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

Henrique Romaniuk Ramalho, Felipe Augusto Lee
henriqueramalho@alunos.utfpr.edu.br, flee@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, parte da matriz de estudos do curso de Engenharia de Computação da Universidade Tecnológica Federal do Paraná, exige dos alunos como forma de avaliação, a confecção de um jogo de plataforma feito em C++ voltado à prática dos conceitos de Paradigma de Orientação à Objetos. Para solidificar tal aprendizagem, foi criado o jogo Magic Golens no qual o jogador deve enfrentar obstáculos e inimigos em diferentes cenários, os quais mudam a cada fase, com o objetivo de passar para a próxima. O jogo é composto de três fases, cada uma com um nível de dificuldade crescente e em um ambiente diferente do anterior. Para o bom desenvolvimento do programa, foi-se elaborado um Diagrama de Classes seguindo a Linguagem de Modelagem Unificada (*Unified Modeling Language*) com o intuito de mapear os requisitos textualmente propostos, usando como base um diagrama arquitetural previamente posto. Então, usufruindo dos conceitos de Orientação à Objetos oferecido pela linguagem C++, o desenvolvimento do jogo seu deu pelo uso de , não somente Classes, Objetos e Relacionamentos, mas também Classes Abstratas, Herança, Polimorfismo, entre outros. Em seguida foram realizados testes pelos próprios desenvolvedores, verificando seu êxito mediante funcionalidade dos requisitos.

Palavras-chave ou Expressões-chave: Trabalho acadêmico focado na implementação C ++, Projeto universitário de técnicas de programação, Um uso de programação orientada a objetos, Um jogo desenvolvido em C ++, Magic Golens, Um projeto baseado em diagrama de classes.

Abstract – *This document shows general information about an academic project, by the means of the subject "Técnicas de Programação", developed in order settle the knowledge acquired in class about programming in C++ focused on Object Oriented concepts. The work, a game based on plataforms, was done by elaborating a Class Diagram in the Universal Modeling Language in order to map the requirements. Afterwards, by using Object Oriented concepts (eg: Inheritance, Abstract Classes…), the game was then programmed and subsequently tested by the developers themselves verifying the functionality of the requirements. In conclusion, the accomplishment of this project was extremely beneficial and constructive in terms of learning and professional training.*

Key-words or Key-expressions: Academic Work Focused on C++ Implementation, University Programming Techniques Project, Object Oriented Programming, Game developed in C++, Magic Golens, Project based on Class Diagram.

INTRODUÇÃO

Durante a jornada de um universitário de Bacharelado em Engenharia da Computação pela matriz curricular típica, ao cursar a matéria de Técnicas de Programação do DAINF/UTFPR, lecionada pelo docente Prof. J. M. Simão, aos estudantes é requisitado o desenvolvimento de um software, em duplas randômicas, em formato de um jogo de plataformas. Tal programa deveria ser feito em C++ baseado em Orientação à Objetos. Foi também encorajado, aos alunos, a usufruir de uma biblioteca gráfica, tal como Allegro e SFML, para a realização do projeto. A avaliação do trabalho seria feita pelo professor responsável,

baseando-se no cumprimento de requisitos textualmente postos previamente, resultando em 50% da nota final do aluno.

O objetivo primeiro da realização deste trabalho é a formação e solidificação de habilidades e conhecimentos adquiridos durantes as aulas, vide programação em C++ baseada no Paradigma de Orientação à Objetos, organização e planejamento de projeto usando-se Linguagem de Modelagem Unificada (*UML – Unified Modeling Language*), et cetera. Foi relevado a importância da prática de programação para aprender seus padrões e fundamentos, o valor do trabalho em equipe e a influência de pessoas mais experientes no aprendizado. É inegável o impacto que esta disciplina, de Técnicas de Programação, tem na formação profissional de um estudante de Engenharia da Computação, visto que lhe é oferecido uma relevante oportunidade de aprendizado, não exclusivamente de programação em C++ Orientado à Objetos, mas também de Diagrama de Classes, Ambientes RAD, Biblioteca Padrão de Templates (*STL – Standard Template Library*), uso de Threads, Padrões de Projeto, Gerenciamento de Versões e Programação Colaborativa.

