TRABALHO PARA A DISCIPLINA DE TÉCNICAS DE PROGRAMAÇÃO DO CURSO DE SISTEMAS DE INFORMAÇÃO DA UTFPR: *Grande Guerra Geométrica*

Caio e Silva Barbieri, Ana Julia Molinos Leite da Silva
caiobarbieri.aluno@utfpr.edu.br, asilva.2023@alunos.utfpr.edu.br

Disciplina: **Técnicas de Programação – CSE20** / S73 – Prof. Dr. Jean M. Simão
Departamento Acadêmico de Informática – DAINF - Campus de Curitiba
Curso Bacharelado em: 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 propõe o desenvolvimento de um jogo de plataforma 2D escrito em C++, com auxílio da biblioteca gráfica SFML (*Simple and Fast Media Library*), segundo o paradigma de Programação Orientada a Objetos (POO). O processo de desenvolvimento, conforme requerido nas especificações do trabalho, consiste na assimilação dos requisitos propostos, seguida da modelagem do projeto utilizando Linguagem de Modelagem Unificada (*Unified Modeling Language - UML*), implementação do *software* e eventuais testes. Sendo assim, os alunos podem experienciar um ciclo adaptado de engenharia de *software*. Para tal, foi desenvolvido o jogo GGG: Grande Guerra Geométrica, um jogo para 2 jogadores em que formas geométricas combatem entre si em diferentes níveis de dificuldade, conforme fases disponíveis. Durante a evolução do projeto, buscou-se manter a fidelidade aos requisitos funcionais e conceituais, também previamente propostos nas especificações. Ao final do trabalho, além da produção do jogo, foi possível obter uma grande experiência no que diz respeito a interseccionalidade de temas como orientação a objetos, engenharia de requisitos, habilidades técnicas e habilidades comportamentais,

Palavras-chave ou Expressões-chave: Programação Orientada a Objetos, Engenharia de Software, Desenvolvimento de Jogo, Herança e Polimorfismo.

INTRODUÇÃO

Este trabalho foi desenvolvido na disciplina de Técnicas de Programação, ministrada na Universidade Tecnológica Federal do Paraná, sob orientação do ~~p~~Professor Dr. Jean Marcelo Simão. Consistindo em 50% da nota de avaliação da disciplina, o trabalho se deu em três componentes: o relatório presente, a apresentação expositiva e o diagrama de classes juntamente com a implementação do software. O objetivo de sua realização foi aplicar de forma prática os conceitos relacionados à programação orientada a objetos vistos em aula durante a disciplina, como por exemplo, coesão, desacoplamento, herança e polimorfismo, além dos recursos da linguagem utilizada.

O contexto de desenvolver um jogo se mostra ideal para o aprendizado de orientação a objetos, uma vez que um ele tem componentes bem definidos. Desse modo, o objeto de estudo do trabalho foi um jogo de plataforma 2D, que deveria atender a diversos requisitos funcionais e conceituais pré-definidos. Para tal, foi desenvolvido o jogo GGG: Grande Guerra Geométrica, um jogo para 2 jogadores em que círculos precisam enfrentar inimigos com formas convexas.

O método utilizado para estruturação do trabalho foi uma adaptação do ciclo de Engenharia de Software. Foram seguidas as seguintes etapas: compreensão de requisitos, modelagem do projeto via diagrama de classes com estrutura pré-definida, realizado em *Unified Model Language (UML),* implementação do software, realizada utilizando-se a linguagem de programação C++ orientada a objetos, e testes constantes em tempo de desenvolvimento.

Comentado [JS1]: Sds,

Esta é a correção do trabalho referente à 2ª parcial. Salienta-se que as críticas são sempre construtivas, sendo os (eventuais) textos de críticas diretos apenas por motivo de ganho de tempo.

Cordialmente

Att.

Prof. J. M. Simão.

Qualidade: 2,2 / 3,0
(Reu/Ap 0,7 ; Txt/Ap 0,8; Cod/Diag 0,7)
Sistema (RF – Tab1): 3,2 / 3,5
Conceitos (RT – Tab2): 2,9 / 3,5

Nota do trabalho como um todo: 8,3/ 10,0.

