TRABALHO PARA A DISCIPLINA DE TÉCNICAS DE PROGRAMAÇÃO DO CURSO DE SISTEMAS DE INFORMAÇÃO DA UTFPR: *MEDIEVAL++*

Gabriel Felipe Miguel Machado, Yago Augusto Constantino Ribeiro
gabmac@alunos.utfpr.edu.br, yagoribeiro@alunos.utfpr.edu.br

Disciplina: **Técnicas de Programação – ICSE20** / 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 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 Medieval++, no qual o jogador enfrenta inimigos em dois cenários distintos. O jogo tem duas fases que se diferenciam por dificuldades para o jogador. Para o desenvolvimento do jogo foram considerados os requisitos textualmente propostos e elaborado modelagem (análise e projeto) via Diagrama de Classes em Linguagem de Modelagem Unificada (*Unified Modeling Language – UML*) usando como base um diagrama assaz genérico e prévio proposto. Subsequentemente, em linguagem de programação C++, realizou-se o desenvolvimento que contemplou os conceitos usuais de Orientação a Objetos como Classe, Objeto e Relacionamento, bem como alguns conceitos ditos avançados, como por exemplo, 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 o uso do jogo feitos pelos próprios desenvolvedores, demonstraram a sua funcionalidade conforme os requisitos e a modelagem elaborada. 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++; paradigma de Orientação a objetos aplicado a jogos; Engenharia de Software e diagrama de classes UML.

INTRODUÇÃO

Este trabalho se deu como uma forma de avaliação da disciplina de Técnicas de Programação da UTFPR, no qual teve como objetivo a realização de um jogo de plataforma na linguagem de programação C++. A fim de aprender, principalmente, o paradigma de Orientação a Objetos. Portanto, foi necessário o uso de diversos requisitos e conceitos para o projeto, os quais serão expostos e explicados neste documento nas respectivas seções.

Dessa forma, o jogo, como acordado com o professor Dr. Jean M. Simão, deve ser em estilo 2D em plataforma. Assim, foi usado a IDE Visual Studio, além da biblioteca externa gráfica SFML. Ademais, foi permitida a possibilidade de expandir o projeto além dos critérios pedidos, como a implementação de áudio, som e animações, as quais não foram utilizadas no trabalho.

Por conseguinte, para a realização da avaliação, foi aplicado o ciclo de Engenharia de Software. Isto é, o levantamento de requisitos e conceitos, análise e interpretação do trabalho via diagrama de classes em UML, implementação em C++, e testes pelo uso do software.

Por fim, serão apresentados as seguintes seções:

1. Explicação do jogo em si: será explicado a funcionalidade do jogo, principalmente as suas mecânicas (e às vezes a lógica correspondente), passando por todos os elementos mais importantes;

2. Desenvolvimento do jogo na versão orientada a objetos: será exposto como foi feito o projeto, através de conceitos do paradigma de Orientação a objetos e Engenharia de Software, além do diagrama de classes UML correspondente. Ademais, é mostrado uma tabela de requisitos funcionais que teve que ser cumprido;

3. Tabela de conceitos utilizados e não utilizados: será mostrado a tabela de conceitos, cujo trabalho teve que seguir, junto ao *status* atual, ou seja, se foi ou não realizado e onde;

4. Discussão e conclusões: será apresentado uma reflexão final sobre o que os desenvolvedores puderam concluir sobre a atividade avaliativa.

5. Divisão do trabalho: será mostrado a participação de cada integrante em relação a tabela de conceitos, junto com um percentual, no fim, de quanto cada um colaborou.

6. Agradecimentos profissionais: aqui serão reconhecidos os esforços dos indivíduos externos que possibilitaram a realização e a compreensão do jogo em si.

EXPLICAÇÃO DO JOGO EM SI

O jogo Medieval++ começa pelo menu principal (conforme a figura 1), no qual o(s) jogador(es) podem escolher entre: ou clicar no botão “Jogar” e entrar no menu de jogadores (ou retomar uma partida salva), ou clicar no botão “Ranking” e entrar no menu de ranking, ou clicar no botão “Sair” e fechar a janela do jogo (ou apertando a tecla ESC).

Ao entrar no menu ranking, poderá ser visto as maiores pontuações cadastradas de diferentes jogadores (conforme a figura 2). Mas também, é possível clicar no botão “Voltar” e reentrar no menu principal (ou apertando a tecla ESC).

No menu de jogadores (conforme a figura 3.1), é possível escolher entre jogar com um ou dois jogadores, no mesmo computador, e escrever um nome para cada (com limite de caracteres). Após o preenchimento da(s) caixa(s) de texto(s), é possível clicar no botão “Confirmar nome(s) e entrar no menu de fases. Se apertado a tecla ESC, volta ao menu principal.

Sobre o personagem jogador (conforme a figura 3.2), é possível controlá-lo usando teclas do teclado, caso haja dois jogadores, a funcionalidade será a mesma para ambos, entretanto as teclas serão diferentes. Portanto, o personagem poderá andar no eixo X, pular e atacar. Se ele sofrer dano, receberá “knockback”, isto é, dependendo da posição dele e do atacante, receberá uma força no eixo X para se afastar do local.

No menu de fases (conforme a figura 4), é necessário escolher em qual fase entrar, ou seja, na primeira fase, “a floresta”, ou na segunda fase, “o castelo”. Ao clicar no botão “Confirmar fase”, o(s) jogador(es) entrarão na fase escolhida. Se apertado a tecla ESC, volta ao menu de jogadores.

