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

Nícolas Auersvalt Marques, Isabela Bella Bortoleto

nicolassauersvalt@alunos.utfpr.edu.br, isabelabortoleto@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: Engenharia da Computação / Sistemas de Informação

### Universidade Tecnológica Federal do Paraná - UTFPR

Avenida Sete de Setembro, 3165 - Curitiba/PR, Brasil - CEP 80230-901

**Resumo** – A disciplina de Técnicas de Programação exige o desenvolvimento de um *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 **GravityRooms**, no qual o jogador enfrenta inimigos na Thalos I (espaçonave). 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 Classe Abstrata, Polimorfismo, Gabaritos, Persistências de Objetos por Arquivos, Sobrecarga de Operadores e Biblioteca Padrão de Gabaritos (*Standard Template Library* - *STL*). Depois da implementação, os testes e uso do jogo feitos pelos próprios desenvolvedores demonstraram sua funcionalidade conforme os requisitos e o modelagem elaborada. Por fim, salienta-se que o desenvolvimento em questão permitiu cumprir o objetivo de aprendizado visado.

**Palavras-chave ou Expressões-chave:** Artigo-Relatório para o Trabalho em Técnicas de Programação, Trabalho Acadêmico Voltado a Implementação em C++.

# INTRODUÇÃO

Para a melhor compreensão dos conteúdos abordados na disciplina de Técnicas de Programação, tornou-se necessário o desenvolvimento de um jogo, fundamentado nos princípios básicos do ciclo de Engenharia de *Software*, com ênfase na modelagem em *UML*. Esse projeto visa consolidar os conhecimentos adquiridos ao longo do curso, proporcionando uma aplicação prática dos conceitos estudados.

Como objeto de estudo e implementação, propõe-se a criação de um jogo deplataforma com estética pixelada, desenvolvido utilizando a biblioteca *SFML* em C++. Essa ferramenta possibilita a construção de um ambiente gráfico adequado para experimentação e testes no contexto da disciplina, permitindo a aplicação dos fundamentos da programação orientada a objetos.

O desenvolvimento do jogo segue um regulamento específico, exigindo a interpretação e adaptação dos conceitos da programação orientada a objetos, bem como o uso da *UML* para modelagem do sistema. Além da implementação técnica, o projeto requer planejamento em grupo e uma análise detalhada dos requisitos, promovendo uma abordagem estruturada e colaborativa. O processo de desenvolvimento segue uma versão simplificada do ciclo de Engenharia de *Software*, incluindo a compreensão dos requisitos, modelagem por meio de diagramas de classes *UML*, implementação em C++ e testes do *software*.

Por fim, para contextualizar este trabalho, inicialmente será apresentado um breve panorama sobre o tema do jogo. Em seguida, será exposta a explicação técnica geral do projeto, detalhando os métodos adotados, as ferramentas utilizadas e as seções subsequentes do relatório.

# EXPLICAÇÃO DO JOGO EM SI

*GravityRooms* é um jogo que adota, como temática central, uma abordagem filosófica da metafísica, com ênfase no pensamento de Martin Heidegger. Através da escolha de inimigos e do nível de dificuldade associado a cada um, o jogo propõe reflexões sobre questões fundamentais, como "o que é o ser?", "o que define a humanidade?" e "robôs possuem consciência?". Dessa forma, a proposta visa estimular uma discussão relevante para os tempos atuais.

O jogo conta com inimigos de quantidade aleatória e limitada, cada um com tipos exclusivos e níveis de dificuldade variados, representando diferentes aspectos da reflexão filosófica proposta. O Ciborgue (um humano com partes robóticas) é o inimigo de menor dificuldade e levanta o questionamento: "até que ponto um humano pode se tornar artificial sem deixar de ser humano?". O Andróide (um robô que imita um humano) apresenta um nível de dificuldade intermediário e instiga a indagação: "um robô pode ter consciência?". Por fim, o Clone (uma cópia exata de um humano) é o inimigo mais desafiador, atacando com projéteis e suscitando uma reflexão ética: "um clone é um ser humano?", e ambos os inimigos se apresentam em uma quantidade aleatória por fase e pontos de vidas específicos.



Figura 1: Dificuldades relativas. Sprites feitas no PixelArt.

Além dos inimigos, o jogo apresenta obstáculos variados, gerados de forma aleatória e limitados em número. Entre eles, há plataformas, espinhos (que podem eliminar o jogador), espinhos retráteis (que ativam e desativam ao longo do tempo) e regiões gravitacionais, que exercem uma forte atração, resultando na morte do personagem

O jogador assume o papel de um tripulante da nave, sendo capaz de atirar nos inimigos, assim como seu clone. Além disso, pode pular, mas possui pouca resistência e munição limitada, tornando a estratégia um elemento essencial para a progressão no jogo.

Todos os inimigos presentes no jogo perseguem o jogador, possuindo diferentes quantidades de vida, proporcionais ao nível de dificuldade. O jogador pode ser controlado pelas setas do teclado (**← → ↓**), enquanto o comando para pular é acionado pela tecla *Enter*, e para atirar, pela tecla Q. O segundo jogador pode ser ativado pressionando M, sendo controlado pelas teclas W, A, S, D, com o disparo realizado pela tecla Z. Para acessar o menu de pausa, utiliza-se a tecla *Esc*, e todas as confirmações dentro do jogo são efetuadas com a tecla *Enter*.

