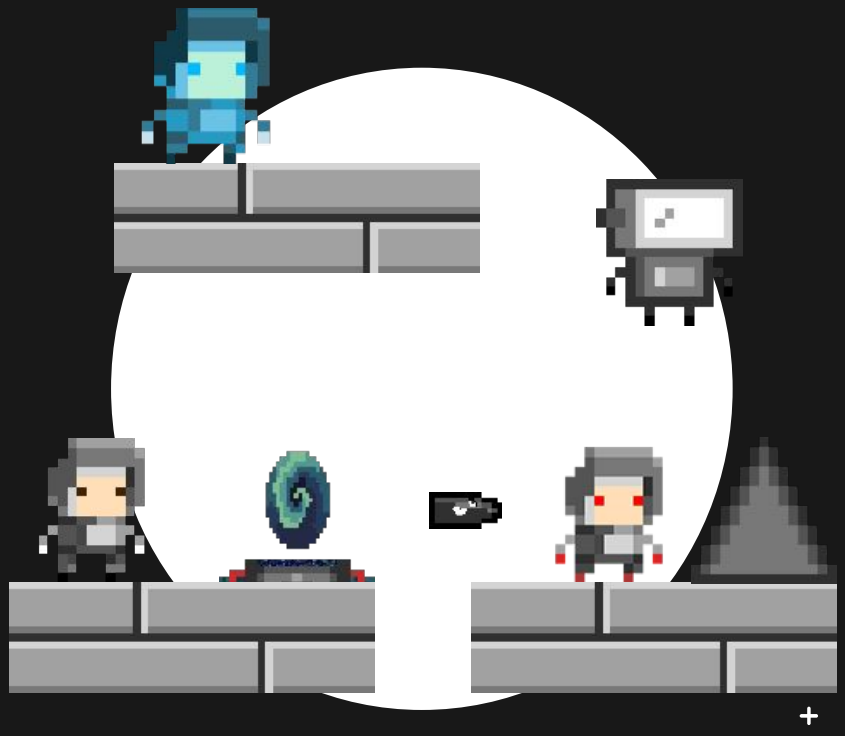
GRAVITY ROOMS

Jogo para Técnicas de Programação (S73)
Engenharia de Computação

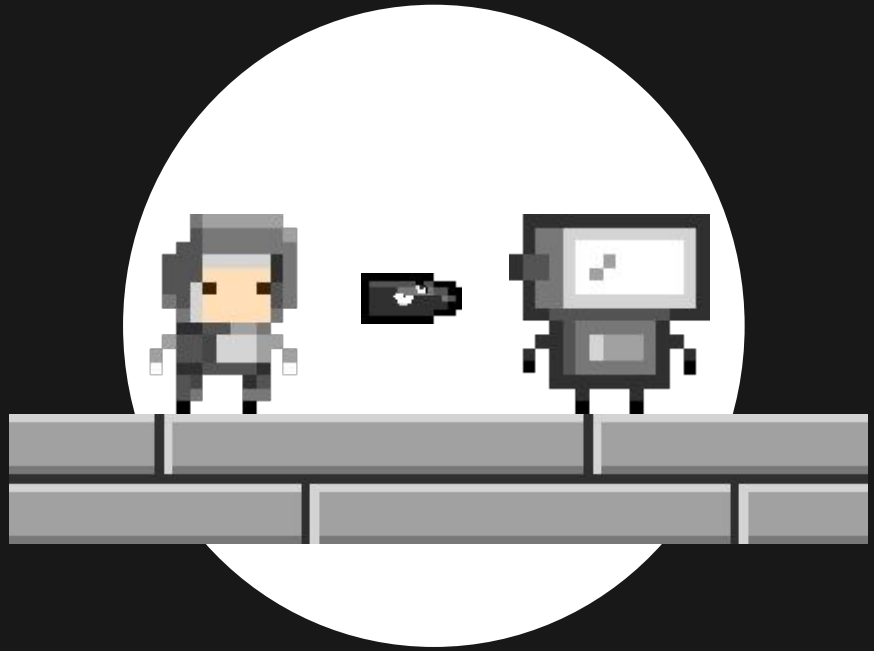
Nícolas Auersvalt Marques
Isabela Bella Bortoleto