Na fase “a floresta” (conforme a figura 5), o(s) jogador(es) é/são criado(s) em uma posição pré-determinada, o(s) qual/quais deve(m) derrotar inimigos de dificuldade fácil (cavaleiros) e médio (mortos-vivos), além de terem cuidado com obstáculos fáceis (plataforma) e médio (barra mágica). Ademais, todos esses personagens devem ficar em cima de plataformas, se não, pela ação da gravidade, ao encostarem na borda inferior da janela, morrem. Se um jogador for criado e ele morrer, o jogo termina e cria a tela de

Figura 1. Menu principal

Figura 2. Menu de ranking

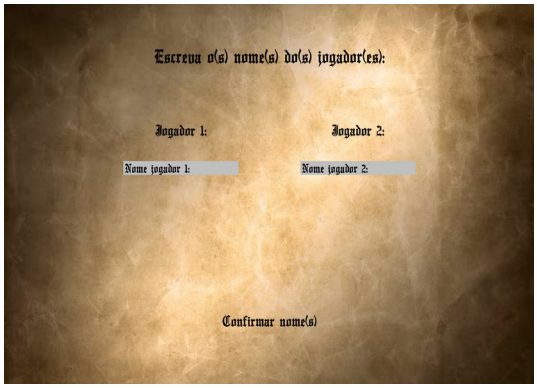

Figura 3.1. Menu de jogadores

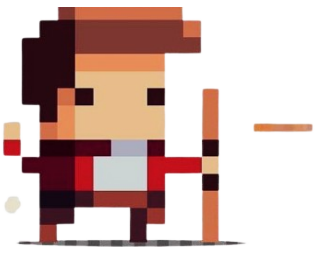

Figura 3.2. Jogador

fim de jogo. Se dois jogadores forem criados e os dois morrerem, o jogo termina e etc.

A gravidade, no jogo, é uma aceleração constante que sempre afeta, até um valor máximo, todas as entidades. No caso dos obstáculos, eles sofrem uma contra força que tem o mesmo valor em módulo da gravidade, a fim de mantê-los estáticos no lugar.

É importante ressaltar que em ambas as fases, as plataformas e os inimigos possuem um valor aleatório de instâncias. Sendo que o valor mínimo para cada é 3 (três) e o valor máximo varia para cada entidade. Assim, na fase "a floresta", o máximo de cavaleiros é 8 (oito), o de mortos-vivos é 5 (cinco), o de barras mágicas é 4 (quatro) e o de plataformas é 7 (sete). Já na fase "o castelo", o máximo de cavaleiros, plataformas (o número mínimo também) e espinhos é diferente para cada tamanho de tela do computador, mas o número máximo de magos é 4 (quatro).

Sobre a mecânica do cavaleiro (conforme a figura 6), é um inimigo que, se colidir com um jogador, dará 1 (um) de dano. Além da movimentação (andar em linha reta), que depende da distância, no eixo X, de sua geração, ou seja, após a entidade percorrer um número de pixel's da janela, ela andará pelo lado contrário e repetirá o processo indefinidamente.

Sobre o morto-vivo (conforme a figura 7), é um inimigo que dará 2 (dois) de dano e que sempre perseguirá o jogador mais próximo, isso se estiver dentro do raio de alcance, visão. Caso o jogador pulasse e estivesse sendo perseguido, a entidade também pularia, isto para se manter no mesmo eixo Y do jogador.

Sobre a barra mágica (conforme a figura 8), como a plataforma, é um obstáculo que se mantêm estático no cenário. Assim, sua funcionalidade se resume a paralisar, por um tempo, um jogador que colida com a barra. Porém, esse efeito ocorre somente nas laterais da barra, caso o jogador encoste na parte de cima, o mesmo não ficará paralisado.

Na fase "o castelo" (conforme a figura 9), o(s) jogador(es) deve(m) matar inimigos de dificuldade fácil (cavaleiros) e difícil (magos), além de terem cuidado com obstáculos fáceis (plataforma) e difícil (espinho). Quanto a tela de fim de jogo, ocorre o mesmo caso da fase "a floresta".

Figura 4. Menu de fases

Figura 5. A floresta

Figura 6. Cavaleiro

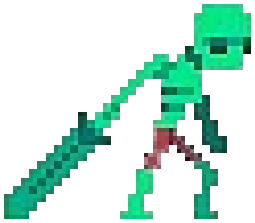

Figura 7. Morto-vivo

Sobre o mago (figura 10), há dois estados diferentes, se ele tiver com mais da metade da vida, dará 1 (um) de dano a um jogador, senão causará 2 (dois) e se moverá mais rápido. Ao contrário do morto-vivo, o mago não pula, mas segue o jogador mais perto no eixo X. Além de poder lançar uma bola de fogo em direção ao jogador mais perto, o qual ao tocá-lo acarreta em um dano equivalente ao do arremessador. O mago somente atirará novamente quando o projétil não estiver na janela.

Sobre o projétil (figura 11), além de dar dano a um personagem, sofre a força da gravidade, e para alcançar o alvo é necessário o atirador aplicar uma força no eixo Y, causando um lançamento oblíquo. Assim, na equação (1) estará descrito como é feito para calcular tal força, considerando o ângulo do lançamento como 45º (quarenta e cinco graus).

Sobre o espinho (figura 12), sendo um obstáculo, fica estático na janela, além de que se um jogador colidir por qualquer lado, sofrerá 1 (um) de dano.

Na tela de fim de jogo (figura 13), o(s) jogador(es) pode(ram) entrar no menu principal e salvar a pontuação obtida ao clicar(em) o botão “Voltar ao menu” ou apertarem a tecla ESC.

Finalmente, nas fases, é possível, ao apertar a tecla ESC, entrar no menu de pause (figura 14). Nela, há duas opções: ou clicar no botão “Retomar” (ou apertar ESC) e voltar para onde todos os personagens estavam, sem perda de pontuação e de memória; ou clicar no botão “Salvar e sair” e salvar a partida, sair da fase e entrar no menu principal.

O salvamento da partida ocorre somente no momento em que o(s) jogador(es) não completaram a fase atual. Quando a partida é retomada, as informações, como as posições, do(s) jogador(es) e das entidades são reinseridas no jogo, permitindo, assim, a continuação da partida.