NOTA (100%): Caio 8,3 / 10,0

NOTA (100%): Ana 8,3 / 10,0

A seção subsequente tem por objetivo explicar o contexto e funcionamento do jogo desenvolvido de maneira detalhada. Em seguida, é feita a análise do cumprimento dos requisitos funcionais pré-definidos e do diagrama de classes confeccionado. Na sequência, há um mapeamento de quais conteúdos vistos durante a disciplina foram efetivamente aplicados no desenvolvimento do projeto. Por fim, há uma seção dedicada a esclarecer como se deu a divisão de atividades ao longo da realização do trabalho.

EXPLICAÇÃO DO JOGO EM SI

O jogo GGG: Grande Guerra Geométrica se passa no seguinte contexto: em um mundo em que os seres consistem em formas geométricas, estimulados por preconceitos contra aquelas formas que não possuem vértices, as formas com ângulos passam a perseguir seus não semelhantes, os círculos, a fim de eliminá-los. Isso posto, os jogadores consistem em dois círculos que foram presos e agora precisam se esquivar dos ataques inimigos e desviar de obstáculos geométricos, fugindo da perseguição e alcançando seu objetivo principal ao fim do jogo: a liberdade.

O jogo conta com menus (figura 1) que permitem ao usuário navegar por meio do teclado e escolher qual fase jogar, bem como a quantidade de jogadores (sendo 2 o máximo). Além disso, é possível se registrar no ranking (figura 2) de pontuação e acompanhar os 10 melhores jogadores, pausar o jogo durante seu andamento e salvar a partida para continuá-la em outro momento pertinente. No jogo, os jogadores precisam se movimentar a partir das teclas “Esquerda”, “Direita” e “Cima” (para o jogador 1), e “A”, “D” e “W” (para o jogador 2). Os círculos iniciam as fases com 30 unidades de vida e vão perdendo-a à medida que recebem dano dos inimigos ou obstáculos. Para ganhar pontos, eles precisam neutralizar os inimigos pulando sobre eles.

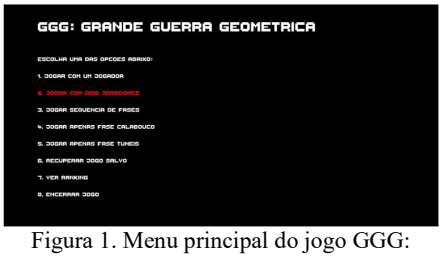

Figura 1. Menu principal do jogo GGG: Grande Guerra Geométrica.

Figura 2. Ranking do jogo GGG: Grande Guerra Geométrica.

A primeira fase (figura 3), chamada de “Calabouço”, se resume em uma espécie de calabouço onde os jogadores precisam percorrer plataformas, combatendo inimigos e desviando dos obstáculos, para chegar ao portal e emergir para a próxima fase. Os principais inimigos nesse momento são os triângulos e os quadrados. Os quadrados, inimigos com nível de maldade baixo, ficam parados e só se movimentam com a aproximação do jogador, o perseguindo. Se ficarem pertos o suficiente, desfere um ataque contra ele. Já, os triângulos, inimigos com nível de maldade médio, também ficam parados, mas lançam projéteis inteligentes voadores, que perseguem o jogador até colidirem com ele ou com qualquer outra entidade do jogo. Neutralizar esses inimigos rende uma quantidade de pontos ao jogador que é proporcional ao seu nível de maldade e sua vida máxima. Além disso, essa fase conta com a plataforma grudenta como obstáculo. Ela diminui a velocidade do jogador ao contato, em uma quantidade proporcional a sua pegajosidade.

Figura 3. Fase 1 do jogo GGG: Grande Guerra Geométrica.

A segunda fase (figura 5), dita fase final, é chamada de “Túneis”. Essa fase se passa em uma sequência de corredores que precisam ser percorridos até o final para que os círculos alcancem sua liberdade e os jogadores vençam o jogo. O novo inimigo da fase é o chefão estrela, com nível de maldade alto, que se mantém estático em seu lugar, disparando 3 projéteis inteligentes voadores simultaneamente contra o jogador. Esse inimigo tem a habilidade de invocar capangas ao redor dele, fazendo com que no decorrer da fase existam mais inimigos a serem neutralizados. Após um tempo, no entanto, os inimigos do tipo chefão estrela se arrependem de seus atos discriminatórios contra os círculos e ficam imensamente tristes, parando de atacá-los. Se o jogador os neutralizar após o arrependimento, ao invés de ganhar pontos, ele os perderá. O obstáculo adicional é o lápis-espinho, ao entrar em contato com esse obstáculo, os jogadores recebem um dano de 2 unidades em sua vida.