Universidade Tecnológica Federal do Paraná



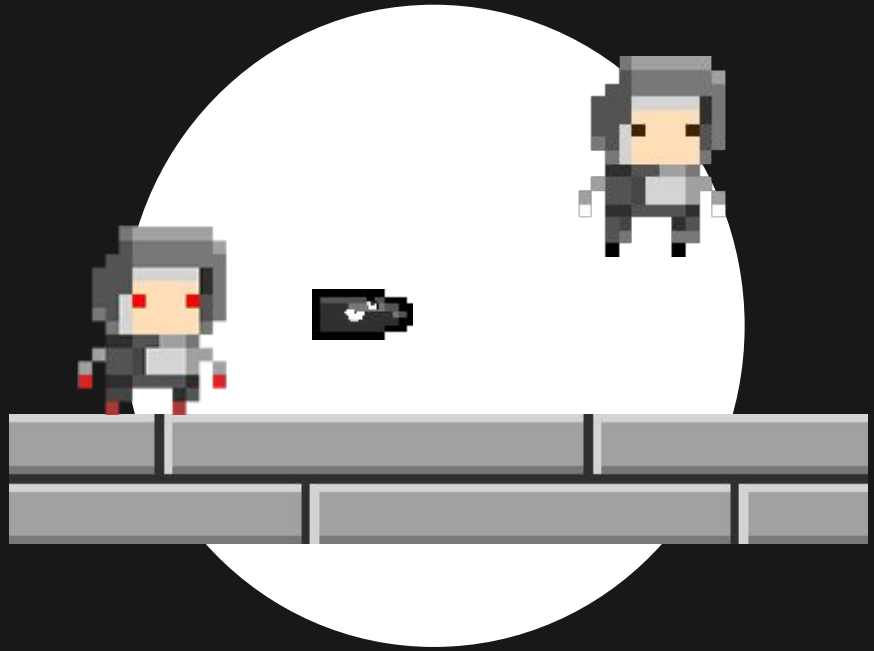
1. REQUISITOS DO JOGO

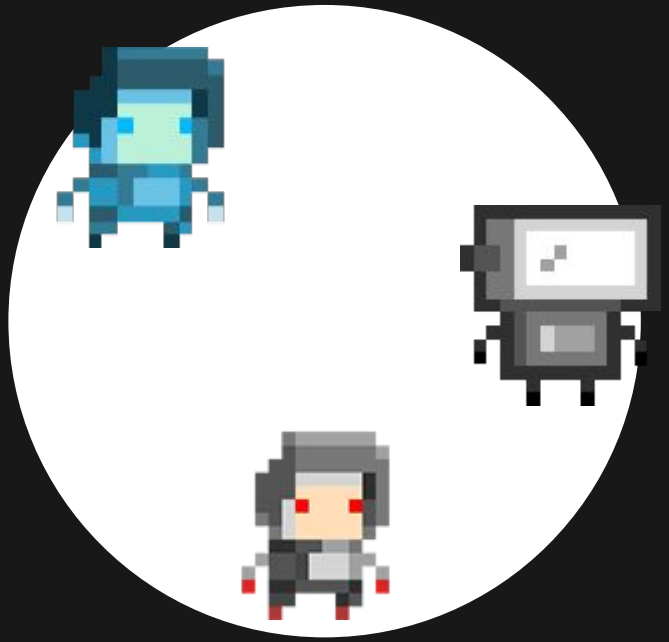
- CMake: (3.22.1).
- Make: (GNU Make 4.3).
- Compilador C++: g++ ou clang++ (g++ (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0).
- Debian 12
- SFML 2.5.1



2. INTRODUÇÃO

- Linguagem C++
- Biblioteca SFML
- Orientação a Objetos
- Diagrama de Classes
- Engenharia de Software
- Padrões de Projeto
- Jogo de Plataforma





3. SOBRE O JOGO

Inspiração em jogos filosóficos e na própria Metafísica.
Inimigos refletem a dificuldade do pensamento:

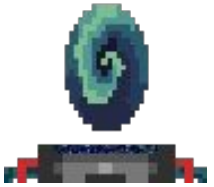
- Ciborgue - humano com órgãos robóticos (fácil);
- Androide - robô que simula um humano (médio);
- Clone - cópia genética exata do tripulante (chefe).

Os inimigos são humanos? O que definiria o ser? Ou o que define ser humano?

3. SOBRE O JOGO

- Jogo de Plataforma
- Menus opções:
  - Ranking
  - Salvamento
  - Pausamento
- Carregamento

- Três inimigos:
  - Ciborgue
  - Androide
  - Clone
- Projétil
- Um ou dois Jogadores



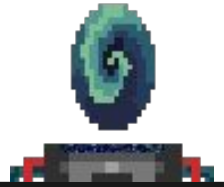
- Obstáculos:
  - Plataforma
  - Espinho
  - Centro Gravitacional
- Fases:
  - Laboratório
  - Nave

3. SOBRE O JOGO

Fórmulas utilizadas:

$$S = v \cdot \Delta t$$

$$|v_f| = \sqrt{\frac{2}{m}\int_{r_{inicial}}^{r_{final}} \frac{G \cdot m_{tripulante} \cdot m_{gravitacional}}{(r_{tripulante-gravitacional})^2} dr} \Rightarrow |v_f| \simeq \sqrt{\frac{2}{m}\sum_{i=1}^{n} \left(\frac{G \cdot m_{tripulante} \cdot m_{gravitacional}}{(r_i)^2}\right) \Delta r}$$



4. TABELA DE REQUISITOS

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, sendo que na hierarquia de personagens há apenas dois tipos de inimigos. Clone 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) e Centro_Gravidade (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.			

4. TABELA DE REQUISITOS

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* Gerenciadore Entidades e Listas. - Classes, Objetos, Atributos e Métodos foram utilizados porque sã 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* Gerenciadore Entidades e Personagens. - Arquivos .h e .cpp, como nas classes nos *namespac* Gerenciadores. - A constância pertinente evita mudanças equivocadas, construtor são mandatórios para inicializar atributos e destrutores pertinent 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 .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.

4. TABELA DE REQUISITOS

N.	Conceitos	Uso	O quê / Onde & Justificativa em uma frase
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.

4. TABELA DE REQUISITOS

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.

4. TABELA DE REQUISITOS

N.	Conceitos	Uso	O quê / Onde & Justificativa em uma frase
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 Objetos com 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.

4. TABELA DE REQUISITOS

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 <u>Matemática Contínua e/ou Física</u>.**		
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*.