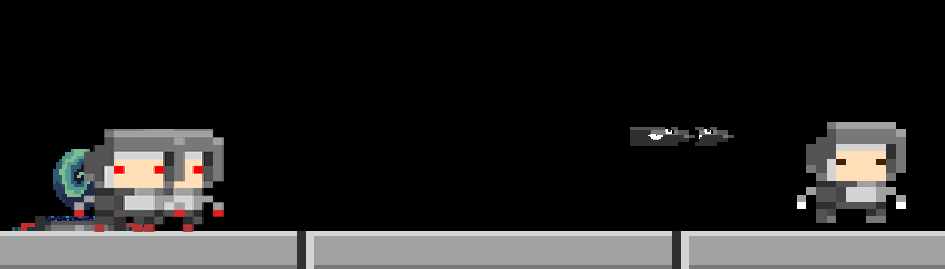
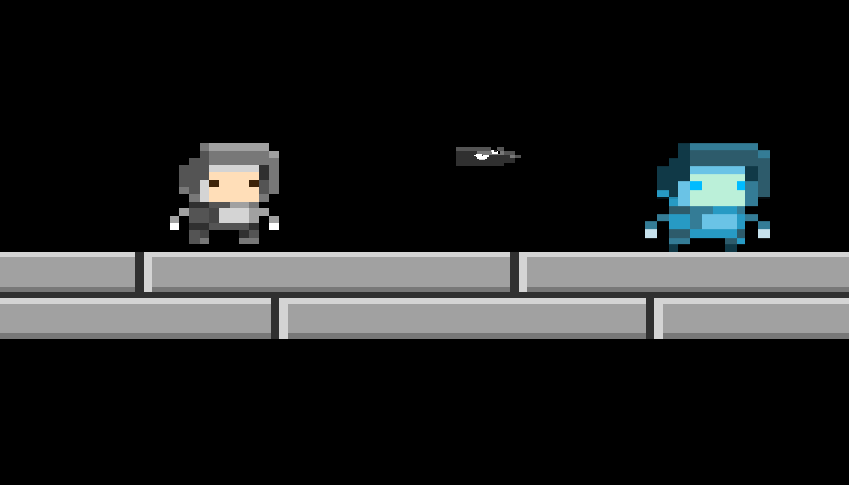


Figura 2 e 3: Tripulante disparando contra Clone e Chefes disparando projéteis contra o tripulante.

A movimentação dos personagens respeita a Função Horária da Posição:

Sua demonstração detalhada pode ser encontrada no **Apêndice A.**

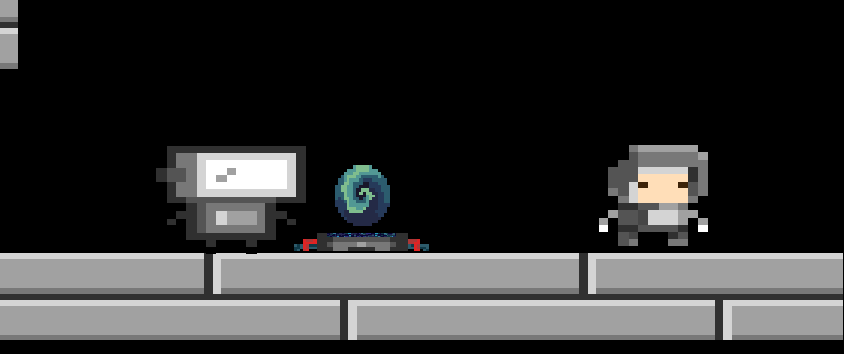


Figura 4 e 5: Androide perseguindo o Tripulante e o segundo jogador disparando projétil.

Além do espinho, obstáculo que causa dano no jogador, na figura 4, é possível notar o objeto Centro Gravitacional, o qual possui uma lógica específica. No momento em que o Tripulante colide com ela, é calculado um dano relativo à velocidade final do jogador. Para o cálculo, segue-se:

A demonstração detalhada pode ser encontrada no **Apêndice B**.

Os espinhos retráteis, quando ativos, ficam com cor vermelha, causando assim dano no jogador. Já quando ficam inativos, ficam com cor verde, não causando dano ao jogador:



Figura 6 e 7: Espinho verde inativo, não causa dano, e vermelho ativo, causando dano.

Na tela de menu, é possível selecionar a fase desejada utilizando as setas (**←↑→↓)** e teclando *Enter*. Cada fase possui um inimigo e um obstáculo exclusivo.



Figura 7: Menu principal e suas opções.

Ao morrer, o jogador pode digitar o nome e teclar *Enter* para ser salvo e comparado no *ranking* final. O salvamento polimórfico escreve em um arquivo de texto no formato *json.*

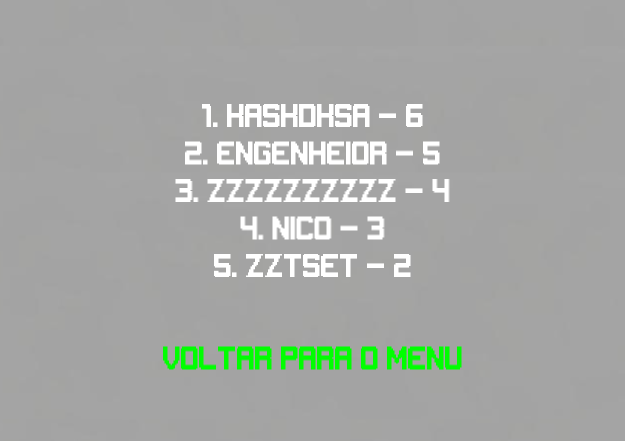


Figura 8 e 9: Tela de fim de jogo e campo de texto e o Ranking e suas posições.

