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

Gabriel Henrique Linke, Pedro Sodré dos Santos
gabriellinke@alunos.utfpr.edu.br, psantos.2000@alunos.utfpr.edu.br

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

Resumo – A disciplina de Técnicas de Programação exige o desenvolvimento de um *software* de plataforma, no formato de um jogo, para fins de aprendizado de técnicas de engenharia de *software*, particularmente de programação orientada a objetos em C++. Para tal, neste trabalho, escolheu-se o jogo "Lemurya" no qual o jogador principal controla o personagem que representa o Rei Reptiliano. O objetivo do Rei Reptiliano é destruir o rei dos Idealistas (magos praticantes de magia negra). Para isso, ele precisa passar por diversos lugares de Lemurya, enquanto enfrenta os inimigos que são manipulados pelo rei dos Idealistas. O jogo tem três fases que se diferenciam por dificuldades para o jogador, sendo que na última fase o jogador encontra o chefão (rei dos Idealistas). Para o desenvolvimento do jogo foram considerados os requisitos textualmente propostos e elaborado modelagem (análise e projeto) usando como recurso o Diagrama de Classes em Linguagem de Modelagem Unificada (*Unified Modeling Language* - *UML*) usando como base um diagrama 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, Persistências de Objetos por Arquivos, Sobrecarga de Operadores e Biblioteca Padrão de Gabaritos (*Standard Template Library* - *STL*). Depois da implementação, os testes e uso do jogo feitos pelos próprios desenvolvedores demonstraram sua funcionalidade conforme os requisitos e o modelagem elaborado. Por fim, salienta-se que o desenvolvimento em questão permitiu cumprir o objetivo de aprendizado visado.

Palavras-chave ou Expressões-chave: Trabalho acadêmico voltado a implementação em C++, Desenvolvimento de jogo digital utilizando o Paradigma Orientado a Objetos.

Abstract – The subject of Técnicas de Programação demands the development of a platform software, in the form of a game, for purposes of learning software engineering techniques, especially object-oriented programming in C++. To this end, in this work, the game "Lemurya" was chosen, in which the main player controls the character that represents the Reptilian King. The main objective of the Reptilian King is to destroy the Idealists king (mages who practice black magic). For that, he needs to go through several places of Lemurya, while fighting against enemies that are manipulated by the Idealists king. The game has three stages that differ in difficulty for the player, being that in the last phase the player meets the boss (the Idealists king). For the development of the game the textually proposed requirements were considered and elaborated modeling (analysis and project) using as resource the Class Diagram in Unified Modeling Language, based on a proposed previous diagram. Subsequently, in C++ programing language, it was performed a development that took into account the usual concepts of object orientation such as Class, Object and Relationship, as well as some advanced concepts like Abstract Class, Polymorphism, Templates, file object persistence, operator overloading, and Standard Template Library – STL. After implementation, the developers' own testing and use of the game demonstrated its functionality as per the requirements and elaborate modeling. Finally, it is emphasized that the development in question made it possible to achieve the intended learning objective.

Key-words or Key-expressions: Academic work related to C++ implementation, Digital game development using Object Oriented Paradigm.

INTRODUÇÃO

Este é um trabalho de Técnicas de Programação que tem como objetivo ampliar a aplicação dos conceitos de programação orientada a objetos. Para isso, buscou-se utilizar ao máximo os conceitos de coesão e desacoplamento em conjunto com o uso de padrões de projeto, aprendidos em sala de aula

Este trabalho consiste na apresentação de um jogo de plataforma, incluindo inimigos e fases, implementado em C++, com a orientação do Professor Jean Marcelo Simão.

Para a devida realização do trabalho, foi utilizado o uso do ciclo clássico da engenharia de software, ou seja, primeiramente foi feita a compreensão dos requisitos (contidos na Tabela 1); a modelagem, feita através do diagrama de classes UML; a implementação feita em C++, com as IDEs do Visual Studio Community 2019 e CodeBlocks; assim como feito os testes para a verificação da ocorrência de erros na implementação.