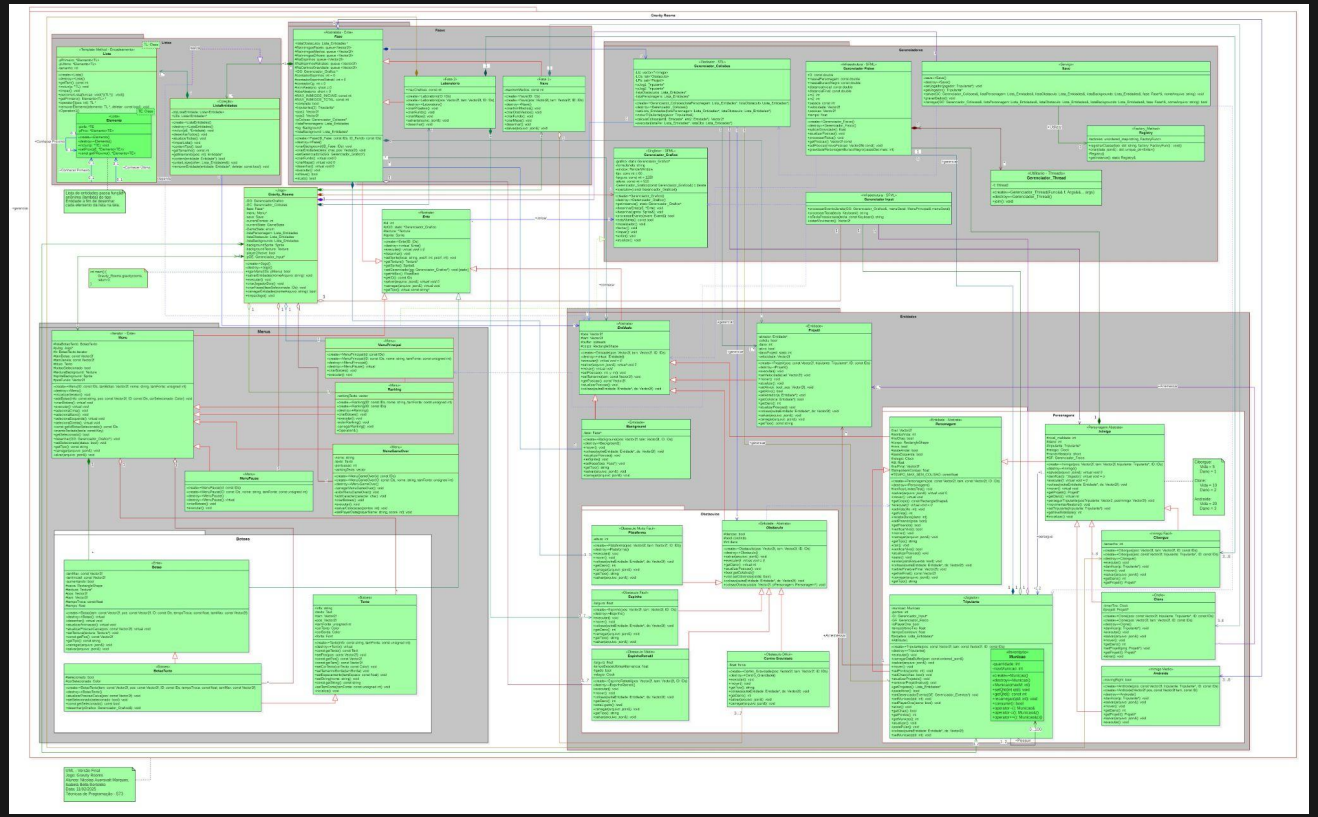
4. TABELA DE REQUISITOS

N.	Conceitos	Uso	O quê / Onde & Justificativa em uma frase
9.3	- Uso efetivo e intensivo de padrões de projeto *GOF*, *i.e.*, mais de 5 padrões.	Parcial	Requisito não cumprido. Singleton foi utilizado no Gerenciador Gráfico. Além disso, *Iterator* foi utilizado nas classes do *namespace* Menus. Já o Factory method foi usado salvamento e o Template method na Lista. Singleton: Garante uma única instância global de Gerenciador_Grafico, evitando duplicação de recursos e conflitos. Iterator: Facilita a navegação entre botões no Menu, permitindo seleção cíclica e controle eficiente dos elementos. Factory Method: Simplifica a criação de instâncias de Ente via Registry, centralizando a lógica de construção com base em dados JSON. Template Method: Define a estrutura de iteração em Lista.h, 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*.

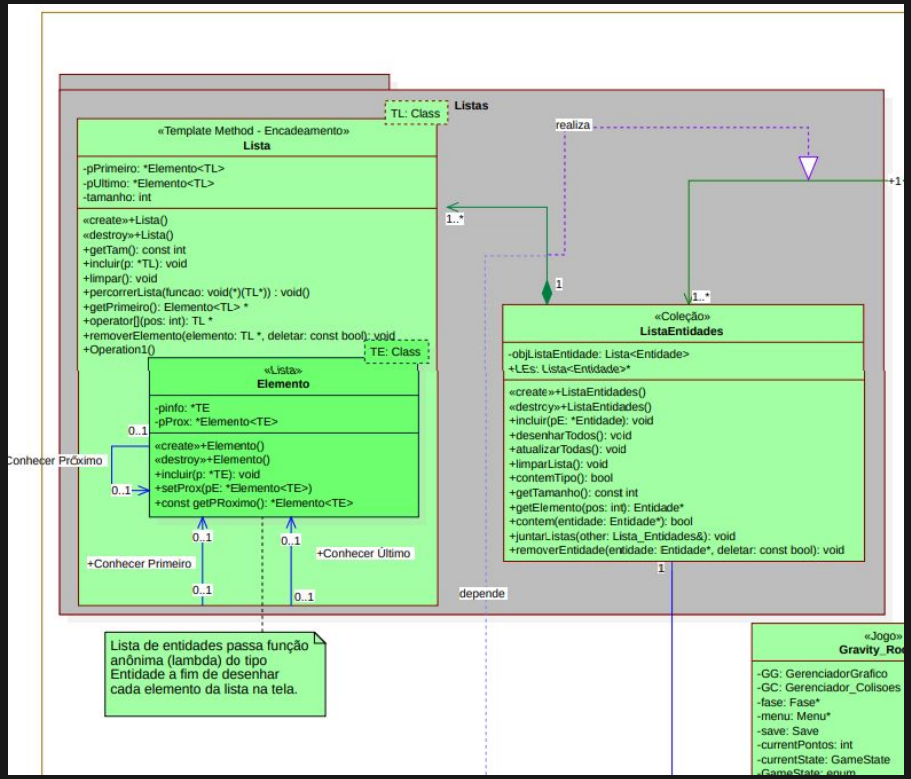
4. TABELA DE REQUISITOS

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*, com divisões de branches, remote e ssh, e divisão de projetos com participantes, usufruindo de merge. *Google Drive* 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**