Ao ir no menu *Ranking*, é possível ver os antigos jogadores e seus respectivos pontos. Ao fechar e abrir o jogo novamente, é certo de que os inimigos e obstáculos estarão mudados, seja em quantidade ou posições dentre as possíveis no mapa.

# DESENVOLVIMENTO DO JOGO NA VERSÃO ORIENTADA A OBJETOS

A prática envolve a análise, projeto e implementação de um jogo de plataforma em *C++*, seguindo a orientação a objetos com ênfase em coesão e desacoplamento. A modelagem deve empregar o Diagrama de Classes *UML*, fundamentado em um modelo genérico previamente definido.

A tabela de requisitos funcionais é essencial para formalizar e organizar as funcionalidades esperadas do jogo, garantindo clareza, projeto e implementação.

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. Pacote Menu, com as classes Menu e Menu Principal 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. Pacote Personagens, com a classe Tripulante, cujos objetos são agregados em Fase, podendo ser um ou dois jogadores se desejado. |
| 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. Pacote Fases, com as classes Fase, Laboratorio e Nave, sendo jogadas sequencialmente após matar todos os inimigos, acessando via o menu principal ou carregando o jogo já salvo. |
| 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. pacote Personagens, com as classes Ciborgue, Clone e Androide. Sendo Clone o chefão e único que lança projéteis. |
| 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. A classe Fase gera inimigos aleatoriamente, garantindo pelo menos 3 instâncias de cada tipo presente. Na fase Laboratório, são gerados os inimigos ciborgue (exclusivo dessa fase) e androide. Na fase Nave, são gerados os inimigos clone (exclusivo dessa fase) e androide. |
| 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. Pacote Obstaculos, com as classes Plataforma, Espinho (dano), Centro\_Gravidade (dano) e Espinho Retrátil (dano). |
| 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. A classe Fase gera aleatoriamente os obstáculos. Na fase Laboratório, os obstáculos são espinhos e espinhos retráteis, sendo o espinho exclusivo dessa fase. Já na fase Nave, os obstáculos gerados são espinhos retráteis e centro de gravidade, com centro de gravidade sendo exclusivo dela. |
| 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. Pacote Fases, que agrega listas de inimigos e obstáculos. Classe Gerenciador\_Colisoes para as colisões e Tripulante. |
| 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. | **REALIZADO** | Cf. Pacote Personagens e Gerenciadores. Classe Gerenciador\_Colisoes, Gerenciador\_Fisico, Personagem, Projetil, Tripulante e Inimigo. |
| 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. Pacote Menus. Classes MenuGameOver, Ranking, Registry e Save. |
| **Total de requisitos funcionais apropriadamente realizados.**  *(Cada tópico realizado efetivamente vale 10%)* | | | 100% |
| Os requisitos dependem em algo uns dos outros, na chamada interdependência de requisitos. | | | |

Para uma compreensão mais clara do projeto do jogo, a exemplificação por meio da *UML* é essencial, pois proporciona maior visibilidade sobre métodos, classes, relações e suas nuances estruturais.

# 

# Figura 9. Diagrama de Classes de base em UML – Finalizado

# TABELA DE CONCEITOS UTILIZADOS E NÃO UTILIZADOS

Além da tabela de requisitos, é fundamental explicitar os conceitos de programaçãoorientada a objetos empregados, juntamente com suas justificativas, para assegurar maiorclareza na estruturação do sistema e no raciocínio projetual do jogo.

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

| **N.** | Conceitos | Uso | O quê / Onde & Justificativa em uma frase |
| --- | --- | --- | --- |
| **1** | **Elementares:** | | |
| 1.1  **&** | - Classes, objetos. &  - Atributos (privados), variáveis e constantes.  - Métodos (com e sem retorno). | Sim | - Todos .h e .cpp, como nas classes nos *namespaces* Gerenciadores, Entidades e Listas.  - Classes, 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 | - Todos .h e .cpp, como nas classes nos *namespaces* Gerenciadores, Entidades e Personagens.  - Arquivos .h e .cpp, como nas classes nos *namespaces* Gerenciadores.  - A constância pertinente evita mudanças equivocadas, construtores são mandatórios para inicializar atributos e destrutores pertinentes para finalizações como desalocações. |
| 1.3 | - Classe Principal. | Sim | - Precisamente nos arquivos .h e .cpp de gravity\_rooms.  - Uma classe Principal é mais ‘purista’ em termos de OO. |
| 1.4 | - Divisão em .h e .cpp. | Sim | - No desenvolvimento completo do jogo, tendo as classes seus .h e .cpp, com exceções devidas a relacionamentos ou aninhamento.  - Permite organizar as classes e afins que compõem o sistema. |
| **2** | **Relações de:** | | |
| 2.1 | - Associação direcional. &  - Associação bidirecional. | Sim | Todos .h e .cpp, como nas classes nos *namespaces* Background e Fase. Background precisa do ID da fase, enquanto a fase mantém uma lista de backgrounds para exibição.  Entre a classe Ente e o Gerenciador Gráfico. Ente usa o Gerenciador Gráfico para desenhar, mas o Gerenciador apenas recebe o sprite. |
| 2.2  **&** | - Agregação via associação.  - Agregação propriamente dita. | Sim | Precisamente no .h e .cpp em gravity\_rooms, no *namespace* Fases.  Os gerenciadores são instanciados enquanto o jogo existir.  A fase usa listas de entidades que podem existir sem ela, assim como os gerenciadores que ela utiliza, que existem independentemente da fase. |
| 2.3  **&** | - Herança elementar.  - Herança em vários níveis. | Sim | Arquivos .h e .cpp dos *namespaces* Menus e Botoes.  Menus precisam de sprites e Botoes precisam de funcionalidades específicas de botões e textos.  Em alguns dos .h e .cpp, como nas classes nos *namespaces* Personagens, Inimigos e Entidades.  Alguns personagens utilizam métodos e atributos semelhantes, criando uma cadeia entre Personagens e Entidades. |
| 2.4 | - Herança múltipla. | Sim | Precisamente no .h e .cpp de BotaoTexto.  A herança múltipla em BotaoTexto é usada para combinar as funcionalidades de Botao e Texto, permitindo que a classe herde atributos e métodos de ambas, criando botões com seleção e textos formatados nos menus. |
| **3** | **Ponteiros, generalizações e exceções** | | |
| 3.1 | - Operador *this* para fins de relacionamento bidirecional | Sim | Precisamente nos .h e .cpp, das classes Fase e Background.  Fase usa *this* como instância atual, permitindo que Background receba e obtenha o *ID* da fase desejada, através de sua própria referência, para carregar o *background* específico. |
| 3.2 | - Alocação de memória (*new* & *delete*). | Sim | Precisamente no .h em Lista.  Lista precisa reservar espaço dinâmico para as entidades e usar *delete* para apagá-las depois. |
| 3.3 | - Gabaritos/*Templates* criada/adaptados pelos autores para Listas. | Sim | Precisamente no .h em Lista e adaptado em Fase.  O *template* de elemento adapta-se ao tipo das entidades. |
| 3.4 | - Uso de Tratamento de Exceções (*try catch*). | Sim | Precisamente no .cpp de Projetil, Lista Entidades.  Elas permitem capturar e comunicar erros, garantindo que o programa continue funcionando. |
| **4** | **Sobrecarga de:** | | |
| 4.1 | - Construtoras e Métodos. | Sim | Usado nos .cpp e .h dos inimigos Clone, Androide e Ciborgue para permitir a criação de instâncias de inimigos com diferentes conjuntos de dados ou para realização de ações específicas. |
| 4.2 | - Operadores (2 tipos de operadores pelo menos). | Sim | Precisamente na Munição e Lista Entidades.  A primeira para decrementar a munição a cada tiro e a segunda, para acessar a posição específica na lista, principalmente aplicada nas atualizações e remoções de projéteis. |
| --- | **Persistência de Objetos (via arquivo de texto ou binário)** | | |
| 4.3 | - Persistência de Objetos. | Sim | Usado nos .cpp e .h das classes Save e Registry é justificada pela necessidade de manter o estado do jogo e as entidades entre execuções. |
| 4.4 | - Persistência de Relacionamento de Objetos. | Sim | Usado nos .cpp e .h das classes Save e Registry, justificado pela necessidade de manter o estado do jogo e as entidades entre execuções. |
| **5** | **Virtualidade:** | | |
| 5.1 | - Métodos Virtuais Usuais. | Sim | Usados nas classes dos *namespaces* Personagens e Obstaculos, essa técnica permite que diferentes tipos de inimigos e obstáculos definem comportamentos específicos. |
| 5.2 | - Polimorfismo. | Sim | Precisamente no .h e .cpp de Tripulante, Ciborgue, Clone, Androide e Projétil, todos derivando de Ente.  Isso facilita a manipulação e a extensão do sistema, permitindo que novos tipos de personagens sejam adicionados sem modificar o código existente. |
| 5.3 | - Métodos Virtuais Puros / Classes Abstratas. | Sim | Precisamente no .h e .cpp de Entidade, Ente e Fase, que são classes abstratas com métodos virtuais puros, garantindo que o comportamento necessário seja definido pelas classes derivadas |
| 5.4 | - Coesão/Desacoplamento efetiva e intensa com o apoio de padrões de projeto (mais de 5 padrões). | Não | Requisito não cumprido. |
| **6** | **Organizadores e Estáticos** | | |
| 6.1 | - Espaço de Nomes (*Namespace*) criada pelos autores. | Sim | Precisamente nos arquivos .h de Menu e Botao, a fim de organizar o código em grupos lógicos e evitar conflitos de nome. |
| 6.2 | - Classes aninhadas (*Nested*) criada pelos autores. | Sim | Precisamente no .h de Tripulante, com a classe Munição, permitindo encapsular comportamentos específicos e criar uma estrutura hierárquica, como a munição específica daquele jogador. |
| 6.3 | - Atributos estáticos e métodos estáticos. | Sim | Usado no Gerenciador Gráfico, Registry, a fim de serem compartilhados por todas as instâncias de uma classe (e evitar assim, com o Singleton, ter uma janela duplicada) . |
| 6.4 | - Uso extensivo de constante (*const*) parâmetro, retorno, método... | Sim | Todos .h e .cpp, como nas classes no *namespace* Gerenciadores, garantindo que valores não sejam modificados, assim como os atributos da classe. |
| **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 | - Precisamente no .h do Gerenciador Gráfico, a fim de armazenar o nome da janela.  - Precisamente no .h do Menu.h , a fim de armazenar o conteúdo a ser mostrado no menu e capturar o nome completo digitado pelo usuário. |
| 7.2 | - Pilha, Fila, Bifila, Fila de Prioridade, Conjunto, Multi-Conjunto, Mapa **OU** Multi-Mapa. | Sim | Precisamente na classe Fase, para gerenciar a criação e o armazenamento de inimigos e obstáculos em diferentes níveis de dificuldade, permitindo a inserção e remoção de elementos de forma ordenada e controlada (de 3 a 7). |
| --- | **Programação concorrente** | | |
| 7.3 | *- Threads* (Linhas de Execução) no âmbito da Orientação a Objetos, utilizando Posix, C-Run-Time **OU** Win32API ou afins*.* | Não | Requisito não cumprido. |
| 7.4 | *- Threads* (Linhas de Execução) no âmbito da Orientação a Objetoscom uso de Mutex, Semáforos, **OU** Troca de mensagens. | Sim | Precisamente na classe Gerenciador\_Thread, utilizada no Gerenciador\_Físico a fim de dividir o processo de soma de Riemann da operação restante na fórmula da velocidade final, travando enquanto não for terminada a soma para evitar valores incorretos. |
| **8** | **Biblioteca Gráfica / Visual** | | |
| 8.1  **&** | - Funcionalidades Elementares.  - Funcionalidades Avançadas como: tratamento de colisões, duplo *buffer* ou outros. | Sim | Precisamente na classe Gerenciador Gráfico, Ente e Gravity Rooms.  Ente lida com Textura e Sprite, enquanto o Gerenciador Gráfico desenha os *sprites* na tela. Em união com Gravity Rooms e Gerenciador Gráfico, primeiro a tela é limpa, depois desenhada (*back buffer*) e, por fim, exibida (*front buffer*) para evitar *flickering*. |
| 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). | Não | Requisito não cumprido. |
| --- | **Interdisciplinaridades via utilização de Conceitos de Matemática Contínua e/ou Física.** | | |
| 8.3 | - Ensino Médio Efetivamente. | Sim | Usado na classe Personagem para calcular o deslocamento utilizando o conceito de física como MRUV (Movimento Retilíneo Uniformemente Variado). |
| 8.4 | - Ensino Superior Efetivamente*.* | Sim | No Gerenciador Físico, usa a soma de Riemann para obter o dano aproximado, aplicando conceitos como trabalho de força variável e variação de energia cinética. |
| **9** | **Engenharia de Software** | | |
| 9.1 | - Compreensão, melhoria e rastreabilidade de cumprimento de requisitos. | Sim | Acrescentado ao projeto Menus, Botões, Gerenciadores específicos e o modelo de relatório preenchido utilizando *Google Docs* por ambos, permitindo monitorar o progresso e adaptar o desenvolvimento conforme mudanças nas necessidades. |
| 9.2 | - Diagrama de Classes em *UML*. | Sim | Manipulado pelo programa *StarUML* 6.3.0, com o diagrama no repositório do *GitHub* ou *Google Drive*. |
| 9.3 | - Uso efetivo e intensivo de padrões de projeto *GOF*, *i.e.*, mais de 5 padrões. | Parcial | Requisito não cumprido.  Singleton foi aplicado ao Gerenciador Gráfico, garantindo uma única instância global para evitar duplicação de recursos e conflitos.  O padrão Iterator foi utilizado nas classes do namespace Menus, facilitando a navegação entre botões e permitindo uma seleção cíclica eficiente.  Já o Factory Method foi empregado em Save e Registry, centralizando a criação de instâncias de Ente com base em dados JSON.  Por fim, o Template Method foi implementado em Lista.h, definindo a estrutura de iteração e permitindo personalização do comportamento durante a execução. |
| 9.4 | - Testes à luz da Tabela de Requisitos e do Diagrama de Classes. | Sim | Verificado e aprimorado a cada *backup* do código, disponível no *Google Drive*. |
| **10** | **Execução de Projeto** | | |
| 10.1  **&** | - Controle de versão de modelos e códigos automatizado (via github).  - Uso de alguma forma de cópia de segurança (*i.e.*, *backup*). | *Sim* | Utilizado *Git* e [*GitHub*](http://github), com divisões de branches, remote e ssh, e divisão de projetos com participantes, usufruindo de merge. [*Google Drive*](https://drive.google.com/drive/folders/1LNBPyoVW8X_zRPqrhXA7f47HVpSJQFhn?usp=drive_link) utilizado para o salvamento dos *backups*. |
| 10.2 | - Reuniões com o professor para acompanhamento do andamento do projeto.  **[ITEM OBRIGATÓRIO PARA A ENTREGA DO TRABALHO]** | *Sim* | Primeira reunião dia 10/12/24 às 15h05. Discutidas táticas para abordar o jogo, como desenvolvimento pelas classes folhas.  Segunda reunião dia 17/12/24 às 15h11. Discutidos métodos de resolução de problemas. |
| 10.3 | - Reuniões com monitor da disciplina para acompanhamento do andamento do projeto.  **[ITEM OBRIGATÓRIO PARA A ENTREGA DO TRABALHO]** | Sim | 05/12/24 às 15:50 - CB-002 - Curso Peteco (*UML*)  12/12/24 às 17h00 - CB-002 - Curso Peteco (Física e Matemática)  12/12/2024 (20h21 - 21h44) - Monitoria (Correção de problemas)  03/02/2025 (20h39 - 21h12) - Monitoria (Relatório) |
| 10.4  **&** | - Escrita do trabalho e feitura da apresentação  - Revisão do trabalho escrito de outra equipe e vice-versa. | *Sim* | Escrito e revisado pelos autores Nícolas e Isabela.  Revisado pelo grupo de Felipe Mossato e Andre Castilhano. |
| **Total de conceitos apropriadamente utilizados.**  *(Cada grande tópico vale 10% do total de conceitos. Assim, caso se tenha feito metade de um tópico, então ele valeria 5%.)* | | | **36/40 (90%) Concluído** |

# DISCUSSÃO E CONCLUSÕES

Dessa forma, foi necessária a adaptação do modelo *UML* proposto para viabilizar o funcionamento adequado do projeto. Além da modelagem *UML*, destaca-se a relevância das técnicas discutidas nas reuniões com o professor, as quais foram fundamentais para garantir um desenvolvimento fluido e eficiente do jogo. Entre essas estratégias, ressaltam-se: a criação preliminar de classes vazias para estruturar os relacionamentos antes de sua implementação; a documentação extensiva do código, permitindo a identificação e resolução precisa de erros; e a formulação de relações temporárias entre classes, possibilitando uma abordagem incremental, do menos complexo para o mais complexo, seguindo um modelo semelhante ao "Espiral".

No que tange aos cursos do PETECO, eles proporcionaram uma visão antecipada dos desafios a serem enfrentados, bem como de suas possíveis soluções e alternativas. Além disso, esses cursos permitiram a aquisição de novas técnicas, o aprimoramento de habilidades e a consolidação de práticas de organização em grupo. Paralelamente, às reuniões com os monitores revelaram-se uma via essencial para a resolução de erros, além de servirem como fonte de sugestões e revisões do código.

No contexto do trabalho em equipe e da construção colaborativa, o versionamento de código e os *backups* mostraram-se ferramentas indispensáveis. Esses mecanismos possibilitaram a interação direta no código, com a devida inclusão de comentários e explicações, a recuperação de versões anteriores e o desenvolvimento simultâneo por diferentes integrantes da equipe. No entanto, tornou-se necessário um estudo preliminar sobre compiladores para compreender determinados erros de código aparentemente insolúveis. Esse estudo revelou que certos problemas decorrem da forma como o compilador interpreta classes e relacionamentos, podendo gerar erros que extrapolam o escopo da disciplina de Orientação a Objetos.

Além do *GitHub*, diversas outras ferramentas foram introduzidas e aprimoradas para o desenvolvimento eficiente do jogo, dentre as quais se destacam: *Neovim*, *CMake*, *gdb*, *ChatGPT*, *Dream Lab* (*Canva*), *Photoshop*, *VSCode*, *LiveShare* (*VSCode*), *clang-format*, *scripts* (.sh) e *JSON*.

A atualização contínua do modelo *UML* e o preenchimento simultâneo dos requisitos durante o desenvolvimento do projeto mostraram-se estratégias fundamentais para a organização e otimização do tempo. Esse método não apenas facilitou a adaptação ao progresso do projeto, mas também proporcionou uma visão abrangente do estado atual do desenvolvimento, permitindo a identificação de melhorias e a compreensão de métodos necessários.

A interpretação do modelo e dos requisitos revelou-se um ponto sensível, exigindo uma análise prévia e um planejamento estratégico não apenas da classe em desenvolvimento, mas também de seus relacionamentos. Além disso, a permissão para modificações em “classes folhas” desde as etapas iniciais do projeto possibilitou o aprimoramento de métodos em classes internas e bases, além da realização de testes temporários.

Conforme discutido em reuniões com o professor, a validação de certas classes ou funções seguiu a abordagem de testá-las em estruturas sem relacionamentos diretos ou com adaptações mínimas. Essa estratégia permitiu a experimentação de operações e metodologias antes de sua aplicação definitiva nas classes designadas, promovendo um desenvolvimento mais modular e organizado.

Os conceitos de coesão e desacoplamento estruturaram uma nova perspectiva no desenvolvimento do projeto, permitindo a aplicação de boas práticas de *design*. No entanto, a implementação integral desses princípios foi limitada por restrições da biblioteca utilizada e pela necessidade de modificações abrangentes em namespaces completos. Essa limitação evidenciou a importância de planejar a coesão e o desacoplamento já na modelagem *UML*, visto que dificuldades de adaptação nesse estágio podem impactar significativamente o tempo de desenvolvimento.

Seguir tabela de requisitos mostrou-se uma forma eficaz de dimensionar o resultado no final, pois ao contabilizar a quantidade de requisitos é possível estipular a qualidade final do projeto.

# DIVISÃO DO TRABALHO

Com base no diagrama e nos conceitos apresentados, o trabalho no jogo foi conduzido de maneira harmônica e síncrona, garantindo uma colaboração contínua. Seguindo linhas de raciocínio estruturadas, cada objetivo foi abordado de forma sequencial e focada, como segue a respectiva distribuição:

Tabela 4. Lista de Atividades e Responsáveis.

| Atividades | Responsáveis |
| --- | --- |
| **Elementares: AMBOS** | |
| - Classes, objetos. & - Atributos (privados), variáveis e constantes. &  - Métodos (com e sem retorno). | Nícolas e Isabela |
| - Métodos (com retorno *const* e parâmetro *const*). &  - Construtores (sem/com parâmetros) e destrutores | Nícolas e Isabela |
| - Classe Principal. | Nícolas e Isabela |
| - Divisão em .h e .cpp. | Nícolas e Isabela |
| **Relações de: AMBOS** | |
| - Associação direcional. & - Associação bidirecional. | Nícolas e Isabela |
| - Agregação via associação. & - Agregação propriamente dita. | Nícolas e Isabela |
| - Herança elementar. & - Herança em vários níveis. | Nícolas e Isabela |
| - Herança múltipla. | Nícolas e Isabela |
| **Ponteiros, generalizações e exceções: AMBOS** | |
| - Operador *this* para fins de relacionamento bidirecional. | Nícolas e Isabela |
| - Alocação de memória (*new* & *delete*). | Nícolas e Isabela |
| - Gabaritos/*Templates* criada/adaptados pelos autores para Listas. | Nícolas e Isabela |
| - Uso de Tratamento de Exceções (*try catch*). | Nícolas e Isabela |
| **Sobrecarga de: AMBOS** | |
| - Construtoras e Métodos. | Nícolas e Isabela |
| - Operadores (2 tipos de operadores pelo menos) | Nícolas e Isabela |
| **Persistência de Objetos (via arquivo de texto ou binário): AMBOS** | |
| - Persistência de Objetos. | Nícolas e Isabela |
| - Persistência de Relacionamento de Objetos. | Nícolas e Isabela |
| **Virtualidade: AMBOS** | |
| - Métodos Virtuais Usuais. | Nícolas e Isabela |
| - Polimorfismo. | Nícolas e Isabela |
| - Métodos Virtuais Puros / Classes Abstratas. | Nícolas e Isabela |
| - Coesão/Desacoplamento efetiva e intensa com o apoio de padrões de projeto (mais de 5 padrões). |  |
| **Organizadores e Estáticos: AMBOS** | |
| - Espaço de Nomes (*Namespace*) criada pelos autores. | Nícolas e Isabela |
| - Classes aninhadas (*Nested*) criada pelos autores. | Nícolas e Isabela |
| - Atributos estáticos e métodos estáticos. | Nícolas e Isabela |
| - Uso extensivo de constante (*const*) parâmetro, retorno, método... | Nícolas e Isabela |
| **Standard Template Library *(STL)* e String OO:**  **AMBOS** | |
| - A classe Pré-definida *String* ou equivalente. **&** *- Vector* e/ou *List* da *STL* (p/ objetos ou ponteiros de objetos de classes definidos pelos autores) | Nícolas e Isabela |
| - Pilha, Fila, Bifila, Fila de Prioridade, Conjunto, Multi-Conjunto, Mapa **OU** Multi-Mapa. | Nícolas e Isabela |
| **Programação concorrente: AMBOS** | |
| *- Threads* (Linhas de Execução) no âmbito da Orientação a Objetos, utilizando Posix, C-Run-Time **OU** Win32API ou afins*.* | Nícolas e Isabela |
| *- Threads* (Linhas de Execução) no âmbito da Orientação a Objetoscom uso de Mutex, Semáforos, **OU** Troca de mensagens. |  |
| **Biblioteca Gráfica / Visual:** | |
| - Funcionalidades Elementares. **&**  - Funcionalidades Avançadas como: tratamento de colisões e duplo *buffer* | Nícolas e Isabela |
| - 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). |  |
| **Interdisciplinaridades via uso de Conceitos de Matemática Contínua e/ou Física: AMBOS** | |
| - Ensino Médio Efetivamente. | Nícolas e Isabela |
| - Ensino Superior Efetivamente*.* | Nícolas e Isabela |
| **Engenharia de Software:AMBOS** | |
| - Compreensão, melhoria e rastreabilidade de cumprimento de requisitos. | Nícolas e Isabela |
| - Diagrama de Classes em *UML*. | Nícolas e Isabela |
| - Uso efetivo e intensivo de padrões de projeto *GOF*, *i.e.*, + de 5 padrões. |  |
| - Testes à luz da Tabela de Requisitos e do Diagrama de Classes. | Nícolas e Isabela |
| **Execução de Projeto: AMBOS** | |
| - Controle de versão de modelos e códigos automatizado (via github). &  *- Uso de alguma forma de cópia de segurança (i.e., backup).* | Nícolas e Isabela |
| - Reuniões com o professor para acompanhamento do andamento do projeto. **[ITEM OBRIGATÓRIO A ENTREGA DO TRABALHO]** | Nícolas e Isabela |
| - Reuniões com monitor da disciplina para acompanhamento do andamento do projeto. ***[ITEM OBRIGATÓRIO PARA A ENTREGA]*** | Nícolas e Isabela |
| - Escrita do trabalho e feitura da apresentação **&**  - Revisão do trabalho escrito de outra equipe e vice-versa. | Nícolas e Isabela |

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

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

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

# AGRADECIMENTOS PROFISSIONAIS

Gostaríamos de expressar gratidão a todos que contribuíram para a realização deste trabalho. Em especial, à equipe de monitoria do curso de Técnicas de Programação do segundo semestre de 2024 pelo apoio e orientação fundamentais durante o desenvolvimento deste jogo. Também ao suporte proporcionado pelo PETECO (Programa de Educação Tutorial em Engenharia de Computação), cujo manual e apresentações foram indispensáveis para a execução deste projeto.

Ainda à equipe do Andre Costa Castilhano e Felipe da Silva Mossato que revisaram este trabalho, oferecendo valiosas sugestões, sejam elas teóricas e filosóficas, revisões documentais e contribuições que enriqueceram o resultado final.

# REFERÊNCIAS CITADAS NO TEXTO

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

[2] STADZISZ, P. C. Projeto de Software usando UML. Apostila CEFET-PR 2002.

<http://www.dainf.ct.utfpr.edu.br/~jeansimao/Fundamentos2/EngSoftware/Apostila%20UML%20-%20Stadzisz%202002.pdf>

[3] SIMÃO, J. M. Site das Disciplina de Fundamentos de Programação 2, Curitiba – PR, Brasil, Acessado em 06/12/2024, às 12:33 -

[https://www.dainf.ct.utfpr.edu.br/~jeansimao/Fundamentos2/Fundamentos2.htm](http://www.dainf.ct.utfpr.edu.br/~jeansimao/Fundamentos2/Fundamentos2.htm).

# REFERÊNCIAS UTILIZADAS NO DESENVOLVIMENTO

**[A]** BEZERRA, E. *Princípios de Análise e Projeto de Sistemas com UML.* 1ª ed. Rio de Janeiro: Editora Campus, 2003. ISBN 85-352-1032-6.

**[B]** HORSTMANN, C. *Conceitos de Computação com o Essencial de C++.* 3ª ed. Porto Alegre: Bookman, 2003. ISBN 0-471-16437-2.

**[C]** HALLIDAY, D.; RESNICK, R. *Fundamentos da Física: Gravitação, Ondas e Termodinâmica.* 10ª ed. Rio de Janeiro: LTC, 2016. ISBN 978-8521637339.

**[D]** GitHub. *GerenciadorGrafico.hpp.* Disponível em: [Magic-Worlds - Gerenciador Gráfico](https://github.com/jaoRamalho/Magic-Worlds/blob/main/Game/include/Gerenciadores/GerenciadorGrafico.hpp#L24). Acesso em: 17 dez. 2024, 15h08.

**[E]** SFML. *Tutoriais.* Disponível em: [www.sfml-dev.org/tutorials](https://www.sfml-dev.org/tutorials). Acesso em: 17 dez. 2024, 15h08.

**[F]** W3Schools. *Git.* Disponível em: [W3Schools - Git](https://www.w3schools.com/git/). Acesso em: 4 fev. 2025.

**[G]** W3Schools. *C++ - OOP.* Disponível em: [W3Schools - C++ OOP](https://www.w3schools.com/cpp/cpp_oop.asp). Acesso em: 4 fev. 2025.

**[H]** GeeksforGeeks. *Object-Oriented Programming in C++.* Disponível em: [GeeksforGeeks - OOP in C++](https://www.geeksforgeeks.org/object-oriented-programming-in-cpp/). Acesso em: 4 fev. 2025.

**[I]** Javatpoint. *C++ OOPS Concepts.* Disponível em: [Javatpoint - C++ OOPS Concepts](https://www.javatpoint.com/cpp-oops-concepts). Acesso em: 4 fev. 2025.

**[J]** Cplusplus. *C++ Documentation.* Disponível em: [Cplusplus - C++ Documentation](https://cplusplus.com/doc/). Acesso em: 4 fev. 2025.

**[K]** *Manual Project Simas.* Disponível em: [Google Docs - Manual Project Simas](https://docs.google.com/document/d/1zZhXefhxtHiROkwvyBsDSwLcbZ6xUYBJE1wMBnSyOuE/edit?tab=t.0). Acesso em: 4 fev. 2025.

**[L]** GitHub. *JogoPlataforma2D-Jungle.* Disponível em: [GitHub - JogoPlataforma2D-Jungle](https://github.com/Giovanenero/JogoPlataforma2D-Jungle). Acesso em: 4 fev. 2025.

**[M]** YouTube. *Criando um Jogo em C++ do ZERO.* Disponível em: [Playlist - Criando um Jogo em C++ do ZERO](https://www.youtube.com/watch?v=gfGE5KY1OQU&list=PLR17O9xbTbIBBoL3lli44N8LdZVvg-_uZ&ab_channel=Gege%2B%2B). Acesso em: 4 fev. 2025.

# Apêndice A

A fórmula do Movimento Retilíneo Uniforme para a Função Horária da Posição segue:

Onde:

S = posição final do personagem (m);

= posição inicial do personagem (m);

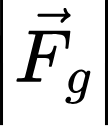
= velocidade do personagem (m/s);

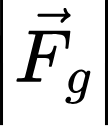
= intervalo de tempo medido (s);

Considerando variações em função do tempo, é possível desconsiderar a posição inicial:

# Apêndice B

Para calcular o dano recebido em uma colisão, considerando a mecânica clássica (desconsiderando efeitos quânticos ou relativísticos, como a radiação de Hawking, que permite a perda de massa dos buracos negros), utilizamos valores pré-definidos e pontos específicos no horizonte de eventos dentro da caixa de colisão do obstáculo. O tratamento ocorre externamente à região gravitacional variável, evitando a necessidade de um modelo interno mais complexo. Assim, segue-se o seguinte pela Lei da Gravitação Universal:



Onde:

= módulo da força gravitacional (N);

G = constante gravitacional (6.674×10⁻¹¹ m³ kg⁻¹ s⁻²);

= massa do tripulante (70kg);

= massa do buraco negro (1×10¹² kg);

= distância entre o tripulante e o centro gravitacional.

Porém, , fazendo cair em um poço gravitacional infinito. Portanto, será considerado = 0.1m e . Para isolar a velocidade final (em módulo):

Onde e são as energias cinética final e inicial do corpo. Para este desenvolvimento, segue-se que .

Dado que:

Portanto,

Se assumirmos que o corpo parte do repouso (=0), então:

(1)

Pela lógica, se o valor da velocidade for muito alto, o tripulante morrerá. (Supondo que uma pessoa pode suportar 5g (Força g) por 10 segundos, resultará em 1.764 km/h como velocidade máxima suportada).

Entretanto, para evitar o processamento por integrais, será utilizada uma aproximação por Soma de Riemann:

(2)

Onde:

posição da partícula no subintervalo i;

Δr = largura dos subintervalos;

n = número de divisões do intervalo [.

Substituindo, portanto, (2) em (1), obtém-se:

Se aumentar o valor de n (mais divisões), a aproximação fica mais precisa e tende ao valor da integral. Porém, por questões de otimização e limitações de variáveis, serão reservadas 3 subdivisões, e como retorno um valor possível de se armazenar em um inteiro, fazendo uma divisão por .