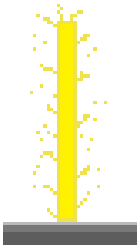

Figura 8. Barra Mágica

Figura 9. O castelo

Figura 10. Mago

Figura 11. Projétil (bola de fogo)

$$\Delta S = \frac{V^2 * \sin(2 * \Theta)}{g}$$

$$\Theta = 45^o$$

$$\Delta S = \frac{V^2 * 1}{g}$$

(1)

$$\Delta S = \frac{(Vx^2 + Vy^2)}{g}$$

$$\Delta S * g = Vx^2 + Vy^2$$

$$\Delta S * g - Vx^2 = Vy^2$$

$$Vy = \sqrt{\Delta S * g - Vx^2}$$

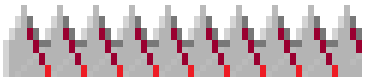

Figura 12. Espinho

Figura 13. Tela de fim de jogo

Figura 14: menu de pause (no castelo)

DESENVOLVIMENTO DO JOGO NA VERSÃO ORIENTADA A OBJETOS

Neste projeto foi-se necessário respeitar e realizar requisitos, criados pelo Prof. Dr. Simão, fundamentais e únicos para a interface gráfica (como menus) e para a mecânica (como o pause e o número de inimigos) do jogo. Em sua maioria, tais critérios funcionaram como uma base, ou seja, esses não limitavam a expansão criativa em recursos e funções adicionais. Como por exemplo, a possibilidade de implementar áudio, som e animações para cada ação que fosse ocorrida durante o uso do software.

Assim sendo, logo abaixo segue a tabela 1, a qual está descrita todos os requisitos, além da situação dos mesmos (podendo variar entre realizado, semi realizado e não realizado). Ademais, na coluna “Implementação” estará escrito como e onde foram implementados tais critérios.

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, escolher ver colocação (*ranking*) de jogadores e escolher demais opções pertinentes (previstas nos demais requisitos).	**REALIZADO**	Cf. As classes Menu, MenuFases, MenuJogadores e MenuRanking e seus respectivos objetos, com suporte da SFML.
2	Permitir um ou dois jogadores com representação gráfica aos usuários do Jogo, sendo que no último caso é para que os dois joguem de maneira concomitante.	**REALIZADO**	Cf. Classe Jogador cujos objetos são agregados em jogo e inicializados cf. classe MenuJogadores.
3	Disponibilizar ao menos duas fases distintas que podem ser jogadas sequencialmente ou selecionadas, via menu, nas quais jogadores tentam neutralizar inimigos por meio de algum artifício e vice-versa.	**REALIZADO**	Cf. classes Fase, Floresta e Castelo, no pacote Fases. Sendo que, por meio do menuFases permite o(s) jogador(es) escolherem a fase.
4	Ter pelo menos três tipos distintos de inimigos, cada qual com sua representação gráfica, sendo que ao menos um deles deve poder lançar projetil contra o(s) jogador(es) e um dos inimigos dever ser um ‘chefão’.	**REALIZADO**	Cf. Classes Mago, Cavaleiro, MortoVivo e MortoVivoThread, sendo o Mago capaz de lançar projétil e o chefe do jogo.
5	Ter a cada fase ao menos dois tipos de inimigos (um deles exclusivo nela) com número aleatório de instâncias, podendo ser várias instâncias (definindo um máximo) e sendo pelo menos 3 instâncias para cada tipo que estiver na fase.	**REALIZADO**	Cf. Classes Floresta e Castelo, as quais criam os inimigos no momento de suas criações.
6	Ter três tipos de obstáculos, cada qual com sua representação gráfica, sendo que ao menos um causa dano em jogador se colidirem.	**REALIZADO**	Cf. classes BarraMagica, Plataforma e Espinho (danoso ao jogador).
7	Ter em cada fase ao menos dois tipos de obstáculos (um deles exclusivo nela) com número aleatório (definindo um máximo) de instâncias (*i.e.*, objetos), sendo pelo menos 3 instâncias por tipo.	**REALIZADO**	Cf. classes Fase, Floresta e Castelo. Sendo que em Fase cria plataformas de Floresta; em Floresta, BarrasMagica; e em Castelo, Plataforma e Espinho
8	Ter em cada fase um cenário de jogo constituído por obstáculos, sendo que parte deles **devem ser** plataformas ou similares, sobre as quais pode haver inimigos e podem subir jogadores. Em cada fase, só poder ter um tipo coincidente de inimigo e um tipo coincidente de obstáculo (que é a plataforma) em relação as demais fases.	**REALIZADO**	Cf. Classes Castelo e Floresta. Sendo que em Castelo e Floresta o obstáculo coincidente é a Plataforma e o inimigo coincidente é o Cavaleiro.
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	**REALIZADO**	Cf. Classe Gerenciador_Colisoes, na qual aplica gravidade em todos as entidades

	deste jogo de plataforma vertical e 2D.		em cada fase.
10	Permitir: (1) salvar nome do usuário, manter/salvar pontuação (incrementada via neutralização de inimigos) do jogador controlado pelo usuário e gerar lista de pontuação (*ranking*). **E** (2) Pausar e **Salvar/Recuperar** Jogada.	**REALIZADO**.	Cf. Classe MenuPause é possível salvar nome do jogador e pontuação obtida. Além de permitir a recuperação da jogada.
Total de requisitos funcionais apropriadamente realizados. *(Cada tópico realizado efetivamente vale 10%)*			**100%** (cem por cento).
Os requisitos dependem em algo uns dos outros, na chamada interdependência de requisitos.			

A fim de complementar a tabela de requisitos, foi feito, por meio de um diagrama de classes UML, a representação de todas as classes e suas inter-relações do jogo. À vista disso, o projeto fora organizado a partir de vários pacotes que englobam a maior parte dos arquivos .h e .cpp. Eles são: Listas, Fases, Menus, Gerenciadores, Entidades, Obstaculos e Personagens.