Visando o desenvolvimento do projeto, primeiramente foi-se compreendido e mapeado os requisitos e necessidades do software. Imediatamente após, realizou-se a modelagem de um Diagramas de Classes em *UML* de Análise, transformando-o em de Projeto. Então, posteriormente foi implementado o jogo em código na linguagem de programação C++ usufruindo do Paradigma de Orientação à Objetos. Em seguida, os próprios desenvolvedores realizaram inúmeros testes para verificar a funcionalidade dos requisitos dentro do software. O meio de comunicação usado foi, em sua maioria, mensagens, ligações de voz e compartilhamento de tela, via *Discord*, para que a compreensão do desenvolvimento do jogo fosse mais clara para ambos os lados. Utilizou-se também um gerenciador de versões de arquivo de arquitetura/sistema distribuído, o git, conjuntamente ao GitHub, formando repositórios, tanto locais quanto online, para a fácil e rápida transferência de arquivos. Por ambos foi utilizado, para a modelagem via Diagrama de Classes em UML, o StarUML, e para implementação, o Ambiente de Desenvolvimento Integrado chamado Microsoft Visual Studio 2019 Community Version, ambas ferramentas gratuitas.

Nas próximas seções deste documento, será abordado a explicação do jogo e suas funcionalidades, jogabilidade e elementos. Subsequentemente, será revelado as mecânicas por trás da lógica do jogo, juntamente aos conceitos utilizados para que tudo funcionasse corretamente.

Explicação do jogo em si

Primeiramente, o software é um jogo de plataformas, onde o jogador assume o controle de um personagem mago que retorna ao ponto inicial toda vez que falhar em seu objetivo de chegar ao final da fase. Caso estiver em dois jogadores, o mago empresta parte de seu poder para o segundo jogador, que incorpora um anjo. Ambos podem lançar um orbe de energia laranja para derrotar os inimigos. Segue a Figura 1, a representação dos personagens.

Figura 1. Anjo, Mago e Orbe de Energia, respectivamente.

O jogo tem um sistema de pontuação que funciona a partir de “penalidade”, e tal penalidade obedece a Sequência de Fibonacci, na qual cada termo subsequente corresponde à soma dos dois anteriores. A cada vez que o Jogador é mandado novamente para o início, ele aumenta sua penalidade pegando o próximo número da dita sequência.

O usuário é constantemente desafiado com golens mágicos em seu caminho, assim como obstáculos e caminhos alternativos, deixando a experiência e jogabilidade mais intrigante. Cada fase tem um nível de dificuldade maior, em um ambiente totalmente diferente, com golens (Figura 2) e obstáculos díspares. Além disso, a última fase propõe um último desafio para o jogador, em que este deve enfrentar um chefão, um golem extremamente maior que os outros. O jogo simula também a gravidade, a colisão com paredes e obstáculos.

Figura 2. Tipos de Inimigos, respectivamente, Golem de Pedra, Golem de Fogo e Golem de Gelo.

Ao abrir o jogo, o usuário se depara com o menu inicial que o dá a opção de jogar, que o leva a outro menu para selecionar quantos jogadores irão participar, ou de carregar uma jogada, onde o software recuperará a última jogada salva, ou também de sair, cuja opção faz com que o jogo se feche. A partir do menu para selecionar o número de jogadores, pode-se escolher um ou dois jogadores, e então, selecionar entre as 3 fases disponíveis. A seguir na Figura 3, os Menus.

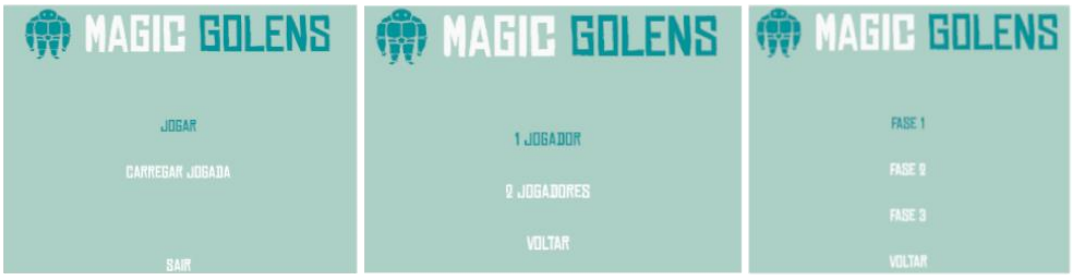

Figura 3. Tela com Menu Inicial, Menu de Seleção de Número de Jogadores e Menu de Fases, respectivamente.

Assim que uma fase se inicia, o personagem aparece diante de uma porta de entrada, e tem como meta achar e chegar até a porta de saída, ambas representadas pela cor branca. O jogador deve ao máximo tentar evitar o contato com os golens e com os obstáculos, visto que são ofensivos e o contato com eles fará com que o personagem tenha que voltar ao ponto de partida e receba outras penalidades.

A fase 1 é tematizada de selva, cujos Inimigos são os Golens de Pedra e Fogo, os obstáculos são lagos de lama. Ao encostar em um golem de Pedra, ele aumenta seu próprio tamanho e vida. Ao encostar em um golem de Fogo, ele aumenta sua velocidade e reduz a cadência de tiro do jogador.

A fase 2 se contextualiza em um lugar onde tudo está em chamas, cujos Inimigos são Golens de Pedra e Fogo, o obstáculo são chamas no chão e caso o jogador entre em contato com as chamas, sua cadência de tiro é reduzida e retorna ao ponto inicial. As consequências do contato com os Golens de Pedra e de Fogo são as mesmas para todas as fases.

A fase 3 se dá em um contexto de temperaturas extremamente baixas, cujos inimigos são Golens de Gelo e de Pedra. Os primeiros atiram cristais de gelo na direção do jogador e o obstáculo são espinhos. Caso o jogador entre em contato com um cristal de gelo, ele é congelado e acaba escorregando na direção em que estava se dirigindo o Cristal de Gelo por um tempo determinado. Caso o jogador encoste nos espinhos, é mandado novamente para o ponto de

partida. Ainda na fase 3, guardando a saída, está o chefão, um Golem de Gelo extremamente grande que atira cristais de gelo no jogador com cadência elevada. Segue a Figura 4, capturas de tela de todas as fases.

Todas as fases têm estalactites que caem e fazem o jogador retornar ao início ao contato.

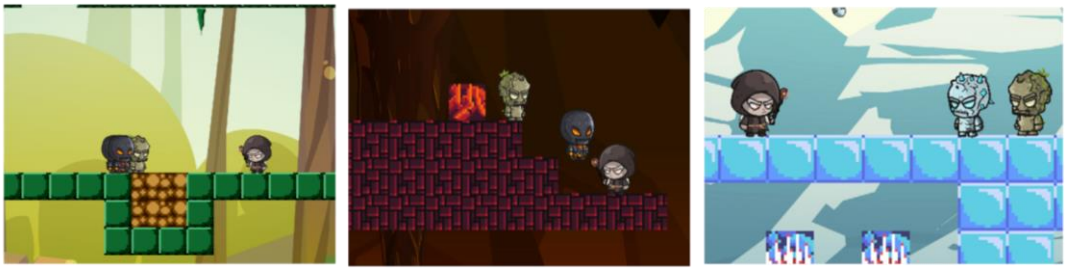

Figura 4. Capturas das Fases 1, 2 e 3, com seus Inimigos e Obstáculos, respectivamente.

DESENVOLVIMENTO DO JOGO NA VERSÃO ORIENTADA A OBJETOS

Nesta seção, serão abordados os requisitos funcionais propostos inicialmente, bem como sua implementação e porcentagem final de requisitos cumpridos.

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 na classe Menu e em suas classes herdadas: MenuFases, MenuInicial, MenuPause, MenuJogadores...
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 com duas classes: Mago e Anjo e seus devidos objetos. No caso de um só jogador, é utilizado somente o Mago.
3	Disponibilizar ao menos duas fases que podem ser jogadas sequencialmente ou selecionadas.	Requisito previsto inicialmente e realizado.	Requisito cumprido nas classes FasePedra, FaseFogo e FaseGelo, na respectiva sequência.
4	Ter três tipos distintos de inimigos (o que pode incluir ‘Chefão’, vide abaixo), sendo que pelo menos um dos inimigos deve ser capaz de lançar projetil contra o(s) jogador(es).	Requisito previsto inicialmente e realizado.	Requisito cumprido nas classes GolemFogo, GolemPedra, GolemGelo, herdadas da classe Inimigo, além da classe Chefao. O GolemGelo também é herdado da classe Atirador, o que significa que é capaz de lançar projéteis.
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 objetos das classes GolemPedra, GolemFogo e GolemGelo ao serem instanciados em números aleatórios.
6	Ter inimigo “Chefão” na última fase	Requisito previsto inicialmente e	Requisito cumprido inclusive via um objeto da classe Chefao.