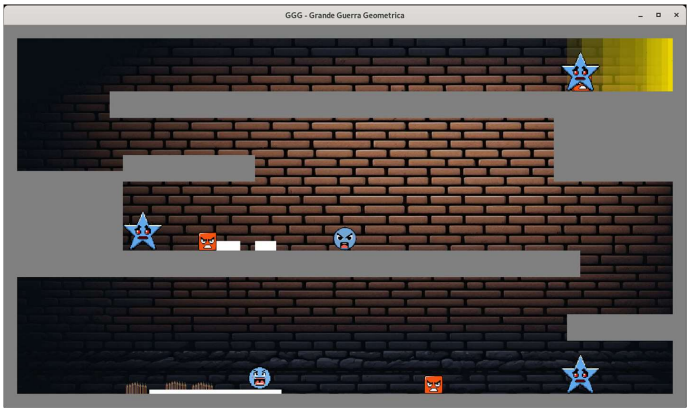

Figura 4. Fase 2 do jogo GGG: Grande Guerra Geométrica, com os inimigos do tipo chefão estrela já em estado de “arrependimento”.

DESENVOLVIMENTO DO JOGO NA VERSÃO ORIENTADA A OBJETOS

Inicialmente o desenvolvimento do jogo se deu por meio do cumprimento da lista de requisitos funcionais exigida, cujo acompanhamento da realização está registrado na Tabela 1.

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 (previstas nos demais requisitos).	Requisito previsto inicialmente e realizado.	Requisito cumprido via classes do namespace Menus, derivadas de Menu e Estado, e seus respectivos objetos, com suporte da classe Texto, responsável pela interface com a biblioteca gráfica 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 e realizado.	Requisito cumprido inclusive via classe Jogador cujos objetos são agregados ao jogo, 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 e realizado.	Requisito cumprido via classes do namespace Fases, Calabouco e Tuneis, cujas instâncias são agregadas ao gerenciador de estados, com auxílio da classe MenuPrincipal, que permite a navegação entre fases.
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 e realizado.	Requisito cumprido conforme classes presentes no namespace Inimigos, Triangulo (lança projéteis), Quadrado e Estrela (chefão que lança projéteis), e seus respectivos objetos, que são agregados à Fase.
5	Ter a cada fase ao menos dois tipos de inimigos com número aleatório de instâncias, podendo ser várias instâncias (definindo um máximo) e sendo pelo menos 3 instâncias por tipo.	Requisito previsto e realizado.	Requisito cumprido inclusive via artifício de geração de números pseudo-aleatória, que define a chance de um um inimigo ser alocado, com máximo definido por meio de atributo do tipo inteiro nas fases.
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 e realizado.	Requisito cumprido inclusive via classes do namespace Obstaculos, Plataforma, PlataformaGrudenta e Lapis (obstáculo que causa dano), cujos objetos também são agregados pelas fases.
7	Ter em cada fase ao menos dois tipos de obstáculos com número aleatório (definindo um máximo) de instâncias (*i.e.*, objetos), sendo pelo menos 3 instâncias por tipo.	Requisito previsto e realizado.	Requisito cumprido inclusive via artifício de geração de números pseudo-aleatória, que define a chance de um obstáculo ser alocado, com máximo definido por meio de atributo do tipo inteiro nas fases.

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 e realizado.	Requisito cumprido inclusive via métodos de criação dos objetos pertinentes ao cenário, implementados na classe Fase.
9	Gerenciar colisões entre jogador para com inimigos e seus projeteis, bem como entre jogador para com obstáculos. Ainda, todos eles devem sofrer o efeito de alguma ´gravidade´ no âmbito deste jogo de plataforma vertical e 2D.	Requisito previsto e realizado.	Requisito cumprido inclusive via gerenciador de colisões. Todas as entidades sofrem ação da gravidade, mas algumas, por outra força aplicada, se sustentam, o que se dá por meio do atributo *acelVertical* nas entidades.
10	Permitir: (1) salvar nome do usuário, manter/salvar pontuação do jogador (incrementada via neutralização de inimigos) controlado pelo usuário e gerar lista de pontuação (*ranking*). E (2) Pausar e **Salvar/Recuperar** Jogada.	Requisito previsto e realizado.	Requisito cumprido inclusive via: 1. Atributo "pontos" do tipo inteiro da classe Jogador; 2. Classes do namespace Menus, responsáveis por fazer a interface com o usuário; 3. Classe GerenciadorEventos responsável por permitir entradas do teclado.
Total de requisitos funcionais apropriadamente realizados. *(Cada tópico vale 10%, sendo que para ser contabilizado deve estar realizado efetivamente e não parcialmente)*			**~~100~~ 90 % (cem por cento)**