Então, antes de explicar resumidamente as relações dos principais objetos, é importante destacar o SFML, o qual possibilitou a plena construção do projeto. Sendo uma biblioteca externa multilinguagem que possibilita a construção de uma interface gráfica para os sistemas operacionais Windows, Linux e MacOS[1]. Com ela foi possível "desenhar" objetos na tela de forma mais abstrata possível, ao usar classes e métodos próprios dela, sem precisar utilizar bibliotecas mais complexas.

Dessa forma, continuando a explicação das relações, dada uma classe .h, ela sempre herda a classe abstrata Ente, e quanto mais específica, mais heranças (múltipla e/ou vários níveis) possuiria. Por exemplo, o Cavaleiro herdaria de Inimigo, de Personagem, de Entidade e de Ente.

Portanto, a classe Jogo, por meio de agregação, conhece todos os menus (com exceção do menuPause), os jogadores, as fases, o Gerenciador_Gráfico, a TelaFimDeJogo e o Ranking. A classe MenuJogadores, em contraste aos demais menus, conheceria o Jogo, ou seja, ela é agregada de Jogo, mas é bidirecional.

Ademais, Castelo e Floresta herdam de Fase, mas também agregam fracamente Inimigos (ocorre em Floresta), Cavaleiro, Plataforma e Mago (em Castelo). Também é importante ressaltar que em Fase ocorrem agregações com Jogador, com os gerenciadores, com ListaEntidades e com hud's.

Em Cavaleiro, MortoVivo e Mago, possuem uma herança comum a Inimigo, mas, ao contrário dos demais, Mago possui uma agregação forte com Projetil, já que é o único personagem capaz de atacar a longas distâncias. Como já foi dito na seção anterior, o Projetil utiliza uma derivação da equação de um lançamento oblíquo, para que possa funcionar plenamente junto com gravidade (conceitos reutilizados da física do Ensino Médio). Além disso, MortoVivo e Mago possuem uma funcionalidade extra, se comparado ao Cavaleiro, isto é, perseguir o jogador. Já MortoVivo apresenta a mecânica do pulo, presente também em Jogador.

Em Plataforma, BarraMagica e Espinho, possuem uma herança comum a Obstaculo, porém BarraMagica e Espinho possuem funções extras. Portanto, BarraMagica paralisa o Jogador, por meio do método "obstacular", e Espinho danifica somente o Jogador através do mesmo método. Diante disso, nenhum dos obstáculos possuem agregação ou associação a Jogador, simplificando a modelagem do código.

Em Jogador, herda de Personagem, mas possui agregações fortes e associações com objetos de classes da biblioteca SFML, como Sprite, Texture e Clock. Além disso, apresenta métodos exclusivos como mover, pular e atacarInimigo (realizado através do Gerenciador_Colisoes).

No pacote Gerenciadores, o Gerenciador_Grafico e o Gerenciador_Colisoes herdam de Ente e não conhecem uma a outra. Porém, enquanto no Gerenciador_Grafico possui associação com a classe RenderWindow da SFML, o Gerenciador_Colisoes possui agregações com Inimigo, Obstaculo, Projetil e Jogador.

Em Ranking (classe agregada fortemente a MenuRanking e a Jogo), não possui herança, mas é a classe responsável pela recuperação dos pontos e nomes dos jogadores. Nela, o MenuRanking

[1] https://www.sfml-dev.org

apenas conhece, acessa as informações, enquanto o Jogo pode modificar as informações guardadas, mediante métodos de Ranking. Também, é interessante informar que toda vez uma informação é modificada no Ranking, ocorre uma reordenação nos dados no final de cada fase.

Por fim, logo abaixo segue o diagrama de classes UML (completo) correspondente ao projeto.

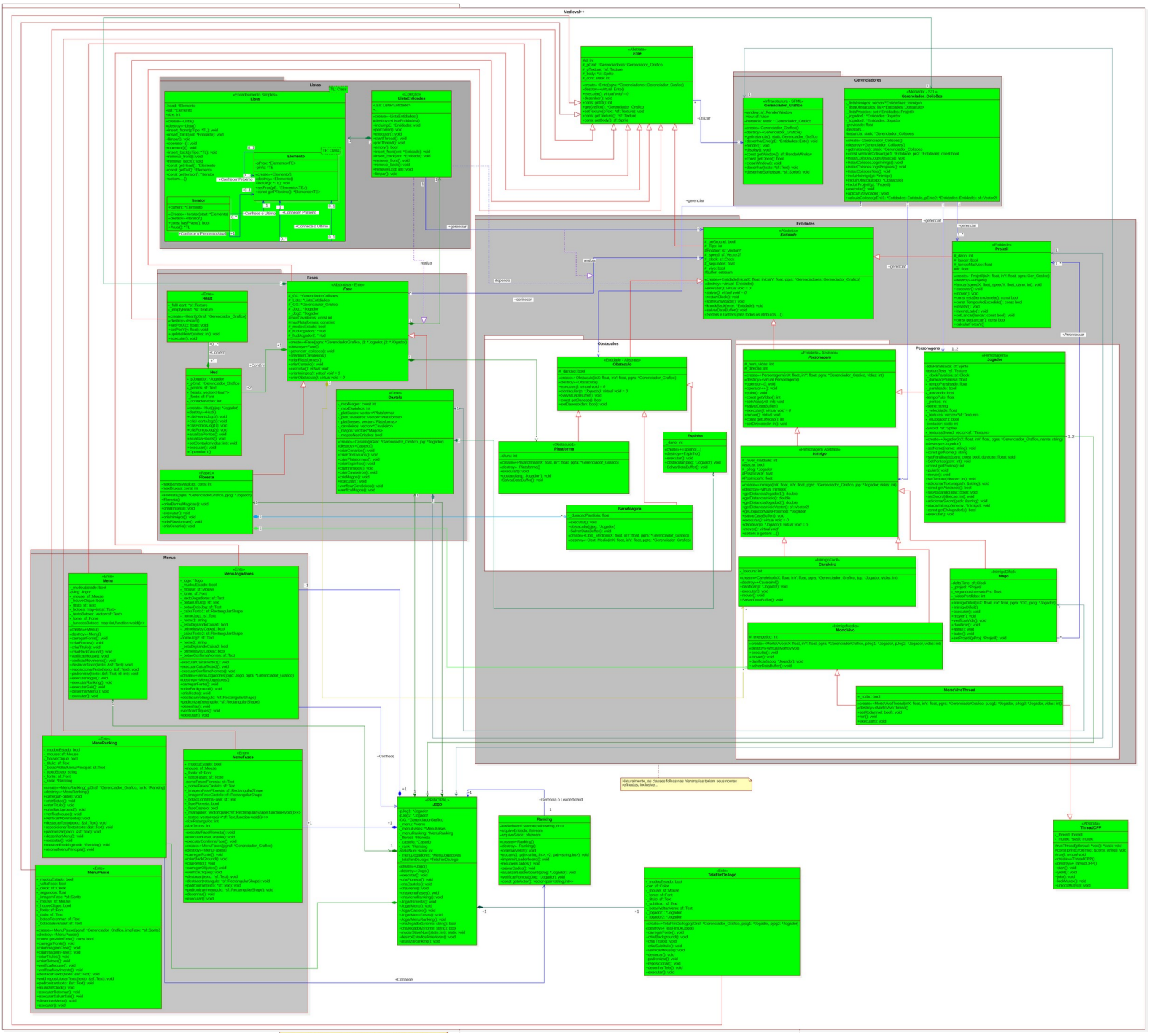

Figura 2. Diagrama de Classes de base em UML

TABELA DE CONCEITOS UTILIZADOS E NÃO UTILIZADOS

Nesta seção, é demonstrado todos os conceitos usados ou não no jogo na tabela 2. Assim, tais conceitos, feitos pelo Prof. Dr. Simão, são imutáveis, isto é, como na tabela 1 (requisitos), não

poderiam ser mudados. Portanto, o cumprimento ou não dos mesmos poderiam afetar o bom funcionamento do projeto.

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	– Na maioria dos .h e .cpp, como nas classes MenuPause e Jogador nos *namespaces* Entidades e Menus. – Classe, Objetos, Atributos e Métodos foram utilizados porque são conceitos elementares na orientação a objetos.
1.2 &	– Métodos (com retorno *const* e parâmetro *const*). – Construtores (sem/com parâmetros) e destrutores	Sim	– Na maioria dos .h e .cpp, como nas classes Gerenciador_Colisões e MenuPause nos *namespaces* Gerenciadores e Menus. – A constância pertinente evita mudanças equivocadas, construtores são mandatórios para inicializar atributos e destrutores pertinentes para finalizações como desalocações.
1.3	– Classe Principal.	Sim	- Main.cpp & Principal.h/.cpp – Uma classe Principal é mais 'purista' em termos de OO.
1.4	– Divisão em .h e .cpp.	Sim	– No desenvolvimento como um todo, como nas classes Inimigo e MenuRanking nos *namespaces* Entidades e Menus. – Permite organizar as classes e afins que compõem o sistema.
2	**Relações de:**		
2.1	– Associação direcional. & – Associação bidirecional.	Sim	– Em vários dos .h e .cpp, como nas classes MenuFase e MenuJogador (no uso de classes da SFML e ao conhecer a classe Jogo) no *namespace* Menus. – Associação direcional e bidirecional melhoram a lógica do sistema, quando os atributos de uma classe fazem necessariamente parte dela.
2.2 &	– Agregação via associação. – Agregação propriamente dita.	Sim	– Em vários dos .h e .cpp, como nas classes Jogo (agregação propriamente dita) e MenuRanking (agregação via associação), no *namespace* Menus. – Tais agregações são essenciais para respeitar a Orientação a Objetos, mas também são úteis para melhorar a lógica do código.
2.3 &	– Herança elementar. – Herança em vários níveis.	Sim	– Em vários dos .h e .cpp, como nas classes Inimigo e Castelo nos *namespaces* Entidades e Fases. – Por meio da herança elementar e em vários níveis, o código fica mais genérico e possibilita o polimorfismo.
2.4	– Herança múltipla.	Sim	– Precisamente no .h e .cpp, da classe MortoVivoThread. – Por meio da herança múltipla, além do código ficar mais genérico e possibilitar o polimorfismo, é possível unir classes bases de pacotes diferentes.
3	**Ponteiros, generalizações e exceções**		
3.1	– Operador *this* para fins de relacionamento bidirecional	Sim	– Precisamente nos .h e .cpp, das classes MenuJogadores e Jogo. – Por meio do operador *this,* é possível implementar relações de associação e agregação bidirecionais.
3.2	– Alocação de memória (*new* & *delete*).	Sim	– Precisamente nos .h e .cpp, das classes Jogo e ListaEntidade. – Com a alocação de memória, é possível inicializar objetos tardiamente, seja por custo de memória, seja por funcionalidade extra.
3.3	– Gabaritos/*Templates* criada/adaptados pelos autores para Listas.	Sim	– Precisamente nos .h e .cpp das classes ListaEntidade, Lista e Elemento. – Com o uso de Gabaritos, é possível generalizar o uso de uma lista, por exemplo, para ser usado para diferentes classes.
3.4	– Uso de Tratamento de Exceções (*try catch*).	Sim	– Em vários dos .h e .cpp, como a classe Ranking e a classe Espinho do *namespace* Obstaculos. – Com o uso do try catch é possível controlar exceções previsíveis e adotar uma contramedida.
4	**Sobrecarga de:**		

4.1	– Construtoras e Métodos.	Sim	– Precisamente .h e .cpp, como na classe Ente (sobrecarga de construtora e de método) e em Gerenciador_Grafico (somente sobrecarga de método), no *namespace* Gerenciadores. – Tais ferramentas flexibilizam a implementação de diferentes métodos para um mesmo "nome" da função.
4.2	– Operadores (2 tipos de operadores pelo menos).	Sim	Foi usado o *operator*-- na classe Personagem (para retirar vida) e o *operator*-- na classe Lista (para remover elementos).
---	**Persistência de Objetos (via arquivo de texto ou binário)**		
4.3	– Persistência de Objetos.	Sim	– Precisamente no .h e .cpp da classe Ranking e em vários métodos de classes como Menu e Entidade. – A persistência de objetos permite o reúso de dados de um mesmo software, mesmo depois de finalizado o programa. – Realizado mediante de arquivo de texto (*.txt*) para salvar pontuação e nome de jogadores, e também através de arquivo .txt para recuperar jogada.
4.4	– Persistência de Relacionamento de Objetos.	Sim	- Presente no formato de métodos da classe Castelo. - A persistência de relacionamento de objetos, além de permitir salvar dados, possibilita a conservação de informações que são interdependentes (como classes) de umas a outras.
5	**Virtualidade:**		
5.1	– Métodos Virtuais Usuais.	Sim	– Em vários .h e .cpp, como nas classes Espinho e Fase dos namespaces Obstaculo e Fases. – Com os métodos virtuais é possível utilizar técnicas como o Polimorfismo, essencial para o OO.
5.2	– Polimorfismo.	Sim	– Em alguns .h e .cpp, como na classe Gerenciador_Colisoes e ListaEntidades dos namespaces Gerenciadores e Lista. – Com as tabelas virtuais, basta conhecer a classe base para executar métodos mais avançados.
5.3	– Métodos Virtuais Puros / Classes Abstratas.	Sim	– Em vários .h e .cpp, como nas classes Ente e Inimigo do namespace Personagens. – Com as classes abstratas é possível utilizar o polimorfismo, se necessário.
5.4	– Coesão/Desacoplamento efetiva e intensa com o apoio de padrões de projeto (mais de 5 padrões).	Não	
6	**Organizadores e Estáticos**		
6.1	– Espaço de Nomes (*Namespace*) criada pelos autores.	Sim	– Vários pacotes (namespaces) criados, como Menus e Fases. – São importantes para organizar o projeto, melhorando a modularização do código
6.2	– Classes aninhadas (*Nested*) criada pelos autores.	Sim	– Particularmente em um .h e .cpp, na classe Lista do *namespace* Listas. – As classes aninhadas são importantes quando uma classe está estritamente interligada a somente uma outra.
6.3	– Atributos estáticos e métodos estáticos.	Sim	– Em alguns .h e .cpp, como nas classes Jogo e Gerenciador_Grafico do *namespace* Gerenciadores. – Os atributos e métodos estáticos são úteis, quando somente uma instância de uma classe será criada. Assim, não sendo necessário uma relação bidirecional.
6.4	– Uso extensivo de constante (*const*) parâmetro, retorno, método...	Sim	– Na maioria dos .h e .cpp, como nas classes Gerenciador_Colisões e MenuPause nos *namespaces* Gerenciadores e Menus. - "A constância pertinente evita mudanças equivocadas, construtores são mandatórios para inicializar atributos e destrutores pertinentes para finalizações como desalocações" (como já dito anteriormente).
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 alguns .h e .cpp, como nas classes Jogador e Gerenciador_Colisoes, nos *namespaces* Personagens e Gerenciadores. – O uso de String, Vector e List facilitam o armazenamento de dados durante a execução do programa, sem precisar implementar do "zero".

7.2	– Pilha, Fila, Bifila, Fila de Prioridade, Conjunto, Multiconjunto, Mapa **OU** Multi-Mapa.	Sim	– Em alguns .h e .cpp, como nas classes Menu e Gerenciador_Colisoes dos *namespaces* Menus e Gerenciadores. – O uso de contêineres mais complexos permite a criação de funções extras ou até mesmo mais eficientes.
---	**Programação concorrente**		
7.3	– *Threads* (Linhas de Execução) no âmbito da Orientação a Objetos, utilizando Posix, C-Run-Time **OU** Win32API ou afins.	Sim	– Particularmente em um .h e .cpp, isto é, na classe ThreadCPP. – O uso de *threads* permite execuções paralelas, fazendo do projeto um *multithreading.*
7.4	– *Threads* (Linhas de Execução) no âmbito da Orientação a Objetos com uso de Mutex, Semáforos, **OU** Troca de mensagens.	Sim	– Particularmente em um .h e .cpp, isto é, na classe ThreadCPP. – O uso do Mutex impede que uma classe interfira na execução de outra.
8	**Biblioteca Gráfica / Visual**		
8.1 **&**	– Funcionalidades Elementares. – Funcionalidades Avançadas como: tratamento de colisões e duplo *buffer*	Sim	Funcionalidades elementares: verificação posição e clique do mouse, controle do personagem (através de teclas do teclado), caixas de textos e controle parcial de fluxo. Funcionalidades avançadas: gravidade, lançamento oblíquo, detecção de colisões, *hud,* salvamento de pontuação e nome de jogadores.
8.2 **OU**	– Programação orientada e evento efetiva (com gerenciador apropriado de eventos inclusive, via padrão de projeto *Observer*) em algum ambiente gráfico. – *RAD – Rapid Application Development* (Objetos gráficos como formulários, botões etc).	Sim	Foi implementado mecânicas de botões, caixas de texto e caixas de seleção. Os botões foram usados em todos os menus e na TelaFimDeJogo, as caixas de textos no MenuJogadores, e as caixas de seleção no MenuFases. Cada um com funcionalidades extras, como limite de caracteres (caixa de texto) e destacar o botão ou a caixa de seleção, conforme o usuário interagi-se com eles.
---	**Interdisciplinaridades via utilização de Conceitos de <u>Matemática Contínua e/ou Física</u>.**		
8.3	– Ensino Médio Efetivamente.	Sim	Conceitos de física: gravidade, lançamento oblíquo, plano cartesiano, colisão de posições. Conceitos de matemática: equação de Pitágoras.
8.4	– Ensino Superior Efetivamente.	Não	
9	**Engenharia de Software**		
9.1	– Compreensão, melhoria e rastreabilidade de cumprimento de requisitos.	Sim	Realizado mediante o uso de testes dentro da dupla e o uso das tabelas de conceitos e de requisitos para acompanhar o andamento do projeto.
9.2	– Diagrama de Classes em *UML*.	Sim	Realizado mediante o uso do Software StarUML.
9.3	– Uso efetivo e intensivo de padrões de projeto *GOF*, *i.e.*, mais de 5 padrões.	Não	Implementado os seguintes padrões: *Iterator e Singleton.*
9.4	– Testes à luz da Tabela de Requisitos e do Diagrama de Classes.	Sim	Realizado mediante o uso de testes feitos pela dupla.
10	**Execução de Projeto**		
10.1 **&**	– Controle de versão de modelos e códigos automatizados (via github). – Uso de alguma forma de cópia de segurança (*i.e.*, *backup*).	Sim	Realizado através do repositório GITHUB. Abaixo segue o link do repositório do projeto **<u>https://github.com/GabFMM/Jogo_TecProg.git</u>**

