU** Multi-Mapa.	Pilha utilizada na Máquina de Estados para gerenciar os diferentes Estados do jogo, Mapa para salvar as diferentes texturas utilizadas e Mapa para salvar a pontuação associada a um nome de jogador, porque ordena automaticamente as pontuações.
8	**Biblioteca Gráfica / Visual**	
8.1	-Funcionalidades Elementares. **&** - Funcionalidades Avançadas como: ● tratamento de colisões ● duplo *buffer*	Usado devido a necessidade de representar graficamente alguns objetos e também simular colisões entre os corpos das entidades.
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).	Foi utilizada a biblioteca gráfica SFML porque ela foi desenvolvida com foco em orientação à objetos; além de sua praticidade e alta eficiência.
8.3	- Ensino Médio Efetivamente.	Usado por conveniência com a situação do projeto.
8.4	-Ensino Superior Efetivamente.	Usado por conveniência com a situação do projeto.
9	**Engenharia de Software**	
9.1	- Compreensão, melhoria e rastreabilidade de cumprimento de requisitos. &	Ajuda no acompanhamento e conclusão do projeto.
9.2	- Diagrama de Classes em *UML*.	Utilizado para obter uma visão geral do projeto.
9.4	- Testes à luz da Tabela de Requisitos e do Diagrama de Classes.	Utilizado para verificar a integridade do código após modificações.
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).	Controle de versão de modelos e códigos na plataforma GitHub para ter todas as etapas do projeto documentadas e estar disponível para download em qualquer instante. Uso de backups no computador para ter controle de versões anteriores, caso fosse preciso retornar a alguma delas.
10.2	- Reuniões com o professor para acompanhamento do andamento do projeto.	Utilizado para demonstrar um acompanhamento do projeto com o orientador. Além disso, para receber orientações sobre que modificações deveriam ser feitas ao UML e ao código durante o desenvolvimento do projeto.
10.3	- Reuniões com monitor da disciplina para acompanhamento do andamento do projeto.	Utilizado para esclarecimento de conceitos e obtenção de orientações de como proceder no trabalho.

REFLEXÃO COMPARATIVA ENTRE DESENVOLVIMENTOS

Um projeto implementado na forma procedimental pode oferecer outros tipos de desafios em termos de gerenciamento de dados, o que pode resultar em maior complexidade para lidar com modificações e erros no código [1].

Por outro lado, na orientação a objetos, pode-se ter mais versatilidade na divisão de tarefas que cada classe fica a cargo, dividindo assim o projeto em "módulos" que talvez possam ser reaproveitados posteriormente.

DISCUSSÃO E CONCLUSÕES

O objetivo deste projeto foi aplicar os conceitos de orientação a objetos que foram aprendidos durante o curso tendo em conta os requisitos que foram previamente dados no início do trabalho, definindo assim um padrão de trabalho, além de simular um processo industrial para ganhar experiência que pode ser aplicada em um campo profissional.

O desenvolvimento deste projeto levou a um maior entendimento sobre os diversos conceitos de orientação a objetos, assim permitindo um aprofundamento acadêmico nos desenvolvedores. Além disso, a experiência de utilização da plataforma Github será útil futuramente, em um contexto profissional.

DIVISÃO DO TRABALHO

A Tabela 4 informa quais atividades foram realizadas por quais desenvolvedores.

Tabela 4. Lista de Atividades e Responsáveis.

Atividades.	**Responsáveis**
Compreensão de Requisitos	Letícia e Bruno
Diagramas de Classes	Letícia e Bruno
Programação em C++	Letícia e Bruno
Menu e Ranking	Mais Letícia que Bruno
Tela Pause e Tela Final	Letícia e Bruno
Implementação do segundo jogador	Bruno
Implementação das fases	Letícia e Bruno
Implementação do inimigo que lança projétil	Mais Letícia que Bruno
Implementação dos inimigos restantes e obstáculos	Letícia e Bruno
Implementação da gravidade	Mais Bruno que Letícia
Implementação de *Template*	Letícia
Implementação de classes aninhadas	Letícia
Implementação da Persistência dos Objetos	Mais Letícia que Bruno
Criação de *Namespaces*	Letícia e Bruno
Sobrecarga de operadores	Mais Bruno que Letícia
Classe GerenteGrafico	Bruno
Classe GerenteColisoes	Letícia e Bruno
Escrita do Trabalho	Leticia e Bruno
Revisão do Trabalho	Leticia e Bruno
Preparação do PowerPoint	Mais Bruno que Letícia

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

AGRADECIMENTOS

Agradecemos ao orientador da matéria de Técnicas de Programação Jean Marcelo Simão que orientou como proceder com o projeto. Também agradecemos aos monitores Thiago A., Giovane, Alexei e Thiago que ofereceram orientação com o projeto via Discord. Agradecendo também ao ex-monitor da matéria Técnicas de Programação Matheus Kunnen que estava disposto a ajudar a qualquer momento.

REFERÊNCIAS CITADAS NO TEXTO

[1] EducacionIT Programação Orientada a Objetos vs Programação Estruturada https://blog.educacionit.com/2018/05/21/programacion-orientada-a-objetos-vs-programacion-estructurada/

REFERÊNCIAS UTILIZADAS NO DESENVOLVIMENTO

[A] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Padrões de projeto. Editora ARTMED® EDITORA S.A. 2008. ISBN 978-85-7780-046-9

[B] Giovane Limas Salvi. GitHub. JogoPlataforma2D-Jungle. Disponível em: Giovanenero/JogoPlataforma2D-Jungle (github.com)

[C] Giovane Limas Salvi. Gege++. Disponível em: Gege++ - YouTube.

[D] Suraj Sharma. SFML C++ Tutorial 19 | Collision detection with screen and objects! (PART 3). Disponível em: SFML C++ Tutorial 19 | Collision detection with screen and objects! (PART 3) - YouTube

[E] Jean Marcelo Simão. Grupo de Slides 10 – Parte A. Linguagem C++ – Noção de Evento - Recursão - *Templates* ou Gabaritos. Disponível em: Microsoft PowerPoint - Fundamentos2-SlidesC++10A-2020-07-20 - Modo de Compatibilidade (utfpr.edu.br)

[F] Fabiano C. Domingos. Grupo de Slides 10 – Parte B – Lista desacopladas. Disponível em: Microsoft PowerPoint - Fundamentos2-SlidesC++10B-2020-07-20 - Modo de Compatibilidade (utfpr.edu.br)

[G] Freepik. Realistic stars galaxy background. Disponível em: Free Vector | Realistic stars galaxy background (freepik.com)

[H] CraftPix.net 2D Game Assets. Alien Character Sprite 2. Disponível em: Alien Character Sprite 2 | OpenGameArt.org

[I] Sdcwa. PNG Pixel Art Sprite Rocks. Disponível em: Painting Cartoon png download - 720*500 - Free Transparent Pixel Art png Download. - CleanPNG / KissPNG