		realizado.	
7	Ter três tipos de obstáculos.	Requisito previsto inicialmente e realizado.	Requisito cumprido inclusive via classes Espinhos, Areia, Fogo e Estalactite, cada um instanciado nas fases.
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 classes Espinhos, Areia, Fogo e Estalactite, cada um instanciado nas fases, sendo o último instanciado aleatoriamente quanto número e posição.
9	Ter representação gráfica de cada instância.	Requisito previsto inicialmente e realizado.	Requisito cumprido inclusive via funções da biblioteca gráfica SFML, que nos permitiu representar graficamente /em uma janela nossos devidos objetos.
10	Ter em cada fase um cenário de jogo com os obstáculos.	Requisito previsto inicialmente e realizado.	Requisito cumprido inclusive via a classe GerenciadorMapa, Mapa e Tile.
11	Gerenciar colisões entre jogador e inimigos, bem como seus projeteis (em havendo).	Requisito previsto inicialmente e realizado.	Requisito cumprido inclusive via classe GerenciadorColisoes.
12	Gerenciar colisões entre jogador e obstáculos.	Requisito previsto inicialmente e realizado.	Requisito cumprido inclusive via classe GerenciadorColisoes.
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 classe Leaderboard.
14	Permitir Pausar o Jogo	Requisito previsto inicialmente e realizado	Requisito cumprido inclusive via classe MenuPause.
15	Permitir Salvar Jogada.	Requisito previsto inicialmente e realizado	Requisito cumprido inclusive via métodos para salvar de cada Entidade e de métodos nas classes de cada fase para recuperá-los.

Requisitos implementados: 15 de 15 (100%).

Segue a explicação das partes cruciais do código.

Gerenciadores

O jogo é estruturado em cima de classes de gerenciadores, sendo elas: *GerenciadorEstado*, *GerenciadorGrafico*, *GerenciadorAtualizacao*, *GerenciadorMapa* e *GerenciadorColisoes*. O primeiro é o principal e coordena os estados do jogo, bem como controla os outros gerenciadores, oferecendo-lhes atributos para realizar operações e inicializa as fases. O *GerenciadorAtualizacoes* é responsável por cuidar dos intervalos tempo, atualizar a posição e os estados dos elementos na tela de uma certa fase. O *GerenciadorGrafico* tem a reponsabilidade de instanciar a janela, atualizar a posição da câmera para seguir o jogador, receber e tratar eventos e mostrar os elementos e objetos na tela. O *GerenciadorColisoes*

organiza as colisões, chamando o método *colidir()*, e também é responsável por retirar as entidades que foram eliminadas durante a execução. O *GerenciadorMapa* armazena e trata informações sobre o mapa de sua respectiva fase, como as texturas dos tiles e as suas posições, além de repassar ao *GerenciadorColisoes* as tiles que estão colidindo em um certo instante.

Ambos os *GerenciadorGrafico* e *GerenciadorAtualizacoes* são baseados em uma lista de Entidades de cada fase, que recebem do *GerencidaorEstado*. É interessante notar que isso só foi organizado desta maneira pois o Paradigma OO tem o recurso de polimorfismo, podendo generalizar vários objetos como Entidades em uma lista.

Entidades

A classe Entidade representa tudo aquilo que tem uma posição, um tamanho e velocidade em um caso real. Ela dá origem a várias outras classes, cada uma com sua particularidade em quesito de atributos e métodos. Todas as entidades do jogo estão contidas no pacote Entidades, que está repleto de funções virtuais, que mais uma vez mostra o importante papel do polimorfismo na programação OO. Cada Entidade é sempre instanciada e inserida em uma lista de entidades de uma fase específica, para posteriormente ser percorrida por gerenciadores para atualizar sua posição, realizar sua impressão e verificar suas colisões. É relevante notar também que cada Entidade tem sua própria maneira de salvar seus atributos em arquivos, realizando sua própria persistência de objeto. Outrossim, cada Entidade é acelerada para baixo, simulando o efeito da aceleração da gravidade.

Menus

Os menus têm o papel de comunicar as intenções do usuário aos gerenciadores. Por meio deles, pode-se criar uma maleabilidade de opções, por exemplo escolher entre 1 ou 2 jogadores ou também qual fase será jogada. O controle da tabela de pontuação é realizado por um Menu específico chamado *Leaderboard*, que usufrui da persistência de informações em arquivos para salvar dados.

Fases

Cada fase é responsável por instanciar seus Inimigos e Obstáculos. Também cria um *GerenciadorMapa* para administrar a criação do seu mapa. Ademais, cada fase é incumbida de recuperar os objetos que foram salvos em arquivos e reinstanciá-los quando requerido, pois estes só fazem sentido no contexto de sua própria fase.

Mapa

Um objeto da classe mapa, criado a cada fase, é encarregado de absorver informações de um arquivo de texto que contém as posições de cada tipo de tile. Então, essas informações são passadas para o *GerenciadorMapa*. Cada tile que compõe o mapa é uma entidade, que pode se especificar em outros tipos de obstáculos, portas ou simplesmente blocos.

Listas

A lista template produzida pelos desenvolvedores é uma lista duplamente encadeada que tem ponteiros fixos no primeiro e último elemento, e também um de percorrimento. A classe Lista tem uma classe aninhada Elemento, que é uma espécie de *container* que aponta para a

informação inserida na lista. A classe *ListaEntidades* é uma lista desacoplada que tem como núcleo a Lista Template parametrizada com *Entidade*.

Jogo

Enquanto a janela estiver aberta, o ciclo do jogo se dá aproximadamente dessa maneira: verifica-se o estado (se está em menu ou em fase), atualiza-se as informações (posições, velocidades, variação de tempo, et cetera.), verifica-se as colisões (isso se estiver em tempo de fase), desenha-se na tela (todos os elementos e objetos devidos).

TABELA DE CONCEITOS UTILIZADOS E NÃO UTILIZADOS

Nesta seção, serão abordados conceitos vistos durante a disciplina que foram e não foram utilizados na confecção do software.

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	Jogo.h & Jogo.cpp
	- Divisão em .h e .cpp.	Sim	No desenvolvimento como um todo.
2	Relações de:		
	- Associação direcional. & - Associação bidirecional.	Sim	Na maioria das classes (conforme diagrama de classes e respectivo código). Exemplo: Associação bidirecional entre MenuFases e MenuInicial, Associação direcional entre ListaEntidades e Entidade.
	- Agregação via associação. & - Agregação propriamente dita.	Sim	Na maioria das classes (conforme diagrama de classes e respectivo código). Exemplo: Agregação forte entre MenuInicial (agregador) e MenuFases, Agregação fraca entre ListaEntidades (agregador) e Entidade.
	- Herança elementar. & - Herança em diversos níveis.	Sim	Na maioria das classes (conforme diagrama de classes e respectivo código). Exemplo: Herança elementar entre Menu (superclasse) e MenuInicial, Herança em diversos níveis entre as classes Entidade (superclasse), Personagem, Inimigo e GolemFogo.
	- Herança múltipla.	Sim	Jogador tem herança da classe Personagem e Atirador, assim como GolemGelo.
3	Ponteiros, generalizações e exceções		
	- Operador *this*.	Sim	Usado em diversos casos, como por exemplo: na construtora da classe Fase, no arquivo Fase.cpp

	- Alocação de memória (*new* & *delete*).	Sim	Usado em diversos casos, como por exemplo: o *new* foi usado na função *criarMapa()* da classe Fase, no arquivo Fase.cpp; o *delete* foi usado na destrutora da classe Fase, no arquivo Fase.cpp.
	- Gabaritos/*Templates* criada/adaptados pelos autores (e.g. Listas Encadeadas via *Templates*).	Sim	Foi criado uma lista template duplamente encadeada na classe Lista, utilizada para criar a ListaEntidades.
	- Uso de Tratamento de Exceções (*try catch*).	Sim	Construtora da classe Fase, no arquivo Fase.cpp
4	Sobrecarga de:		
	- Construtoras e Métodos.	Sim	Sobrecarga de construtora na classe Textbox, e sobrecarga do método *setPosicaoJogadores()* na classe Fase.
	- Operadores (2 tipos de operadores pelo menos).	Sim	Sobrecarga do operador [] e do operador ! no GerenciadorColisoes.
	Persistência de Objetos (via arquivo de texto ou binário)		
	- Persistência de Objetos.	Sim	Há persistência de Objetos em arquivos de texto presentes na pasta *salvar*. Posteriormente pode-se recuperar tais objetos. Cada Entidade tem seu próprio método de se salvar. Cada Fase consegue recuperar objetos salvos.
	- Persistência de Relacionamento de Objetos.	Sim	As relações que as estalactites têm com os jogadores são recuperadas.
5	Virtualidade:	Sim	
	- Métodos Virtuais.	Sim	Diversas Classes, por exemplo: a função *atualizar()* na classe Personagem.
	- Polimorfismo	Sim	Observando o diagrama de classes, há polimorfismo no pacote Interface, Entidades e Fases. Por exemplo: as classes MenuFases e MenuInicial ambas têm o método *executarEnter()*, herdado da superclasse Menu, entretanto executam procedimentos diferentes.
	- Métodos Virtuais Puros / Classes Abstratas	Sim	As classes abstratas são: Menu, Entidade, Personagem, Fase. Em todas elas há métodos virtuais puros, como por exemplo: o método *executarEnter()* na Classe Menu.
	- Coesão e Desacoplamento	Sim	No desenvolvimento como um todo.
6	Organizadores e Estáticos		
	- Espaço de Nomes (*Namespace*) criada pelos autores.	Sim	Foi utilizado no arquivo IdsColidiveis.h
	- Classes aninhadas (*Nested*) criada pelos autores.	Sim	Dentro do arquivo Lista.h, na classe Lista há uma classe aninhada chamada Elemento.
	- Atributos estáticos e métodos estáticos.	Não	
	- Uso extensivo de constante (*const*) parâmetro, retorno, método...	Sim	No desenvolvimento 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	A classe *string* foi utilizada no desenvolvimento como um todo. *Vector* foi utilizado dentro da classe GerenciadorColisoes.

	- Pilha, Fila, Bifila, Fila de Prioridade, Conjunto, Multi-Conjunto, Mapa OU Multi-Mapa.	Sim	Foi utilizado *Conjunto* (*set*) na classe GerenciadorColisoes, *Mapa* (*map*) na classe GerenciadorMapa.
	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	*Especificar aqui quais funcionalidades*. Da biblioteca gráfica SFML, foram utilizadas as funcionalidades: Vector2f e Vector2i, estruturas que armazenam dois valores, floar e int respectivamente; classes Texture, Sprite, RectShape, RenderWindow, View e seus métodos; das funcionalidades de eventos e input por meio de teclado e mouse, entre outros.
	- Programação orientada e evento em algum ambiente gráfico. OU - *RAD – Rapid Application Development* (Objetos gráficos como formulários, botões etc).	Sim	Uso de um sistema dentro da Biblioteca SFML que suporta eventos.
	Interdisciplinaridades por meio da utilização de Conceitos de Matemática e/ou Física.		
	- Ensino Médio.	Sim	Conhecimento sobre plano cartesiano, eixo das abcissas e ordenadas, conceitos de aceleração, desaceleração, movimento retilíneo uniformemente variado, aceleração gravitacional.
	- Ensino Superior.	Sim	Conceitos mais avançados sobre vetores, normalização de vetores, soma e subtração de vetores, entendimento da sequência de Fibonacci, distância entre dois pontos aplicados à corpos extensos.
9	Engenharia de Software		
	- Compreensão, melhoria e rastreabilidade de cumprimento de requisitos. &	Sim	Foi utilizado na primeira etapa do ciclo clássico de Engenharia de Software, onde todos os requisitos foram considerados e, de antemão, compreendidos.
	- Diagrama de Classes em *UML*.	Sim	Foi utilizado na segunda etapa do ciclo clássico de Engenharia de Software, onde se deu o planejamento e modelagem de análise e projeto via Diagrama de Classes em UML.
	- Uso efetivo (quiçá) intensivo de padrões de projeto (particularmente GOF).	Não	
	- Testes a luz da Tabela de Requisitos e do Diagrama de Classes.	Sim	Foi utilizado na quarta e última etapa do ciclo clássico de Engenharia de Software, onde após implementado código, foi-se testado visando verificar requisitos e funcionalidades.

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	Foi-se utilizado extensivamente um gerenciador de versões de arquivo, de arquitetura/sistema distribuído, o git, conjuntamente ao GitHub[2], formando repositórios, tanto locais quanto online. Ainda nesse sistema, visando a segurança, criou-se *branches* de backup do projeto.
	- Reuniões com o professor para acompanhamento do andamento do projeto.	Sim	Feitas todas as 4 Reuniões: (21/04), (28/04), (05/05) e (07/05).
	- Reuniões com monitor da disciplina para acompanhamento do andamento do projeto.	Sim	Foram feitas somente 6 reuniões, pois as dúvidas foram agrupadas para serem atendidas, visto que havia comumente duplas já em reunião. Um outro fator foi a restrição de horário da dupla. Reuniões: Skora (11/03), Skora (22/03), Augusto (08/04), Augusto (26/04), Skora (29/04), Augusto (07/05).
	- Revisão do trabalho escrito de outra equipe e vice-versa.	Sim	Equipe Garret & Yuske.

Conceitos Utilizados: 36 de 40 (90.00%)

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

No.	Conceitos	Situação
1	Elementares	Classe, Objetos e Atributos foram utilizados para que o desenvolvimento fosse orientado ao Paradigma de OO.
2	Relações	Associação foi utilizado pois a orientação ao Paradigma de OO as exige.
3	Ponteiros, generalizações e exceções	A alocação dinâmica permite-nos construir um software muito mais maleável e adaptável às necessidades do usuário sem utilizar memória sem necessidade. O operador *this* pode simplificar muitas lógicas e evitar, algumas vezes, variáveis com mesmo nome. Gabarito/Template criado por nós facilitou muitos métodos que necessitavam acessar vários objetos do mesmo tipo.
4	Sobrecarga e Persistência	Sobrecarga de Construtoras e Métodos permite maior maleabilidade para os diferentes propósitos. Sobrecarga de operadores nos proporciona maior clareza, e reutilização de código. A persistência de Objetos e seus relacionamentos permitiu recuperar informações de usos em outros momentos e evitar a perda de dados importantes.
5	Virtualidade	A virtualidade pura de métodos, que proporciona o polimorfismo, é extremamente importante para criar a possibilidade de tratar Objetos específicos de maneira geral, abrindo possibilidades para diferentes funcionalidades. É a essência da OO.
6	Organizadores e Estáticos	Os namespaces seriam utilizados para organizar o código em diferentes seções para o melhor entendimento do contexto geral. O namespace foi utilizado para evitar inconsistências de código algumas vezes causada pelo *enum*. As classes aninhadas dão ainda mais possibilidades de representar o mundo como este é, por meio de código. Não foi utilizado métodos estáticos, entretanto estes poderiam ser utilizados para indicar um atributo comum à todos os objetos de uma classe. Os métodos const são ótimas maneiras de manter o código seguro, sem acessos indevidos. Por meio desses, também é possível verificar inconsistências de código.

7	Standard Template Library (STL) e String OO	A classe pré-definida *string* foi utilizada para facilitar o manuseio de strings, além de deixar mais fácil o entendimento. Os gabaritos da STL foram também extremamente úteis no desenvolvimento como um todo. Não se foi utilizado Threads, principalmente por falta de tempo.
8	Biblioteca Gráfica / Visual	Utilizou-se da Biblioteca Gráfica SFML 2.5, pois nos proporcionou uma fácil e rápida implementação gráfica e visual dos nossos objetos, além de ser muito útil para lidar com eventos.
9	Engenharia de Software	Praticou-se o clássico ciclo de Engenharia de Software para a realização do projeto como um todo, levantando requisitos, modelando uma solução, implementando-a e testando a solução à luz dos requisitos. Não se utilizou padrões de projeto pois não se achou a tempo um padrão de projeto que se encaixava à já estruturada base do jogo.
10	Excecução do Projeto	Utilizar um versionador de arquivos foi extremamente útil no trabalho em equipe, além de garantir uma sólida segurança ao comumente realizar backups. As reuniões com o Professor e com os Monitores tiveram um grande impacto na maneira com que o projeto se desenvolveu. Revisar o trabalho de outra equipe é uma ótima maneira de verificar erros que possam ter passado despercebidos.

REFLEXÃO COMPARATIVA ENTRE DESENVOLVIMENTOS

Foi percebido por ambos os desenvolvedores a superioridade, em diversos casos e aspectos, da programação Orientada à Objetos. Ela permite, ao implementador, ter uma noção e visão muito mais ampla do progresso do trabalho e facilita a reutilização e expansão do código. Além do mais, poder trabalhar com uma das linguagens de baixo nível mais famigeradas, ao mesmo tempo em que se é possível representar a vida real no código por meio de classes e seus objetos, é um recurso inigualável.

DISCUSSÃO E CONCLUSÕES

O desenvolvimento deste trabalho deve ser considerado como uma oportunidade de ouro que não se deve ser desperdiçada, visto a imensa gama de conceitos aprendidos e habilidades adquiridas durante sua execução. Pode-se, até mesmo, concluir que a matéria de Técnicas de Programação é uma das, senão a mais importante na formação de um universitário de Engenharia da Computação da UTFPR. O desenvolvimento do software, sem sombra de dúvidas, exerceu seu papel de fixar os conhecimentos e os conteúdos aprendidos durante aula sobre Programação em C++ Orientado à Objetos, Diagrama de Classes em UML e Práticas de Engenharia de Software. A experiência trouxe uma sólida base para a formação acadêmica e profissional de ambos os participantes.

DIVISÃO DO TRABALHO

Nesta seção, serão abordados os tópicos feitos por cada integrante da dupla.

Tabela 4. Lista de Atividades e Responsáveis.

Atividades.	**Responsáveis**
Levantamento de Requisitos	Henrique e Felipe

Diagramas de Classes	Henrique
Programação em C++	Henrique e Felipe
Implementação de *Template*	Henrique
Implementação da Persistência dos Objetos	Mais Felipe que Henrique
Implementação da Persistência de Relação de Objetos	Felipe
Tratamento de Colisões	Henrique e Felipe
Mecânicas físicas	Felipe
Gerenciador Gráfico	Henrique
Mapa e Tiles	Henrique e Felipe
Fases	Henrique e Felipe
Gráficos & Texturas	Henrique e Felipe
Inimigos	Henrique e Felipe
Jogadores	Mais Henrique que Felipe
Menus	Henrique
Leaderboard	Henrique
Junção de versões de código	Felipe
Escrita do Trabalho	Felipe
Revisão do Trabalho	Henrique e Felipe

AGRADECIMENTOS

Agradecimentos ao Prof. Dr. Jean M. Simão pelo conteúdo disponibilizado em sua página[2], que foi importante para a realização deste trabalho. Agradecimentos também aos monitores da disciplina Augusto Mudrei Correia e Lucas Eduardo Bonacio Skora, por toda atenção e suporte oferecido, aos designers que disponibilizaram seus trabalhos gratuitamente e a Rodrigo Yuske Yamauchi e Victor Hugo Garrett pela revisão do documento.

REFERÊNCIAS CITADAS NO TEXTO

[1] Endereço para o Repositório Online no GitHub
https://github.com/Lee3007/TecProgGame

[2] SIMÃO, J. M. Site da Disciplina de Técnicas de Programação, Curitiba – PR, Brasil. Último acessado em 10/05/2021, às 10:32:
https://pessoal.dainf.ct.utfpr.edu.br/jeansimao/Fundamentos2/Fundamentos2.htm

REFERÊNCIAS UTILIZADAS NO DESENVOLVIMENTO

[A] SIMÃO, J. M. Site da Disciplina de Técnicas de Programação, Curitiba – PR, Brasil. Último acessado em 10/05/2021, às 10:32:
https://pessoal.dainf.ct.utfpr.edu.br/jeansimao/Fundamentos2/Fundamentos2.htm