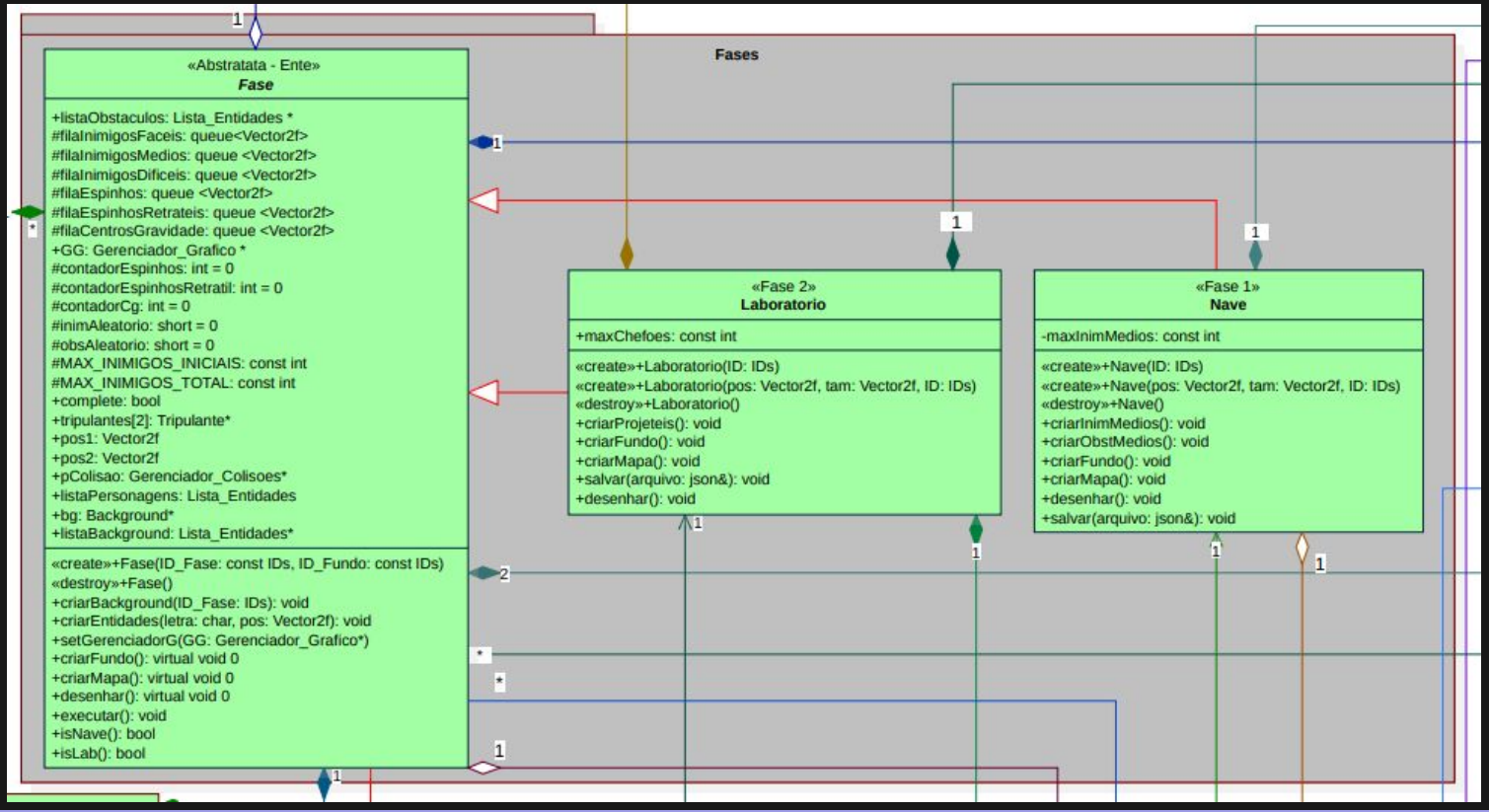
5. DIAGRAMA DE CLASSES



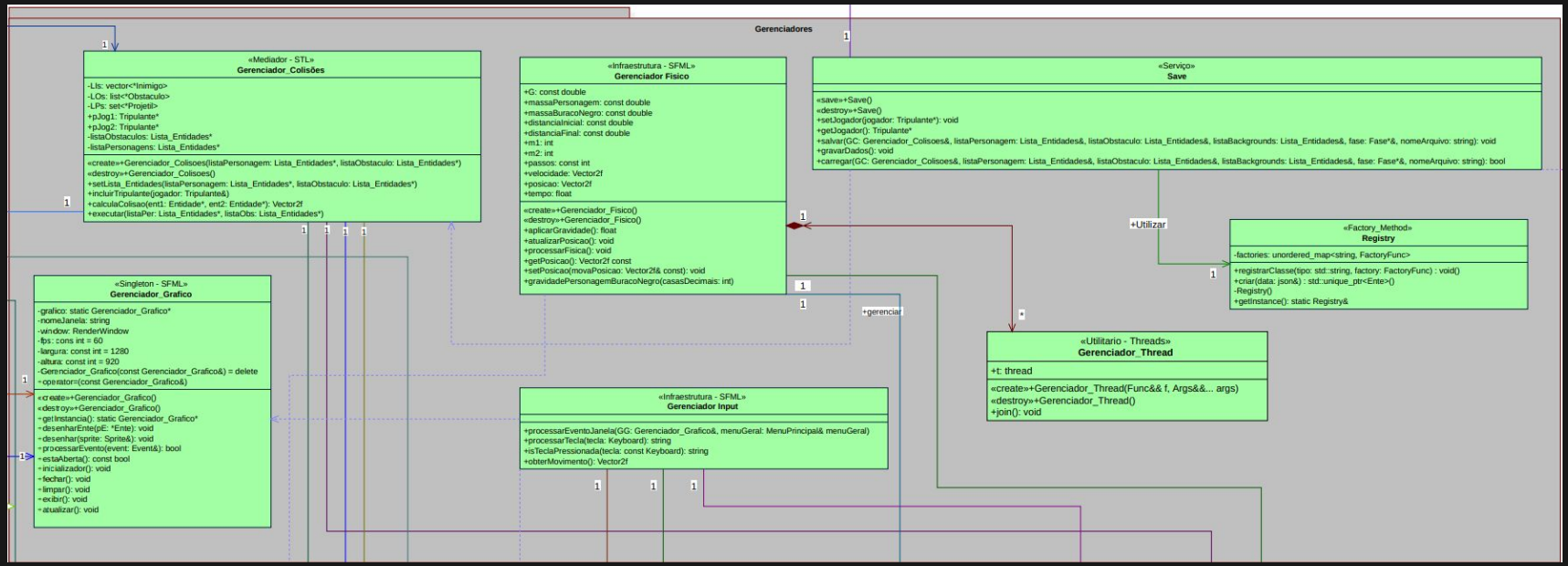
5. DIAGRAMA DE CLASSES



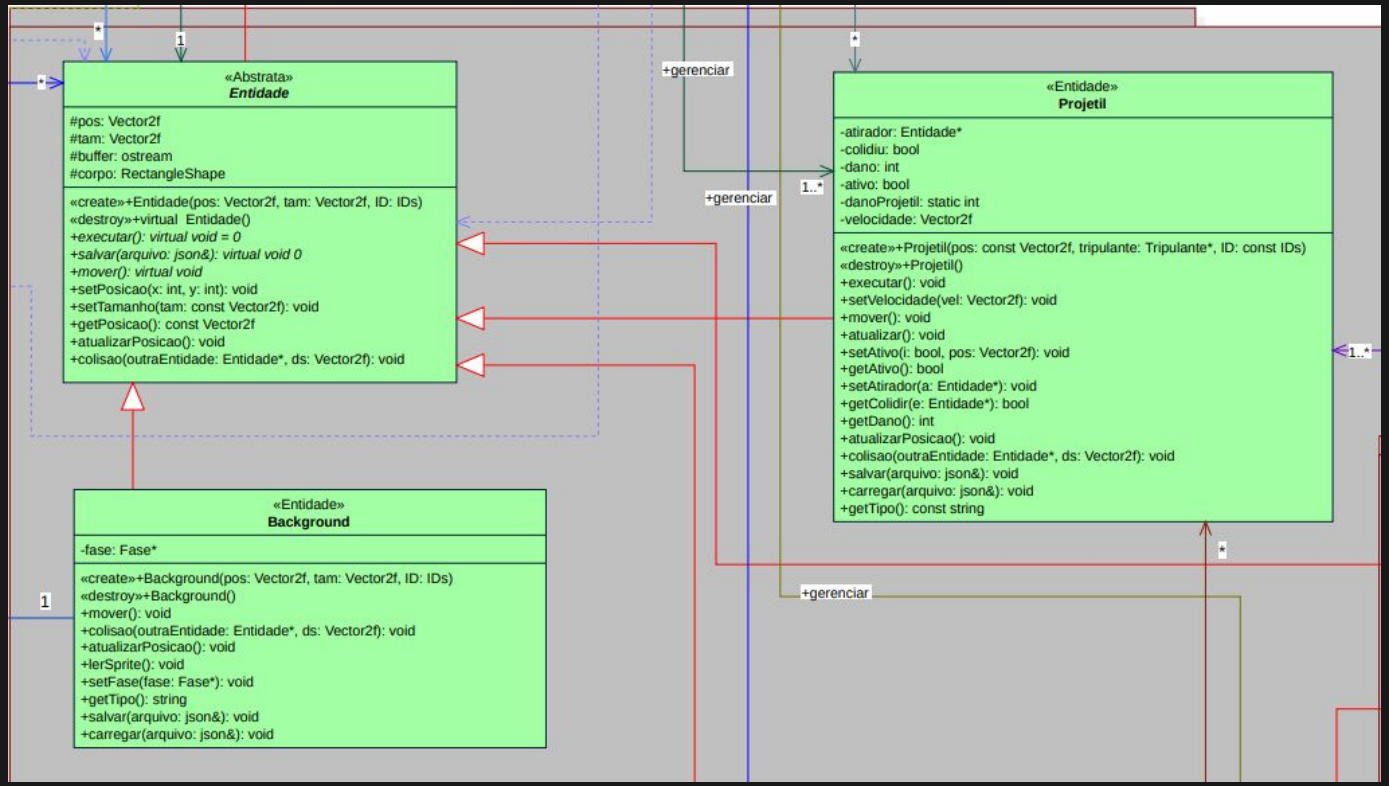
5. DIAGRAMA DE CLASSES



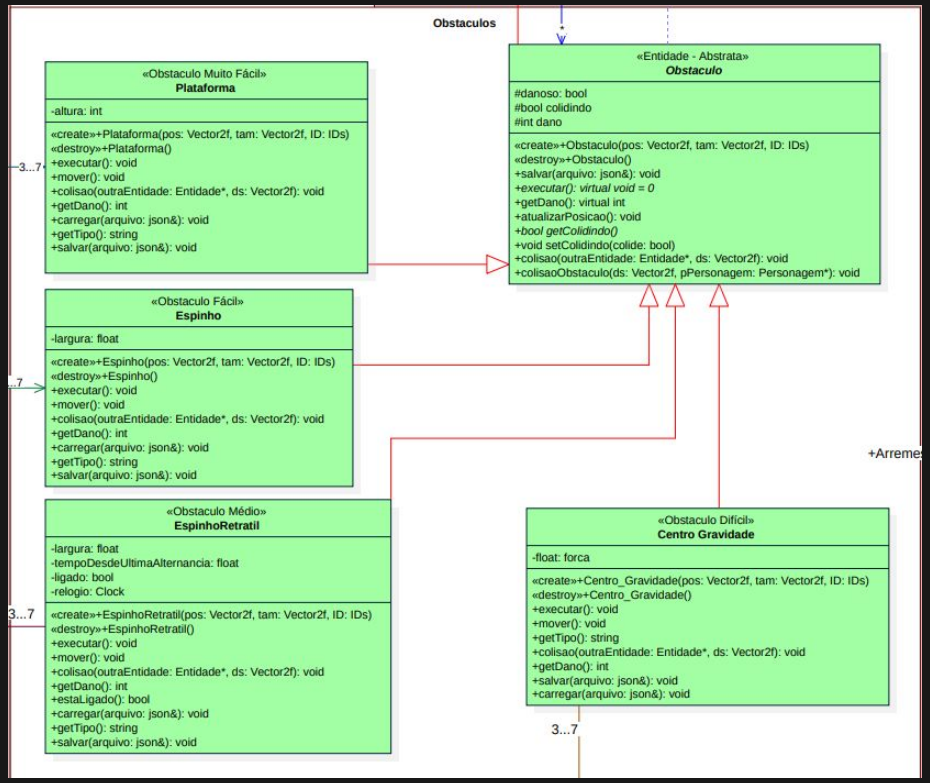