10.2	– Reuniões com o professor para acompanhamento do andamento do projeto.	Sim	Realizadas 2 (duas) reuniões nos dias 10 e 17/12.
10.3	– Reuniões com monitor da disciplina para acompanhamento do andamento do projeto.	Sim	Gabriel – 1 (uma) reunião com o PETECO no dia 12/12/2024 e 1 (uma) reunião com o monitor de duração 1 (uma) hora no dia 06/02/2025. Yago – 2 (duas) reunião com o monitor de duração 1 (uma) hora nos dias 06 e 11/02/2025.
10.4 &	– Escrita do trabalho e feitura da apresentação – Revisão do trabalho escrito de outra equipe e vice-versa.	Sim	Gabriel Felipe Miguel Machado realizou a escrita do relatório. E a equipe formada por João Antonio Teixeira Sucupira e João Gabriel Klug Teixeira, realizou a revisão do presente documento.
Total de conceitos apropriadamente utilizados.			**92.5%** (noventa e dois por cento e meio).

DISCUSSÃO E CONCLUSÕES

Neste trabalho, ao longo de todo o seu desenvolvimento, desde a interpretação dos requisitos, dos conceitos e do diagrama de classes UML base, foi necessário uma coesão dos objetivos de cada um dos integrantes da dupla para, assim, formular um fim único para o jogo. Dessa forma, é imprescindível notar o papel da comunicação entre os membros de uma mesma equipe, ou seja, a discussão de ideias novas ou uma sugestão de uma remodelação do planejamento, durante o processo, é vital para a sustentabilidade do projeto.

Além do papel da interação, troca de ideias, formular um plano de ação previamente da codificação, mostrou-se ser um ato importante a ser feito em qualquer esquema que, porventura, seja feito ou não em um grupo de indivíduos. Portanto, o planejamento, isto é, definir metas, delegar funções, estabelecer meios de comunicação entre a equipe, modelar um meio, um modo, de como alcançar os objetivos, entre outros, é, e sempre será, uma prática essencial a ser realizada antes e, também, durante o processo das atividades.

Assim sendo, os desenvolvedores podem concluir que o resultado obtido, considerando o prazo estipulado, é aceitável e satisfatório. Entretanto, com as habilidades e os conhecimentos adquiridos, foi notável a potencialidade do jogo a ser expandido, isto é, com estímulos e tempo suficientes, o projeto é capaz de ser melhorado a um patamar que, no momento, talvez nem seja possível visualizar totalmente.

Por fim, os integrantes compreenderam a linguagem de programação C++, o paradigma Orientação a Objetos e a biblioteca externa SFML. Ademais, ao obterem experiência nessa atividade grande e desafiadora, a dupla estará melhor preparada, tanto para o mercado de trabalho, tanto para áreas de ciências, pesquisas e especializações.

DIVISÃO DO TRABALHO

Nesta seção estará descrito a separação das funções da tabela de conceitos entre cada integrante. Assim, segue abaixo a tabela 4, a qual estará explícito os conceitos e o responsável correspondente. Ademais, vale ressaltar que haverá situações nas quais ambos ou não da equipe atuaram no mesmo tópico.

Tabela 4. Lista de Atividades e Responsáveis.

N.	Atividades	Responsáveis
1	**Elementares: AMBOS**	
1.1	– Classes, objetos. & - Atributos (privados), variáveis e constantes. & – Métodos (com e sem retorno).	Yago e Gabriel
1.2	– Métodos (com retorno *const* e parâmetro *const*). & – Construtores (sem/com parâmetros) e destrutores	Yago e Gabriel
1.3	– Classe Principal.	Yago e Gabriel
1.4	– Divisão em .h e .cpp.	Yago e Gabriel
2	**Relações de: AMBOS**	
2.1	– Associação direcional. & - Associação bidirecional.	Yago e Gabriel
2.2	– Agregação via associação. & - Agregação propriamente dita.	Yago e Gabriel
2.3	– Herança elementar. & - Herança em vários níveis.	Yago e Gabriel
2.4	– Herança múltipla.	Yago
3	**Ponteiros, generalizações e exceções: AMBOS**	
3.1	– Operador *this* para fins de relacionamento bidirecional.	Yago e Gabriel
3.2	– Alocação de memória (*new* & *delete*).	Yago e Gabriel
3.3	– Gabaritos/*Templates* criada/adaptados pelos autores para Listas.	Yago e Gabriel
3.4	– Uso de Tratamento de Exceções (*try catch*).	Yago
4	**Sobrecarga de: MAIS YAGO**	
4.1	– Construtoras e Métodos.	Mais Yago que Gabriel
4.2	– Operadores (2 tipos de operadores pelo menos)	Yago
---	**Persistência de Objetos (via arquivo de texto ou binário): YAGO**	
4.3	– Persistência de Objetos.	Yago
4.4	– Persistência de Relacionamento de Objetos.	Yago
5	**Virtualidade: AMBOS**	
5.1	– Métodos Virtuais Usuais.	Yago e Gabriel
5.2	– Polimorfismo.	Yago e Gabriel
5.3	– Métodos Virtuais Puros / Classes Abstratas.	Yago e Gabriel
5.4	– Coesão/Desacoplamento efetiva e intensa com o apoio de padrões de projeto (mais de 5 padrões).	Não realizado
6	**Organizadores e Estáticos: AMBOS**	
6.1	– Espaço de Nomes (*Namespace*) criada pelos autores.	Yago e Gabriel
6.2	– Classes aninhadas (*Nested*) criada pelos autores.	Yago e Gabriel
6.3	– Atributos estáticos e métodos estáticos.	Yago e Gabriel
6.4	– Uso extensivo de constante (*const*) parâmetro, retorno, método...	Yago e Gabriel
7	**Standard Template Library *(STL)* e String OO: AMBOS**	
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)	Yago e Gabriel
7.2	– Pilha, Fila, Bifila, Fila de Prioridade, Conjunto, Multiconjunto, Mapa **OU** Multi-Mapa.	Yago e Gabriel
---	**Programação concorrente: AMBOS**	
7.3	– *Threads* (Linhas de Execução) no âmbito da Orientação a Objetos, utilizando Posix, C-Run-Time **OU** Win32API ou afins.	Yago
7.4	– *Threads* (Linhas de Execução) no âmbito da Orientação a Objetos com uso de Mutex, Semáforos, **OU** Troca de mensagens.	Yago
8	**Biblioteca Gráfica / Visual: AMBOS / MAIS GABRIEL**	
8.1	– Funcionalidades Elementares. **&** – Funcionalidades Avançadas como: tratamento de colisões e duplo *buffer*	Yago e Gabriel
8.2	- Programação orientada e evento efetiva (com gerenciador apropriado de eventos inclusive, via padrão de projeto *Observer*) em algum ambiente gráfico. **OU** – *RAD – Rapid Application Development* (Objetos gráficos como formulários, botões etc).	Gabriel
---	**Interdisciplinaridades via uso de Conceitos de Matemática Contínua e/ou Física: AMBOS**	
8.3	– Ensino Médio Efetivamente.	Yago e Gabriel
8.4	– Ensino Superior Efetivamente.	Não realizado
9	**Engenharia de Software: MAIS YAGO**	

9.1	– Compreensão, melhoria e rastreabilidade de cumprimento de requisitos.	Yago e Gabriel
9.2	– Diagrama de Classes em *UML*.	Yago
9.3	– Uso efetivo e intensivo de padrões de projeto *GOF*, *i.e.*, + de 5 padrões.	Não realizado
9.4	– Testes à luz da Tabela de Requisitos e do Diagrama de Classes.	Yago e Gabriel
10	**Execução de Projeto: AMBOS**	
10.1	– Controle de versão de modelos e códigos automatizados (via github). & – *Uso de alguma forma de cópia de segurança (i.e., backup).*	Yago e Gabriel
10.2	– Reuniões com o professor para acompanhamento do andamento do projeto.	Yago e Gabriel
10.3	– Reuniões com monitor da disciplina para acompanhamento do andamento do projeto.	Yago e Gabriel
10.4	– Escrita do trabalho e feitura da apresentação **&** – Revisão do trabalho escrito de outra equipe e vice-versa.	Gabriel

Aqui consta o quanto cada um da equipe trabalhou em termos de realização e de colaboração, segundo os dados fornecidos na tabela 4:
– Gabriel aproximadamente trabalhou em 81% das atividades as realizando ou colaborando nelas efetivamente.
– Yago aproximadamente trabalhou em 94% das atividades as realizando ou colaborando nelas efetivamente.

AGRADECIMENTOS PROFISSIONAIS

A equipe agradece abertamente pelo apoio do grupo PETECO pelo fornecimento de informações e pelas reuniões realizadas presencialmente sobre o modo de realização do trabalho; pela ajuda, mediante encontros on-line, dada pelo monitor Daniel Zagroba da disciplina de Técnicas de Programação; pelos vídeos gravados e distribuídos, pelo Matheus Augusto Burda, sobre cálculos vetoriais; e pela revisão do presente relatório, feita pela equipe formada por João Antonio Teixeira Sucupira e João Gabriel Klug Teixeira.

REFERÊNCIAS CITADAS NO TEXTO

[1] SIMÃO, J. M. Site das Disciplina de Fundamentos de Programação 2, Curitiba – PR, Brasil, Acessado em 06/02/2025, às 21:37 - https://pessoal.dainf.ct.utfpr.edu.br/jeansimao/Fundamentos2/Fundamentos2.htm.

[2] Site oficial da biblioteca externa SFML. Acessado em 06/02/2025, às 21:39 - https://www.sfml-dev.org.

[3] BURDA, A. M. Canal do educativo do Youtube sobre a biblioteca SFML, Acessado em 06/02/2025, às 21:45 - https://www.youtube.com/@burdacanal9100.

REFERÊNCIAS UTILIZADAS NO DESENVOLVIMENTO

[A] Site oficial da documentação da biblioteca externa SFML. Acessado em 06/02/2025, às 21:39 – https://www.sfml-dev.org/documentation/3.0.0/.