Sequencialmente, observando o diagrama de classes na figura 5, é possível esclarecer com maiores detalhes como se deu a modelagem do projeto. A classe principal do software é chamada de Jogo e agrega tanto os dois jogadores quanto os objetos das classes pertencentes ao pacote Gerenciadores (à exceção do Gerenciador de Colisões), que se relacionam entre si por meio de associação e são responsáveis por dar suporte à execução do jogo. O seu *executar* inicia o menu principal por meio do Gerenciador de Estados.

As classes GerenciadorEventos e GerenciadorGrafico, assim como as classes do pacote ElementosGraficos e a classe Coordenada, são responsáveis por realizar a interface com a biblioteca gráfica utilizada no software, *Simple and Fast Multimedia Library (SFML)*, utilizando seus recursos e permitindo maior desacoplamento no código. O gerenciador gráfico possui todos os métodos necessários para manipulação da janela e da câmera e elementos gráficos no geral, e atua conforme o padrão de projeto *Façade*.

No pacote ElementosGraficos, a classe Texto é responsável pelos métodos de formatação e renderização de elementos textuais, enquanto a classe Forma é responsável pelos métodos de todos os outros elementos visuais presentes no jogo. Por fim, a classe Coordenada facilita a passagem por parâmetro e realização de operações com informações como dimensões ou posições, amplamente utilizadas nesse contexto.

O gerenciador de eventos também utiliza recursos da biblioteca gráfica já citada, mas para cuidar da captação de eventos como entradas do teclado, que são convertidas em strings e repassadas ao estado executado atualmente, com o auxílio da classe GerenciadorEstados.

Este, por sua vez, é responsável por controlar os estados de jogo, armazenando todos os estados alocados, permitindo a troca e recuperação de estados por meio de seus métodos, de

modo que um estado possa ter acesso a outro via ponteiro. A alocação e execução de um estado se dá única e exclusivamente pelo gerenciador de estados, permitindo maior controle na execução do jogo.

O gerenciador de estados tem uma relação de agregação forte com todos os estados, que consistem nas classes descritas nos pacotes Menus e Fases. Essas, por sua vez, fazem parte da implementação do padrão de projeto *State*, correspondendo aos estados concretos. O contexto se trata da classe principal, que pode assumir diferentes estados durante a execução do jogo.

Finalizando o pacote de Gerenciadores, há a classe GerenciadorColisoes, cujo objeto é responsável por dar suporte de maneira particular a cada fase, administrando as colisões entre todas as entidades agregadas, por meio do método polimórfico "reagir a colisão". Todas as classes do pacote se utilizam do padrão de projeto *Singleton*, uma vez que a existência de mais de um objeto de qualquer uma dessas classes se faz desnecessária e/ou poderia ocasionar falhas na execução do jogo.

A classe abstrata Ente apresenta os principais métodos intrínsecos a todos os componentes do jogo: executar e desenhar que, por essa razão, são virtuais puros. As entidades citadas são objetos da classe Entidade, derivada de Ente. Elas são agregadas pela fase por meio da classe ListaEntidade, que é estruturada utilizando a classe template Lista parametrizada com o tipo Entidade, ambas pertencentes ao pacote Listas. Esta última utiliza possui duas classes aninhadas: Elemento, nodo de encadeamento duplo, e Iterador, implementado com base no padrão de projeto *Iterator,* possibilitando percorrer os objetos e chamar esses métodos, realizando o conceito de polimorfismo.

No pacote Entidades é possível perceber a relação hierárquica entre as entidades do jogo, que são compostas por personagens, divididos entre inimigos e jogadores, obstáculos e projéteis. No pacote Personagens é possível notar o polimorfismo que se destaca em métodos como atacar, danificar e receber dano. Os objetos do pacote Inimigos no geral se relacionam por meio de associação direcional com os objetos da classe Jogador, a fim de atacá-los ou persegui-los. Essa última relação também se dá entre o projétil e o jogador. Já no pacote Obstáculos, o polimorfismo se faz presente por meio do método obstacular. Desse modo, é possível notar a constante aplicação dos conceitos de orientação a objetos no desenvolvimento do projeto.

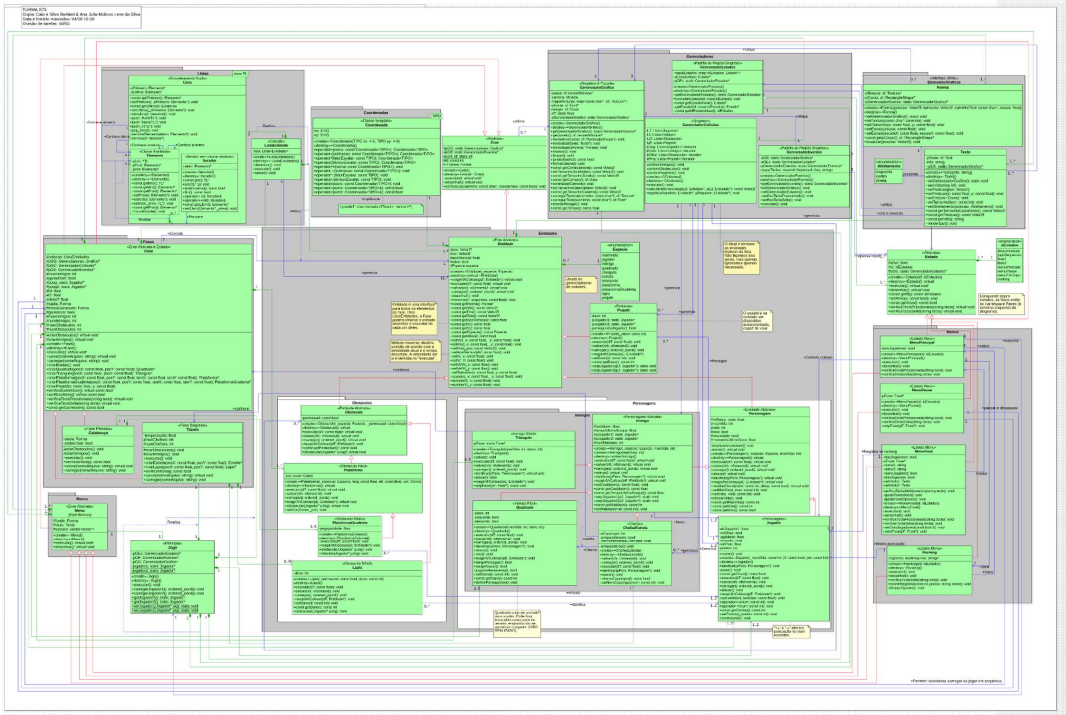

Figura 5. Diagrama de Classes completo em UML.[1]

TABELA DE CONCEITOS UTILIZADOS E NÃO UTILIZADOS

O acompanhamento dos requisitos conceituais realizados e não realizados pode ser feito por meio da tabela 2. É possível observar que foi utilizada uma grande quantidade de conceitos vistos em disciplina, alcançando 85% dos requisitos conceituais inicialmente propostos.

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

N.	Conceitos	Uso	Onde / O quê / Justificativa em uma linha
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* Entidades, Personagens, Inimigos, Obstaculos, Gerenciadores, Coordenadas, ElementosGraficos, Menus Fases e Listas.
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* Entidades, Personagens, Inimigos, Obstaculos, Gerenciadores, Coordenadas, ElementosGraficos, Menus Fases e Listas.
1.3	- Classe Principal.	Sim	Main.cpp & Jogo.h/.cpp

[1] Não foi possível apresentar o diagrama de classes de forma legível devido a extensão do projeto.

1.4	- Divisão em .h e .cpp.	Sim	No desenvolvimento como um todo, como nas classes nos *namespaces* Entidades, Personagens, Inimigos, Obstaculos, Gerenciadores, Coordenadas, ElementosGraficos, Menus Fases e Listas..
2	**Relações de:**		
2.1	- Associação direcional. & - Associação bidirecional.	Sim	Associação direcional em vários dos .h e .cpp, como nas classes nos *namespaces* Gerenciadores, em que diversos objetos precisam conhecer os demais gerenciadores. Além disso, como nas classes do *namespace* Inimigos e na classe Projeteis, devido a necessidade de perseguir e/ou atacar os jogadores, entre outros.
2.2	- Agregação via associação. & - Agregação propriamente dita.	Sim	Em vários dos .h e .cpp, como nas classes no *namespaces* Menus, que agregam via associação e de maneira direta objetos da classe Texto. Além disso, como na classe GerenciadorEstados, que agrega fracamente via associação os objetos das classes concretas derivadas da classe Estado, e como no *namespace* Entidades, cujas classes agregam via associação objetos da classe Forma, entre outros.
2.3	- Herança elementar. & - Herança em diversos níveis.	Sim	Em alguns dos .h e .cpp, como nas classes nos *namespaces* Entidades, Fases e Menus, que derivam de Ente. Além disso, nos *namespaces* Personagens, Inimigos e Obstáculos, em que há diversas outras derivações.
2.4	- Herança múltipla.	Sim	Precisamente nos .h e .cpp, das classes Fase, que deriva de maneira múltipla de Ente e Estado, das classes no *namespace* Menus, que deriva de Menu e Estado.
3	**Ponteiros, generalizações e exceções**		
3.1	- Operador *this* para fins de relacionamento bidirecional.	Sim	Precisamente nos .h e .cpp, das classes Calabouco e Tuneis, em que as fases conhecem os inimigos e os inimigos (precisamente do tipo Triângulo e Estrela) precisam conhecer a fase a fim de chamar o método responsável por criar projéteis.
3.2	- Alocação de memória (*new* & *delete*).	Sim	Na maioria dos .h e .cpp, como nas classes nos *namespaces* Entidades, Personagens, Inimigos, Obstaculos, Gerenciadores, Coordenadas, ElementosGraficos, Menus Fases e Listas.
3.3	- Gabaritos/*Templates* criada/adaptados pelos autores (*e.g.*, Listas Encadeadas via *Templates*).	Sim	Precisamente nos .h e .cpp das classes ListaEntidade e Lista<TL>.
3.4	- Uso de Tratamento de Exceções (*try catch*).	Sim	Precisamente nos .h e .cpp das classes Calabouco e Tuneis, utilizado para salvamento dessas fases.
4	**Sobrecarga de:**		
4.1	- Construtoras e Métodos.	Sim	Sobrecarga de métodos nos .h e .cpp das classes GerenciadorGrafico e das classes do *namespace* ElementosGraficos. Sobrecarga de construtora nos .h e .cpp da classe Inimigo, para possibilitar instanciá-los sem especificar espécie.
4.2	- Operadores (2 tipos de operadores pelo menos – Quais?).	Sim	Foram sobrecarregados vários operadores em Coordenadas.hpp, inclusive +, -, > e < (estes dois últimos usados no cálculo das colisões). Também a classe Iterador, da Lista, contém sobrecarga do operador "++", tal qual na STL.
---	**Persistência de Objetos (via arquivo de texto ou binário)**		

4.3	- Persistência de Objetos.	Sim	Utilização de arquivo de texto para salvamento realizado via json utilizando biblioteca Nlohmann.
4.4	- Persistência de Relacionamento de Objetos.	Não	
5	**Virtualidade:**		
5.1	- Métodos Virtuais Usuais.	Sim	Em diversos .h e .cpp, como nas classes dos *namespaces* Entidades, Personagens, Inimigos, Obstaculos, Menus, Fases.
5.2	- Polimorfismo.	Sim	Em diversos .h e .cpp, como nas classes dos *namespaces* Entidades, Personagens, Inimigos, Obstaculos, Menus, Fases.
5.3	- Métodos Virtuais Puros / Classes Abstratas.	Sim	Em diversos .h e .cpp como nas classes Ente, Entidade, Personagem, Inimigo, Obstaculo, Estado, Menu e Fase
5.4	- Coesão/Desacoplamento efetiva e intensa com o apoio de padrões de projeto.	Sim	Foram utilizados os seguintes padrões de projeto: S*ingleton*, *State*, *Iterator* e *Façade*.
6	**Organizadores e Estáticos**		
6.1	- Espaço de Nomes (*Namespace*) criada pelos autores.	Sim	Em todo o projeto. *Namespaces:* Coordenadas,, ElementosGraficos e Menus.
6.2	- Classes aninhadas (*Nested*) criada pelos autores.	Sim	Precisamente nos .h e .cpp da classe lista há duas classes aninhadas: Elemento e Iterador.
6.3	- Atributos estáticos e métodos estáticos.	Sim	Em diversos .h e .cpp, como nas classes do *namespace* Gerenciadores.
6.4	- Uso extensivo de constante (*const*) parâmetro, retorno, método...	Sim	Em diversos .h e .cpp, especificamente nos métodos *get* que são constantes e retornam constantes (excetuando-se retorno de ponteiros), bem como são constantes os parâmetros de construtoras em geral.
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	Em diversos .hpp e .cpp, como nas classes dos *namespaces* Menus e Fase, e especificamente na classe GerenciadorEventos, a classe *String* é utilizada. Em diversos .hpp e .cpp, especificamente nas classes do *namespace* Menus, *Vector* foi usado para armazenar as opções de texto que são exibidas para o usuário.
7.2	- Pilha, Fila, Bifila, Fila de Prioridade, Conjunto, Multi-Conjunto, Mapa **OU** Multi-Mapa.	Sim	Em diversos .hpp e .cpp, como nas classes GerenciadorEventos, GerenciadorGrafico, GerenciadorEstados, associando informações a seus nomes, caminhos ou identificações.

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	Não	
8	**Biblioteca Gráfica / Visual**		

8.1	- Funcionalidades Elementares. **&** - Funcionalidades Avançadas como: • tratamento de colisões • duplo *buffer*	Sim	Foram utilizadas diversas funcionalidades da biblioteca gráfica *Simple and Fast Multimedia Library* (*SFML*), relacionadas ao uso de janela e câmera para execução do jogo. Além disso, recursos textuais dessa mesma biblioteca foram aplicados na construção dos menus e recursos relacionados a formas foram utilizados na construção das fases. O tratamento de colisões foi feito por meio do cálculo da intersecção pelo tamanho das formas 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).	Não	
---	**Interdisciplinaridades via utilização de Conceitos de Matemática Contínua e/ou Física.**		
8.3	- Ensino Médio Efetivamente.	Sim	Foram aplicados conceitos de física como movimento retilíneo uniforme variado na movimentação de todas as entidades, além de conceitos relacionados à gravidade.
8.4	- Ensino Superior Efetivamente.	Não	
9	**Engenharia de Software**		
9.1	- Compreensão, melhoria e rastreabilidade de cumprimento de requisitos. &	Sim	Realizado por meio de: Solicitações de orientação feitas durante as reuniões com o professor e monitores. Atualização constante do estado de realização dos requisitos por meio do presente relatório. Uso da ferramenta *Project* para gestão de *Issues* do *GitHub*, vinculada ao próprio repositório do software desenvolvido.
9.2	- Diagrama de Classes em *UML*.	Sim	Baseado inclusive em modelo inicial fornecido pelo professor neste documento.
9.3	- Uso efetivo e intensivo de padrões de projeto *GOF*, *i.e.*, mais de 5 padrões.	Não	Utilizou-se os seguintes padrões, já citados anteriormente: *Singleton*, *State*, *Iterator* e *Façade*.
9.4	- Testes à luz da Tabela de Requisitos e do Diagrama de Classes.	Sim	Cotejou-se (especialmente no final) o estado corrente do programa com o diagrama. Ora ou outra de checar o diagrama-base fornecido pelo professor.
10	**Execução de Projeto**		
10.1	- Controle de versão de modelos e códigos automatizado (via github e/ou afins). & - Uso de alguma forma de cópia de segurança (*i.e.*, *backup*).	Sim	Realizado por meio da ferramenta de versionamento *Git* atrelada ao repositório *Github*[1], além de backup e compartilhamento de arquivos e documentos no *Google Drive,* backup local frequente no HD.
10.2	- Reuniões com o professor para acompanhamento do andamento do projeto. **[ITEM OBRIGATÓRIO PARA A ENTREGA DO TRABALHO]**	Sim	2 reuniões: 1. Dia 21 de agosto às 15h30min; 2. Dia 28 de agosto às 15h30min.

10.3	- Reuniões com monitor da disciplina para acompanhamento do andamento do projeto. **[ITEM OBRIGATÓRIO PARA A ENTREGA DO TRABALHO]**	Sim	5 reuniões cujas informações a respeito do que se foi apresentado e/ou discutido foram encaminhadas via e-mail. 1. Dia 13/08, das 15:30 às 17:00, curso com Peteco; 2. Dia 14/08, das 13:00 às 14:00, curso com Peteco; 3. Dia 20/08, das 15:50 às 16:40, reunião com o monitor Edison via Discord; 4. Dia 21/08, das 13:00 às 13:50, curso com Peteco; 5. Dia 27/08, das 09:30 às 10:15, reunião com o monitor Nicky via Discord.
10.4	- Revisão do trabalho escrito de outra equipe e vice-versa.		Revisado pela dupla: Daniel Insaurriaga Zagroba e Samuel Stroschein Yokoyama.
Total de conceitos apropriadamente utilizados. *(Cada grande tópico vale 10% do total de conceitos. Assim, por exemplo, caso se tenha feito metade de um tópico, então valeria 5%.)*			**82,505% ~~(oitenta e cinco por cento)~~**

DISCUSSÃO E CONCLUSÕES

Durante o avanço do projeto, ficou claro como a programação orientada a objetos torna o desenvolvimento do código mais prático e eficiente. A reutilização de métodos por meio do conceito de herança facilitou a implementação de diversos recursos no jogo. Por outro lado, a engenharia de requisitos também se mostrou deveras necessária à organização e implementação do código. Ter noção prévia de quais seriam os objetivos e necessidades de cada classe, ao modelar o projeto construindo seu diagrama de classes com antecedência, auxiliou a minimizar erros e tornou o desenvolvimento mais fluido, especialmente no que diz respeito ao trabalho em equipe.

DIVISÃO DO TRABALHO

A divisão de atividades durante todo o transcorrer do projeto se manteve equilibrada e pode ser vista de maneira mais detalhada na tabela 3.

Tabela 3. Lista de Atividades e Responsáveis.

Atividades.	Responsáveis
Compreensão de Requisitos	Caio e Ana Julia
Diagramas de Classes	Caio e Ana Julia
Programação em C++	Caio e Ana Julia
Implementação de *Template*	Caio
Implementação da Persistência dos Objetos	Caio
Implementação do namespace Menus	Ana Julia
Implementação de salvamento do Ranking	Ana Julia
Implementação do Gerenciador de Colisões	Caio
Implementação dos demais Gerenciadores do namespace	Ana Julia
Implementação do namespace Inimigos	Caio com ajuda da Ana Julia
Implementação da classe Jogador	Caio
Implementação do namespace Fases	Caio
Implementação do namespace Coordenadas	Ana Julia
Implementação do namespace ElementosGraficos	Ana Julia

Implementação da classe abstrata Ente	Caio
Implementação da classe abstrata Entidade	Caio
Implementação da classe abstrata Personagem	Caio
Testes das implementações	Caio e Ana Julia
Preparação de apresentação	Caio
Escrita do Trabalho	Ana Julia com ajuda do Caio
Revisão do Trabalho	Caio e Ana Julia

Em termos de realização (*e.g.*, modelagem e escrita de código) e colaboração (*e.g.*, revisão de código e testes):

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

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

AGRADECIMENTOS PROFISSIONAIS

Agradecimentos ao PETECO por realizar um curso de apoio aos estudantes.

Agradecimentos especiais ao ~~p~~Professor Dr. Jean M. Simão, aos monitores Edison, Gabrielle e Nicky, e ao integrante do grupo PETECO, Jean. O acompanhamento e o feedback por eles fornecidos foi fundamental para nortear a tomada de decisões, o planejamento e a priorização ao longo do desenvolvimento do projeto.

REFERÊNCIAS CITADAS NO TEXTO

[1] BARBIERI, C. S.; SILVA, A. J. M L. ggg. Disponível em <https://github.com/caio-o/ggg>. Acessado em 02/09/2024, às 19h:47.

REFERÊNCIAS UTILIZADAS NO DESENVOLVIMENTO

[A] DEITEL, H. M.; DEITEL, P. J. C++ Como Programar. 5ª Edição. Bookman. 2006.

[B] SIMÃO, J. M. Site das Disciplina de Fundamentos de Programação 2, Curitiba – PR, Brasil, Disponível em <http://www.dainf.ct.utfpr.edu.br/~jeansimao/Fundamentos2/Fundamentos2.htm>. Acessado em 02/09/2024, às 20:13.

[C] GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. Design Patterns. Elements of Reusable Object-Oriented Software. Editora Pearson Education Corporate Sales Division. 1994. ISBN 0-201-63361-2.

[D] BURDA, M. A. Tutorial Jogo SFML. Disponível em <https://www.youtube.com/playlist?list=PLSPev71NbUEBIQlT2QCd-gN6l_mNVw1cJ>. Acessado em 02/09/2024, às 20:16.

[E] SALVI, G. L. Criando um Jogo em C++ do ZERO. Disponível em <https://www.youtube.com/playlist?list=PLR17O9xbTbIBBoL3lli44N8LdZVvg-_uZ>. Acessado em 02/09/2024, às 20:21.