5. DIAGRAMA DE CLASSES



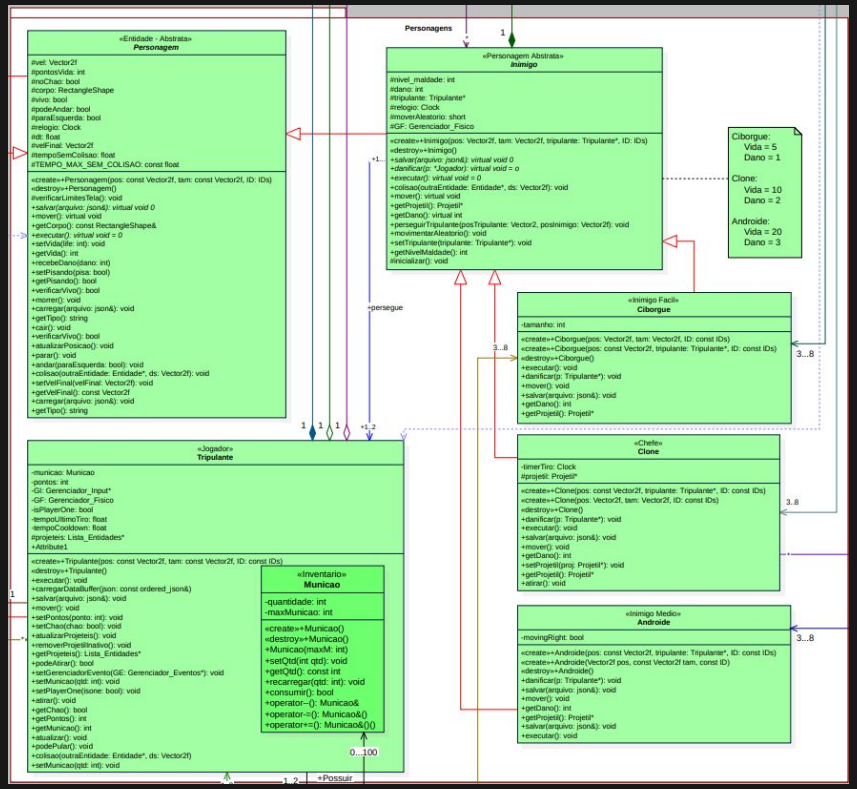
5. DIAGRAMA DE CLASSES



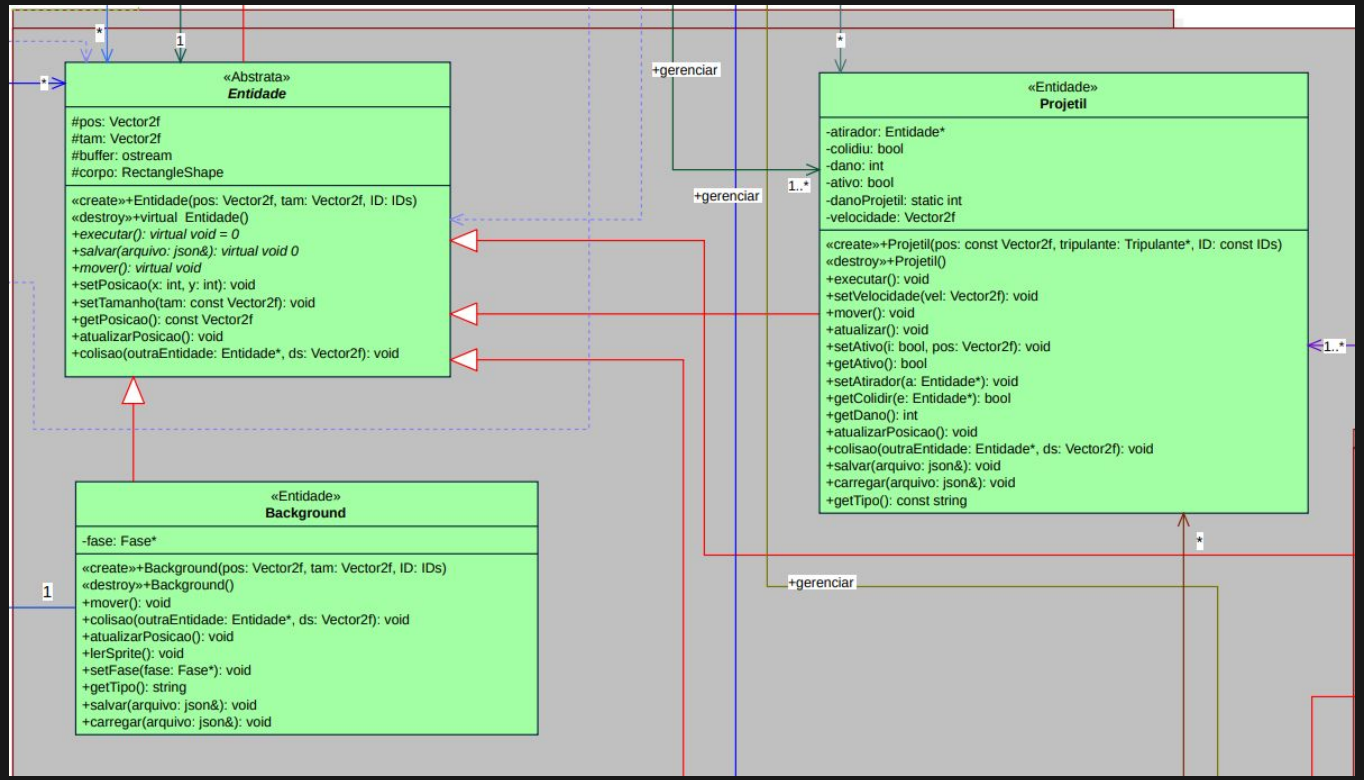
5. DIAGRAMA DE CLASSES



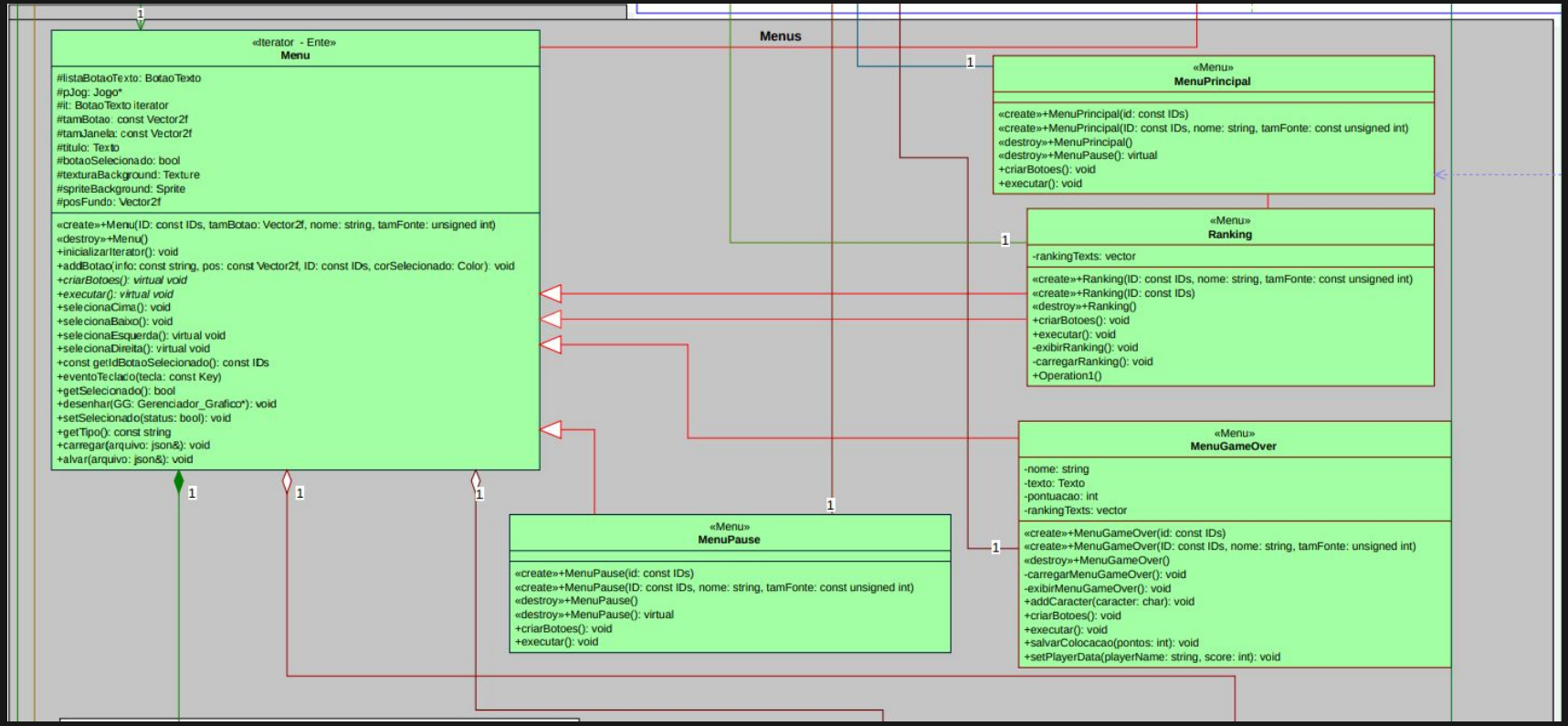
5. DIAGRAMA DE CLASSES



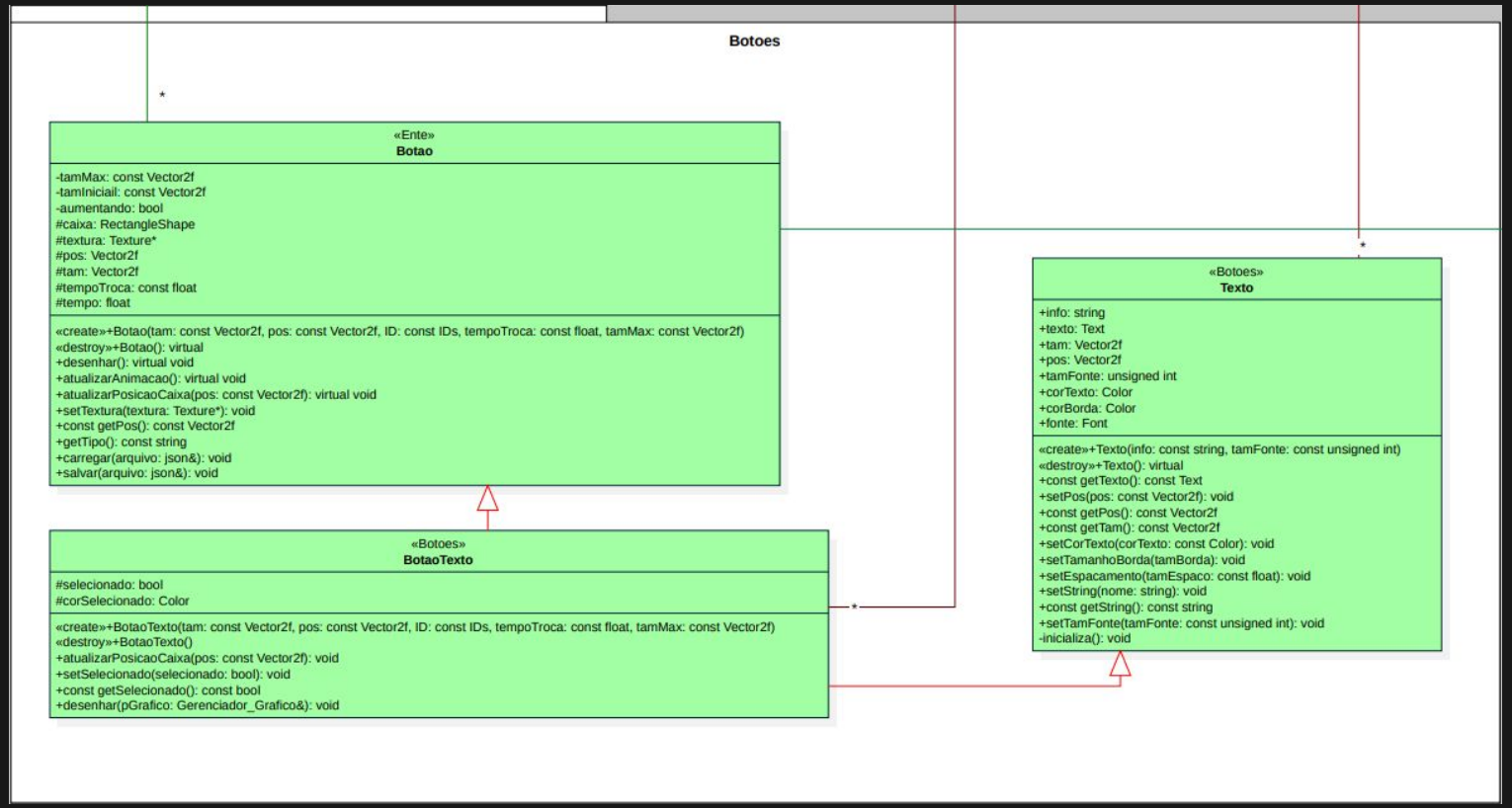