Dado isso, será apresentado os devidos procedimentos, detalhadamente, de cada passo para a realização deste projeto, assim como a apresentação do jogo em si e dos devidos requisitos implementados.

EXPLICAÇÃO DO JOGO EM SI

Lemurya é um jogo de plataforma que se passa na cidade submersa de Lemurya, contendo 3 fases, podendo ser jogados no modo solo (1 jogador) ou cooperativo (2 jogadores). Ao iniciar o jogo, o usuário se depara com o menu principal, onde poderá escolher entre começar um novo jogo ou carregar um jogo já salvo. Nesse menu, também há opção de ranking, que pode ser utilizada para ver os 5 jogadores com maior pontuação e suas respectivas pontuações.

Figura 1. Imagem do Menu

Figura 2. Imagem do Ranking.

Após ser selecionada a opção de “New Game”, o usuário escolherá o número de jogadores (Solo – 1 jogador ou Coop – 2 jogadores), assim como a fase em que será jogada.

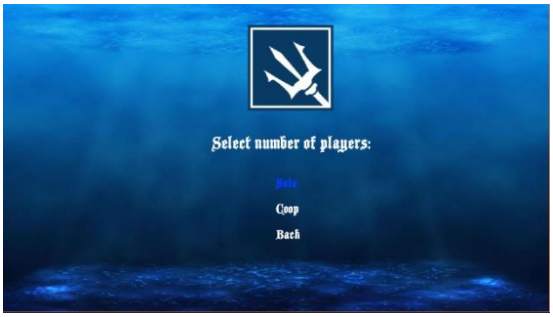


Figura 3. Seleção do número de jogadores.

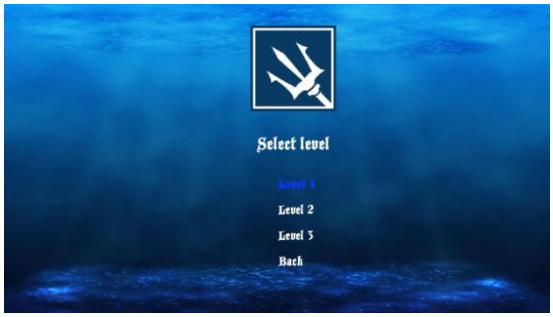


Figura 4. Seleção da fase.

Ao abrir a Fase 1 e 2, o jogador deverá chegar ao final dela, passando para a fase conseguinte. Na última fase (Fase 3), seu objetivo torna-se derrotar o chefão (Mago). Nessas fases, para atingir seu objetivo, o jogador controlará o Rei Reptiliano e terá de passar por diversos desafios, como superar inimigos e obstáculos.

Figura 5. Exemplo da 1° Fase

Figura 6. Exemplo de 3° Fase, com chefão no topo.

Ao decorrer da fase, o jogador pode utilizar as teclas 'WASD' para movimentar-se e a tecla 'E' para realizar seu ataque e derrotar inimigos (setas direcionais e 'Enter', respectivamente, no caso do 2° jogador). Ao neutralizar os inimigos, os jogadores recebem pontos, que podem ser visualizados no canto superior direito da tela (Score). No modo Coop a pontuação dos dois jogadores é conjunta.

Ao fim da partida, os pontos poderão aparecer no Ranking (Figura 2), após o jogador inserir seu nome na tela de final de jogo. Essa tela pode ser a de 'Game Over', caso o jogador tenha morrido, ou a de vitória, caso o jogador tenha vencido o jogo (derrotando o chefe na fase 3).

Figura 7. Tela de fim de jogo

No decorrer da partida, o jogador deverá se preocupar com sua barra de vida, a qual aparece no canto superior esquerdo da tela (figuras 5 e 6). A vida diminui gradativamente quando o jogador encosta em um dos inimigos. Se a vida do jogador chegar a 0, ou caso caia em um dos abismos, o jogador será derrotado, acarretando na tela de 'Game Over' (figura 7). No caso de dois jogadores, aparecerão duas barras de vida, e esta tela aparecerá quando qualquer um dos dois jogadores for neutralizado.

Figura 8. Exemplo de jogo com 2 jogadores.

No meio do jogo, também há opção de pause, que abre quando o jogador aperta a tecla ‘Esc’, ou quando a janela do jogo perde o foco. Nela, há as opções de voltar ao jogo, salvar o progresso atual, retornar ao menu principal e fechar o jogo.

Figura 9. Menu Pause.

DESENVOLVIMENTO DO JOGO NA VERSÃO ORIENTADA A OBJETOS

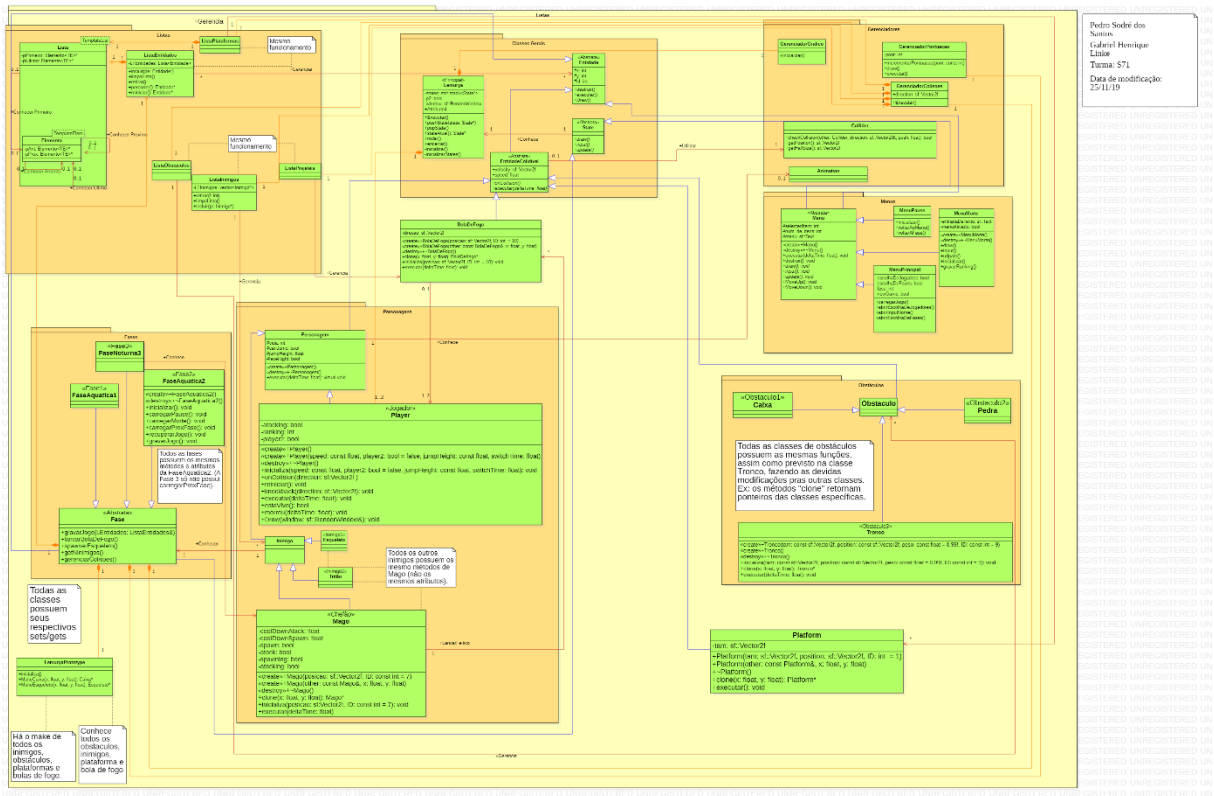
Tabela 1. Lista de Requisitos do Jogo e suas Situações.

N.	Requisitos Funcionais	Situação	Implementação
1	Apresentar menu de opções aos usuários do Jogo.	Requisito previsto inicialmente e realizado.	Requisito cumprido via classe Menu (abstrata) e os respectivos objetos de suas subclasses (Menu Principal, Menu Pause e Menu Morte).
2	Permitir um ou dois jogadores aos usuários do Jogo, sendo que no último caso seria para que os dois joguem de maneira concomitante.	Requisito previsto inicialmente e realizado.	Requisito cumprido inclusive via classe Player cujos objetos são agregados em jogo, podendo ser um ou dois efetivamente.
3	Disponibilizar ao menos duas fases que podem ser jogadas sequencialmente ou selecionadas.	Requisito previsto inicialmente e realizado.	Requisito cumprido, inclusive com 3 fases, sendo através de objetos das classes (FaseAquatica1, FaseAquatica2 e FaseNoturna3).
4	Ter três tipos distintos de inimigos (o que pode incluir ‘Chefão’, vide abaixo), sendo que pelo menos um	Requisito previsto inicialmente e realizado.	Requisito cumprido, criando objetos das classes, Tritão, Esqueleto e Mago, sendo este último, o chefão capaz de

	dos inimigos deve ser capaz de lançar projetil contra o(s) jogador(es).		lançar bolas de fogo (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 5 instâncias por tipo.	Requisito previsto inicialmente e realizado.	Requisito cumprido inclusive via método gerarInimigos, definido na classe abstrata Fase e redefinido nas suas classes derivadas.
6	Ter inimigo "Chefão" na última fase	Requisito previsto inicialmente e realizado.	Requisito cumprido inclusive via objeto da classe Mago que é instanciado na última fase (FaseNoturna3).
7	Ter três tipos de obstáculos.	Requisito previsto inicialmente e realizado.	Requisito cumprido inclusive via classes Tronco, Pedra e Caixa cujos objetos são agregados na fase através da classe LemuryaPrototype.
8	Ter em cada fase ao menos dois tipos de obstáculos com número aleatório de instâncias (i.e., objetos) sendo pelo menos 5 instâncias por tipo.	Requisito previsto inicialmente e realizado.	Requisito cumprido inclusive via método gerarObstaculos definido na classe Fase e redefinido nas suas classes derivadas.
9	Ter representação gráfica de cada instância.	Requisito previsto inicialmente e realizado.	Requisito cumprido inclusive com a utilização da biblioteca gráfica SFML e via classe GerenciadorGrafico.
10	Ter em cada fase um cenário de jogo com os obstáculos.	Requisito previsto inicialmente e realizado.	Requisito cumprido via classe abstrata Fase e suas derivadas, com seus obstáculos sendo agregados através dos métodos gerarObstaculos, novoJogo e recuperarJogo.
11	Gerenciar colisões entre jogador e inimigos, bem como seus projeteis (em havendo).	Requisito previsto inicialmente e realizado.	Requisito cumprido inclusive via classes Collider, que verifica a colisão, e o Gerenciador de Colisões, que verifica as colisões percorrendo todas as listas de entidades que são colidíveis.
12	Gerenciar colisões entre jogador e obstáculos.	Requisito previsto inicialmente e realizado.	Idem para o requisito acima.
13	Permitir cadastrar/salvar dados do usuário, manter pontuação durante jogo, salvar pontuação e gerar lista de pontuação (*ranking*).	Requisito previsto inicialmente e realizado.	Requisito cumprido inclusive via classes GerenciadorDePontuação, atributo ranking do jogador, método do menu principal que gera a lista de pontuação.

14	Permitir Pausar o Jogo	Requisito previsto inicialmente e realizado	Requisito cumprido inclusive via classe MenuPause, que é instanciada caso o usuário pressione a tecla ESC.
15	Permitir Salvar Jogada.	Requisito previsto e realizado	Requisito realizado via classe MenuPause, e método checkSalvarJogo das classes derivadas de Fase, que permite salvar a jogada.

Principais Classes e suas relações em UML



A principal classe do UML é a classe que dá o nome ao jogo (Lemurya). Por meio de seu objeto, o jogo é controlado através de uma pilha de states (State é outra classe do jogo), esta que é um stack da STL e um dos atributos da classe Principal. Nesse sentido, a classe State conhece o jogo, por meio disso são chamados os métodos 'Push' e 'Pop' da classe principal, fazendo a transição entre menus e fases do jogo.

Há também o pacote de listas, que são implementas por duas formas, ou por lista template, que é o caso de 'listaEntidades' e 'listaPlataformas', ou por vector da STL, que é o caso das listas 'listaInimigos', 'listaProjeteis' e 'listaObstaculos'.

No pacote de gerenciadores, há a classe gerenciador de colisões, através dele que é feito a verificação de todas as colisões do jogo, percorrendo as listas. Em fase, é chamado apenas o método 'executar()' do 'GerenciadorDeColisoes'. Também é importante lembrar que o cálculo da colisão é feito pela classe 'Collider', o gerenciador apenas percorre as listas.

Na classe 'LemuryaPrototype', há ponteiros para inimigos, obstáculos, bolas de fogo e plataformas, ela é uma classe que é instanciada na fase e é por meio dela que são criados os

elementos do jogo, chamando o método ‘Make()’ , que chama o método ‘clone()’ de cada classe associada.

TABELA DE CONCEITOS UTILIZADOS E NÃO UTILIZADOS

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

N.	Conceitos	Uso	Onde / O quê
1	**Elementares:**		
	- Classes, objetos. & - Atributos (privados), variáveis e constantes. & - Métodos (com e sem retorno).	Sim	Todos .h e .cpp
	- Métodos (com retorno *const* e parâmetro *const*). & - Construtores (sem/com parâmetros) e destrutores	Sim	Todos .h e .cpp
	- Classe Principal.	Sim	main.cpp & Lemurya.h/.cpp
	- Divisão em .h e .cpp.	Sim	No desenvolvimento como um todo.
2	**Relações de:**		
	- Associação direcional. & - Associação bidirecional.	Sim	Bidirecional entre FaseNoturna3 e Mago Direcional entre BolaDeFogo e Player.
	- Agregação via associação. & - Agregação propriamente dita.	Sim	Agregação via associação entre Lista de Entidades e Lista. Agregação propriamente dita entre GerenciadorDeColisoes e ListaPlataformas, por exemplo.
	- Herança elementar. & - Herança em diversos níveis.	Sim	Elementar em EntidadeColidivel e Entidade. Em diversos níveis entre Mago e Entidade, por exemplo.
	- Herança múltipla.	Sim	Fase herdando de Entidade e State, por exemplo.
3	**Ponteiros, generalizações e exceções**		
	- Operador *this*.	Sim	Em diversos momentos. Ao inicializar o prototype dentro de fase, por exemplo.
	- Alocação de memória (*new* & *delete*).	Sim	Em diversos momentos. Delete em destrutoras e new nos métodos de carregar um novo estado do jogo, como no carregarPause de fase.
	- Gabaritos/*Templates* criada/adaptados pelos autores (e.g. Listas Encadeadas via *Templates*).	Sim	Classe List.
	- Uso de Tratamento de Exceções (*try catch*).	Sim	Classe GerenciadorGrafico.
4	**Sobrecarga de:**		

	- Construtoras e Métodos.	Sim	Construtoras nos objetos que podem ser clonados. Métodos em FaseAquatica1, por exemplo.
	- Operadores (2 tipos de operadores pelo menos).	Não	-
	Persistência de Objetos (via arquivo de texto ou binário)		
	- Persistência de Objetos.	Sim	Via métodos de gravarJogo e recuperarJogo das fases.
	- Persistência de Relacionamento de Objetos.	Sim	Persistência do relacionamento dos inimigos com os players, por exemplo
5	**Virtualidade:**		
	- Métodos Virtuais.	Sim	Na classe Fase, por exemplo
	- Polimorfismo	Sim	Chamadas de update e Draw da classe Entidades, por exemplo.
	- Métodos Virtuais Puros / Classes Abstratas	Sim	Classes menu e fase, por exemplo.
	- Coesão e Desacoplamento	Sim	No projeto como um todo
6	**Organizadores e Estáticos**		
	- Espaço de Nomes (*Namespace*) criada pelos autores.	Não	-
	- Classes aninhadas (*Nested*) criada pelos autores.	Sim	Classes List e Entidade
	- Atributos estáticos e métodos estáticos.	Parcialmente	Atributo estático VIEW_HEIGHT em Lemurya
	- Uso extensivo de constante (*const*) parâmetro, retorno, método...	Sim	No projeto como um todo
7	**Standard Template Library *(STL)* e String OO**		
	- 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	Em GerenciadorDePontuacao e ListaObstaculos, por exemplo.
	- Pilha, Fila, Bifila, Fila de Prioridade, Conjunto, Multi-Conjunto, Mapa **OU** Multi-Mapa.	Sim	Em Lemurya (Pilha) e setRanking em MenuPrincipal (Multimap).
	Programação concorrente		
	- *Threads* (Linhas de Execução) no âmbito da Orientação a Objetos, utilizando Posix, C-Run-Time **OU** Win32API ou afins.	Não	-
	- *Threads* (Linhas de Execução) no âmbito da Orientação a Objetos com uso de Mutex, Semáforos, **OU** Troca de mensagens.	Não	-
8	**Biblioteca Gráfica / Visual**		
	- Funcionalidades Elementares. & - Funcionalidades Avançadas como: • tratamento de colisões • duplo *buffer*	Sim	Representações gráficas e escrever texto através da biblioteca SFML.
	- Programação orientada e evento em algum ambiente gráfico. **OU** - *RAD – Rapid Application Development* (Objetos gráficos como formulários, botões etc).	Sim	Utilização dos Eventos da biblioteca SFML na implementação dos menus.

	Interdisciplinaridades por meio da utilização de Conceitos de Matemática e/ou Física.		
	- Ensino Médio.	Sim	Conceitos de matemática e física básica, como velocidade, por exemplo.
	- Ensino Superior.	Sim	Norma de vetores, por exemplo.
9	**Engenharia de *Software***		
	- Compreensão, melhoria e rastreabilidade de cumprimento de requisitos. &	Sim	Durante reuniões com o professor e monitor.
	- Diagrama de Classes em *UML*.	Sim	No desenvolvimento como um todo.
	- Uso efetivo (quiçá) intensivo de padrões de projeto (particularmente GOF).	Sim	Classes LemuryaPrototype(Prototype)State(State) MenuPrincipal(Iterator)
	- Testes a luz da Tabela de Requisitos e do Diagrama de Classes.	Sim	Durante reuniões com o professor e monitor.
10	**Execução de Projeto**		
	- Controle de versão de modelos e códigos automatizado (via SVN e/ou afins) **OU** manual (via cópias manuais). & - Uso de alguma forma de cópia de segurança (backup).	Sim	Via GitHub e cópias manuais
	- Reuniões com o professor para acompanhamento do andamento do projeto.	Sim	3 Reuniões (25/10, 12/11, 22/11).
	- Reuniões com monitor da disciplina para acompanhamento do andamento do projeto.	Sim	8 Reuniões (14/10, 16/10, 18/10, 25/10, 6/11, 13/11, 18/11, 20/11).
	- Revisão do trabalho escrito de outra equipe e vice-versa.	Sim	Gustavo Brunholi Chierici e Jhonny Kristyan

Tabela 3. Lista de Justificativas para Conceitos Utilizados e **Não** Utilizados no Trabalho.

No.	**Conceitos**	***Listar apenas os utilizados* Situação**
1	**Elementares**	Os conceitos elementares como os de classe, objetos, atributos, métodos, entre outros, foram utilizados por conta de serem conceitos fundamentais da Programação Orientada a Objetos.
2	**Relações**	As relações de associação, agregação e herança foram utilizadas por serem partes fundamentais em um projeto desenvolvido com Programação Orientada a Objetos.
3	**Ponteiros, generalizações e exceções**	O operador this foi utilizado em alguns momentos em que era necessário retornar o ponteiro do objeto atual. A alocação de memória foi utilizada constantemente nos momentos em que se fez necessária a criação de objetos em tempo de execução. Templates foram utilizados para aumentar a reusabilidade do código. O tratamento de exceções (try catch) foi utilizado para impedir que o programa fosse inicializado caso houvesse a falta de algum arquivo essencial para o seu funcionamento.

4	**Sobrecargas e persistência de objetos**	A sobrecarga de construtoras e métodos foi utilizada para possibilitar a redefinição dos mesmos, para que assim, seja possível utilizá-los em diferentes situações. A sobrecarga de operadores não foi utilizada porque não foi possível encontrar uma situação em que ela fosse essencial. A persistência de objetos e relações foi utilizada para possibilitar salvar a atual versão do programa e posteriormente voltar-se para a última versão salva.
5	**Virtualidade**	Métodos virtuais, classes abstratas e polimorfismo foram utilizados, pois, são recursos que podem facilitar muito a Programação Orientada a Objetos. Coesão e desacoplamento foram utilizados pois são de suma importância na criação de um projeto.
6	**Organizador e Estáticos**	Foi realizado o uso extensivo de const para respeitar-se o princípio do menor privilégio. Classes aninhadas foram utilizados pois elas fazem com que uma classe não possa ser instanciada sem estar vinculada a uma classe de ordem superior. Atributos estáticos foram utilizados para valores que deveriam ser compartilhados por todas as instâncias de uma classe. Métodos estáticos não foram utilizados pois não foi encontrada uma forma de inseri-los no projeto. Namespace não foi utilizado pois a sua utilização só foi cogitada ao fim do projeto, quando o tempo para a finalização já era escasso.
7	**Componentes STL e programação concorrente**	Os componentes da STL foram utilizados para acelerar o desenvolvimento do projeto, pois são componentes prontos que já foram testados e otimizados, tendo baixíssimo risco de conterem algum erro de implementação. Programação concorrente não foi utilizada por conta de ser cogitada apenas próximo ao fim do projeto, momento em que o tempo para a implementação era escasso.
8	**Uso de biblioteca gráfica e interdisciplinaridade**	A matemática e a física foram essenciais para o desenvolvimento, pois estavam presentes no projeto como um todo. O uso da biblioteca gráfica SFML foi de suma importância para facilitar a implementação da parte gráfica do programa.
9	**Engenharia de *Software***	O diagrama de classes foi utilizado majoritariamente para a projeção do *software*. A partir do diagrama foi possível analisar como estava a implementação do projeto, e verificar se ela estava cumprindo os requisitos. Padrões de projeto foram utilizados para melhorar a coesão e desacoplamento, assim, permitindo que o código se tornasse mais flexível e reutilizável.
10	**Execução de projeto**	O controle de versões foi utilizado para manter um histórico de todas as versões feitas no projeto, e para que houvesse uma cópia de segurança caso necessário. As reuniões com o professor e com o monitor da

		disciplina foram realizadas para que ambos pudessem opinar a respeito do projeto, dando dicas para a sua melhoria, e para que pudessem acompanhar o seu desenvolvimento. A revisão foi realizada para garantir a qualidade do trabalho.

REFLEXÃO COMPARATIVA ENTRE DESENVOLVIMENTOS

Ambos paradigmas (procedimental e orientado a objetos) tem seus pontos fortes e seus pontos fracos, e por isso, antes de desenvolver um *software*, devesse refletir sobre qual dos paradigmas pode ser melhor aplicado.

O desenvolvimento procedimental pode ser uma melhor opção, por exemplo, quando o código a ser implementado tem uma menor complexidade.

Por outro lado, o desenvolvimento utilizando a Programação Orientada a Objetos pode ser uma boa opção ao implementar-se um *software* maior e mais complexo, pois os recursos oferecidos, como relações entre objetos e o polimorfismo, facilitam diversas funções, além de facilitar entendimento do sistema no geral.

DISCUSSÃO E CONCLUSÕES

O desenvolvimento deste projeto certamente acarretou em um grande desenvolvimento pessoal de ambos discentes responsáveis pela sua confecção. Através dele foi possível perceber a importância do levantamento de requisitos e da análise e projeto de um *software,* antes de inicializar a implementação propriamente dita.

Além disso, o projeto também fez com que os envolvidos pudessem praticar a Programação Orientada a Objetos, conceito que recentemente havia sido introduzido. Ademais, os desenvolvedores também puderam sentir pela primeira vez a sensação de estar fazendo parte de um projeto dessa complexidade e aplicar os conceitos de engenharia.

CONSIDERAÇÕES PESSOAIS

Os discentes acreditam que a limitação de 12 páginas foi um empecilho no desenvolvimento deste documento, visto o tamanho do projeto que nele foi relatado.

DIVISÃO DO TRABALHO

Tabela 4. Lista de Atividades e Responsáveis.

Atividades.	**Responsáveis**
Levantamento de Requisitos	Pedro e Gabriel
Diagramas de Classes	Pedro
Programação em C++	Gabriel e Pedro em geral
Implementação de *Template*	Gabriel
Implementação da Persistência dos Objetos...	Gabriel
Implementação de State	Pedro
Implementação de Menus	Mais Pedro que Gabriel
Implementação de Prototype	Pedro e Gabriel
Implementação de Ranking	Pedro e Gabriel

Implementação de Projétil	Gabriel
Implementação de Fases	Mais Gabriel que Pedro
Implementação de Colisões	Gabriel
Implementação de Listas	Mais Gabriel que Pedro
Implementação do 2° Jogador	Gabriel
Escrita do Trabalho	Pedro e Gabriel
Revisão do Trabalho	Pedro e Gabriel

AGRADECIMENTOS

Agradecimentos a Gustavo Brunholi Chierici e Jhonny Kristyan pela revisão do trabalho, a Felipe Alves pela assistência durante o desenvolvimento do projeto, e ao Professor Dr. Jean M. Simão pelas aulas ministradas na disciplina de Técnicas de Programação, que possibilitaram a realização do projeto, e também pela assistência e supervisão durante o desenvolvimento deste.

REFERÊNCIAS CITADAS NO TEXTO

[1] DEITEL, H. M.; DEITEL, P. J. C++ Como Programar. 5ª Edição. Bookman. 2006.
[2] SIMÃO, J. M. Site das Disciplina de Fundamentos de Programação 2, Curitiba – PR, Brasil, Acessado em 24/11/2019, às 14:35 http://www.dainf.ct.utfpr.edu.br/~jeansimao/Fundamentos2/Fundamentos2.htm.

REFERÊNCIAS UTILIZADAS NO DESENVOLVIMENTO

[1] SIMÃO, J. M. Site das Disciplina de Fundamentos de Programação 2, Curitiba – PR, Brasil, Acessado em 24/11/2019, às 14:35 http://www.dainf.ct.utfpr.edu.br/~jeansimao/Fundamentos2/Fundamentos2.htm.