5. DIAGRAMA DE CLASSES



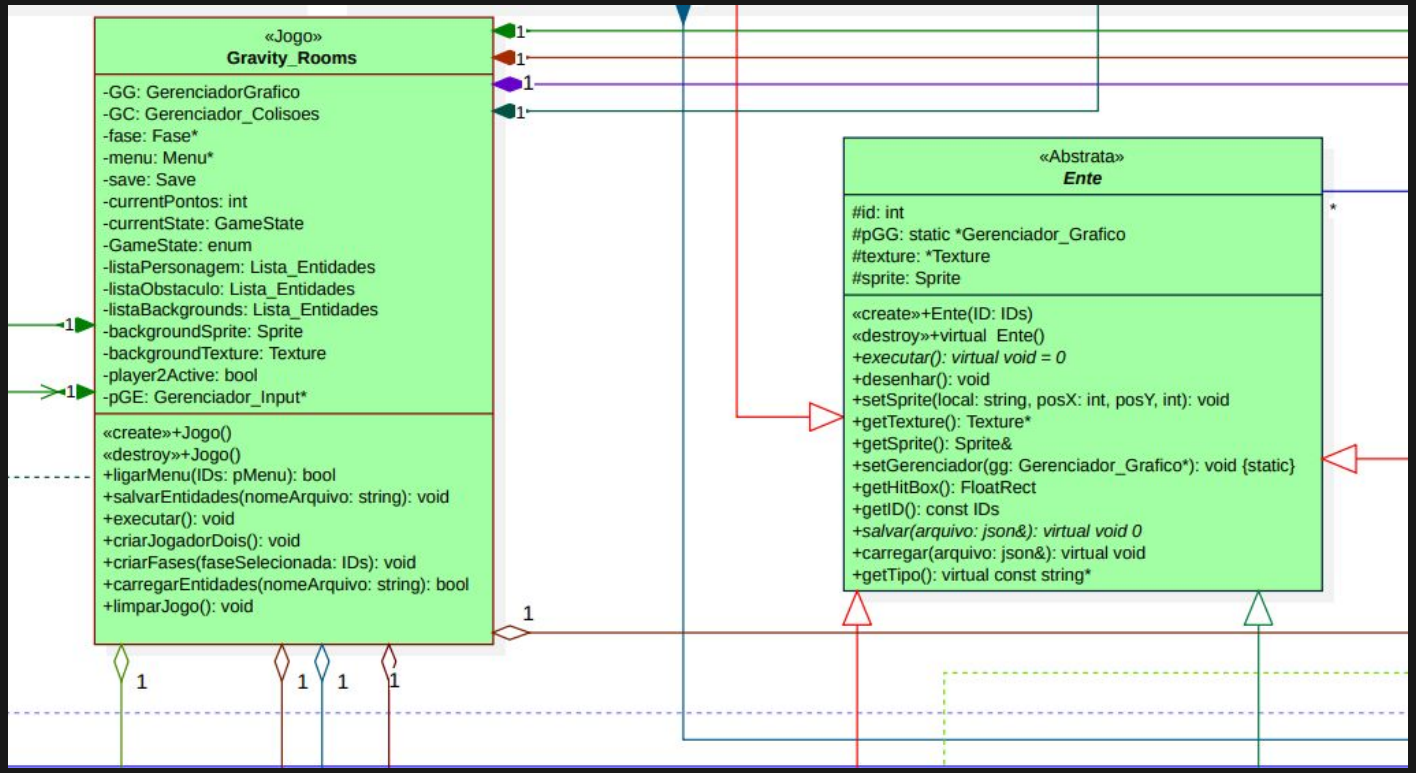
5. DIAGRAMA DE CLASSES



5. DIAGRAMA DE CLASSES



5. DIAGRAMA DE CLASSES



6. EXEMPLOS DO JOGO

6. EXEMPLOS DO JOGO

7. VIDEO GAMEPLAY

8. RESULTADOS E CONCLUSÕES

REQUISITOS

- Funcionais: 100%
- Conceituais: 90%

APRENDIZADO

- Orientação a Objetos
- Engenharia de Software
- Diagrama de Classes
- Padrões de Projeto

REUNIÕES

- Classes Vazias
- Diagnóstico de Erros
- Início de Projeto

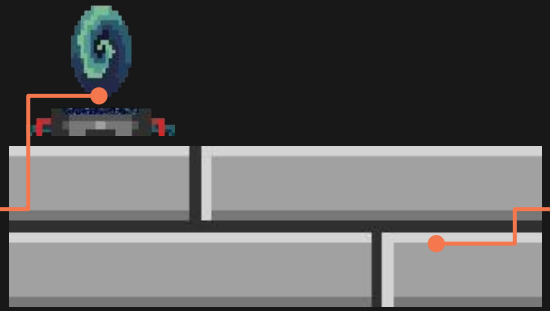
9. AGRADECIMENTOS ESPECIAIS

PETECO

Cursos e dicas

MONITORES

Ajuda com erros e voluntariado férias



PROFESSOR

Métodos, técnicas e conselhos

GRUPOS

(Mossato e Castilhano)
Revisão do projeto

MUITO OBRIGADO!

Para mais informações: