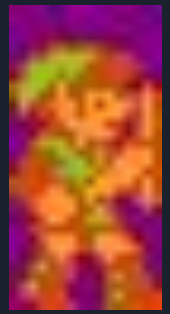
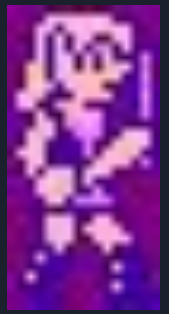
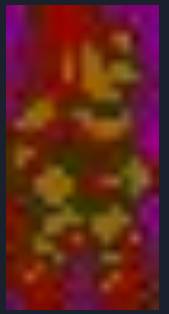
ZELDA++

TecProg S71
Thiago Seiji Miyasawa, Victor Boechat Errera

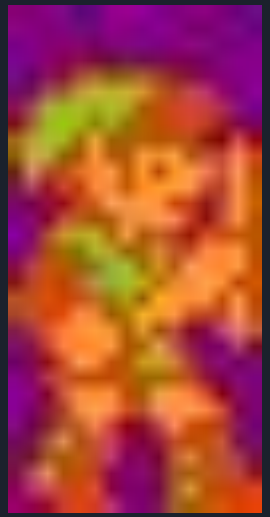
Inspiração

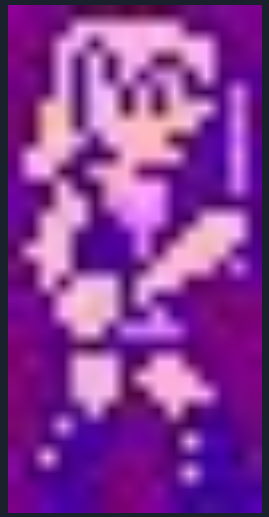
Personagens

Jogadores





pode atacar os inimigos e projéteis

enquanto está atacando ele fica imune ao inimigos e aos projéteis

Moa

é um inimigo voador

fica voando em lado para outro

pode ficar invisível por um um período

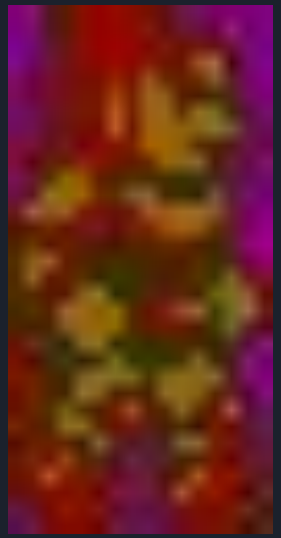
Octorok

é um inimigo voador

ele fica voando de um lado para outro

ele pode atirar projéteis

Ganondorf

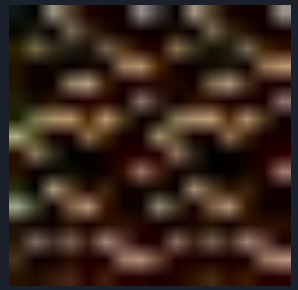
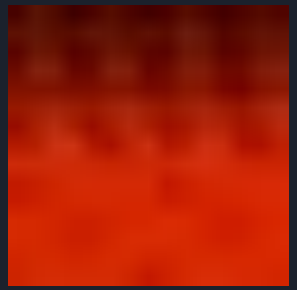


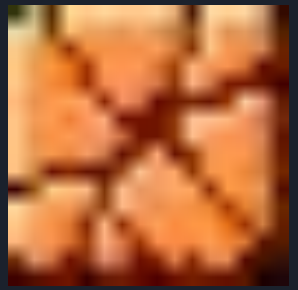
é um boss terrestre

ele fica andando de um lado para outro da fase

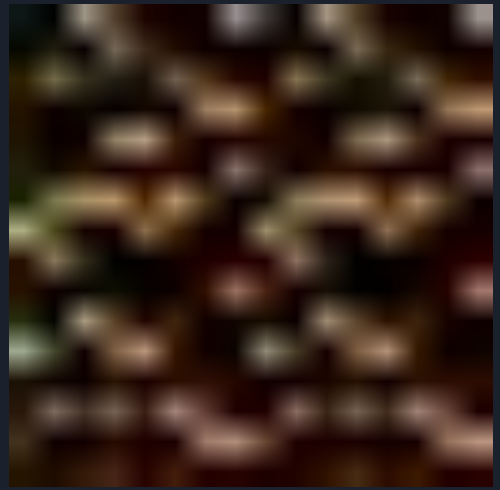
fica irritado de tempos em tempo, nesse estado ele dá o dobro de dano

Obstáculos



Espinhos



e um obstáculo sólido(possui colisão)

ele é danoso, (ou seja se o personagem encostar nesse obstáculo ele da dano)

Lava

obstáculo não sólido

qualquer personagem que encostar nela morre

Plataforma

Obstáculo sólido

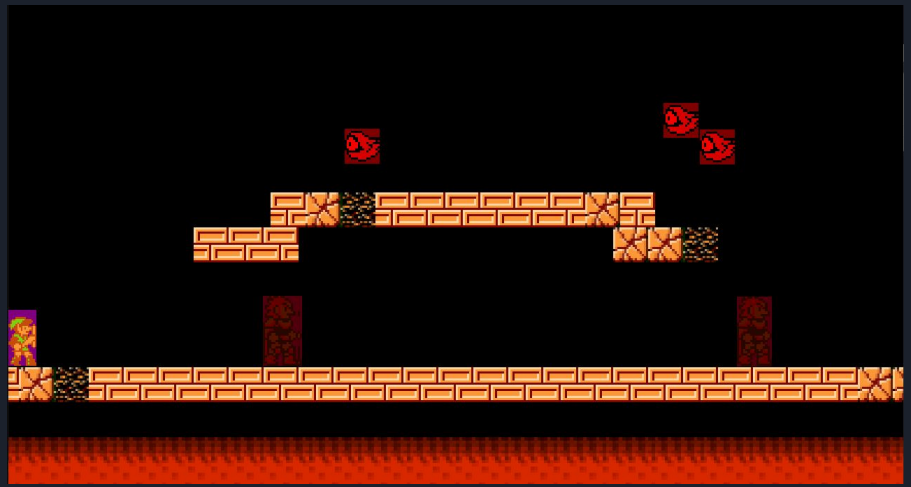
bloco básico da construção das fase

Plataforma Falsa

obstáculo sólido

funciona como uma plataforma comum, porém se um jogador ficar muito tempo em cima dela ela irá desaparecer

Fases



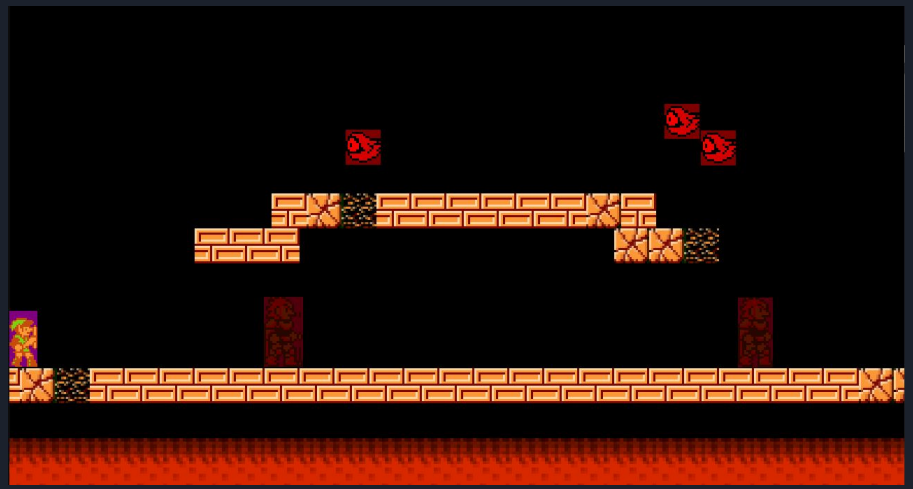
Primeira Fase

possui dois tipos de inimigos a moa e o octorok

possui todos os obstáculos

tem como objetivo matar o máximo de inimigos sem morrer em um limite de tempo

Segunda Fase



possui 2 tipos de inimigos a moa e o ganon(boss)

possui todos os tipos de obstáculos

possui o mesmos objetivo da primeira fase

Requisitos Funcionais

N.	Requisitos Funcionais	Situação	Implementação
1	Apresentar graficamente menu de opções aos usuários do Jogo, no qual pode se escolher fases, ver colocação (*ranking*) de jogadores e demais opções pertinentes.	Requisito previsto inicialmente e realizado.	Requisito cumprido através das classes Menu e ranking 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 seria para que os dois joguem de maneira concomitante.	Requisito previsto inicialmente e realizado	Requisito cumprido via classe Jogador, sendo possível escolher se serão 1 ou 2 através do menu.
3	Disponibilizar ao menos duas fases que podem ser jogadas sequencialmente ou selecionadas, via menu, nas quais jogadores tentam neutralizar inimigos por meio de algum artifício e vice-versa.	Requisito previsto inicialmente e realizado.	Requisito realizado pelas classes Fase, PrimeiraFase e SegundaFase.
4	Ter pelo menos três tipos distintos de inimigos, cada qual com sua representação gráfica, sendo que ao menos um dos inimigos deve ser capaz de lançar projetil contra o(s) jogador(es) e um dos inimigos deve ser um 'Chefão'.	Requisito previsto inicialmente e realizado.	Requisito cumprido através das classes Moa, Octorok, Ganondorf e seus respectivos objetos.
5	Ter a cada fase ao menos dois tipos de inimigos com número aleatório de instâncias, podendo ser várias instâncias e sendo pelo menos 3 instâncias por tipo.	Requisito previsto inicialmente e realizado.	Requisito cumprido inclusive via funções nas classes PrimeiraFase e SegundaFase, sendo eles gerados toda vez que todos os inimigos da tela morrem.
6	Ter três tipos de obstáculos, cada qual com sua representação gráfica, sendo que ao menos um causa dano em jogador se colidirem.	Requisito previsto inicialmente e realizado.	Requisito realizado através das 4 classes derivadas da classe obstáculo.
7	Ter em cada fase ao menos dois tipos de obstáculos com número aleatório de instâncias (*i.e.*, objetos), sendo pelo menos 3 instâncias por tipo.	Requisito previsto inicialmente e realizado.	Requisito realizado pelas funções convertePlatF e converteEsp na classe fase.
8	Ter em cada fase um cenário de jogo constituído por obstáculos, sendo que parte deles seriam plataformas ou similares, sobre as quais pode haver inimigos e podem subir jogadores.	Requisito previsto inicialmente e realizado.	Requisito cumprido inclusive via função virtual pura criaMapa da classe Fase e implementada nas classes derivadas
9	Gerenciar colisões entre jogador para com inimigos e seus projéteis, bem como entre jogador para com obstáculos. Ainda, todos eles devem sofrer o efeito da gravidade no âmbito deste jogo de plataforma vertical e 2D.	Requisito previsto inicialmente e realizado.	Requisito cumprido através da classe GerenciadorColisões e da função gravidade na classe entidade
10	Permitir: (1) salvar nome do usuário, manter/salvar pontuação do jogador (incrementada via neutralização de inimigos) controlado pelo usuário e gerar lista de pontuação (*ranking*). **E** (2) Pausar e **Salvar** Jogada.	Requisito previsto inicialmente e realizado.	Requisito cumprido através das classes Menu , MenuGameOver e Ranking.
Total de requisitos funcionais apropriadamente realizados. *(Cada tópico vale 10%, sendo que para ser contabilizado deve estar realizado efetivamente e não parcialmente)*			**100%** (cem por cento).

Requisito 1


zelda++
iniciar
continuar
ranking

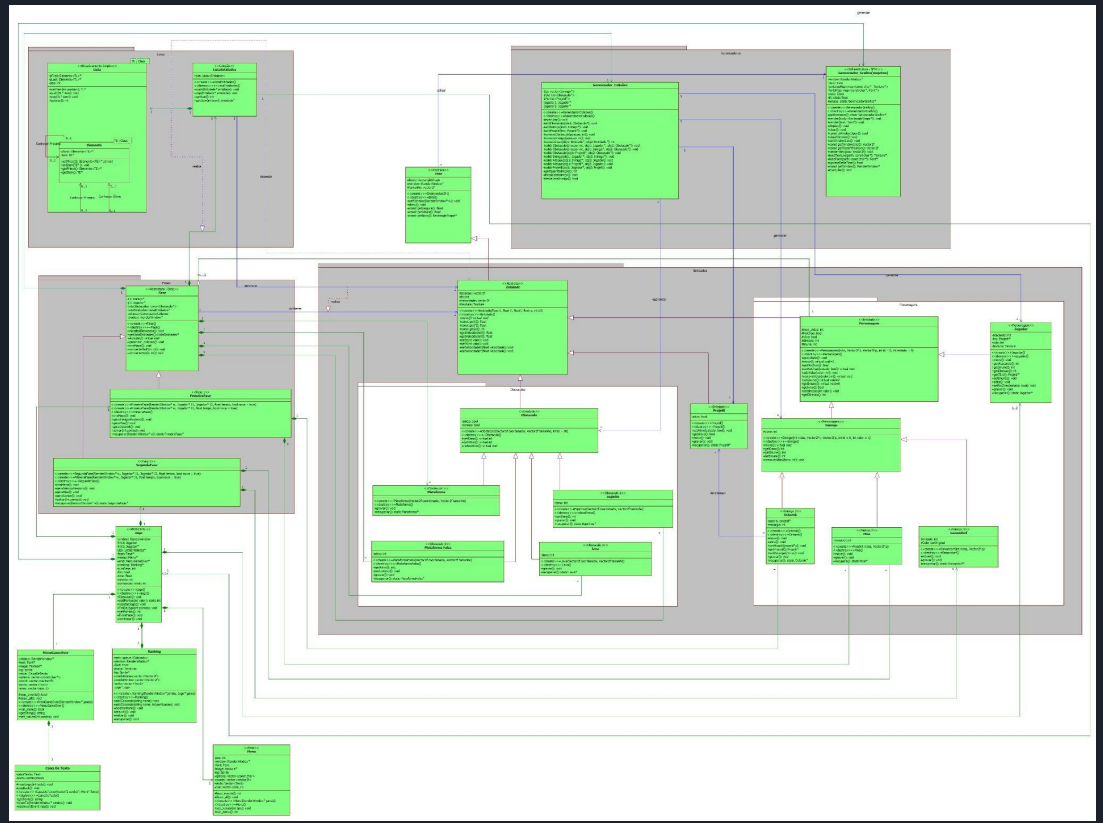

zelda++
1 jogador
2 jogadores


zelda++
Fase 1
Fase 2

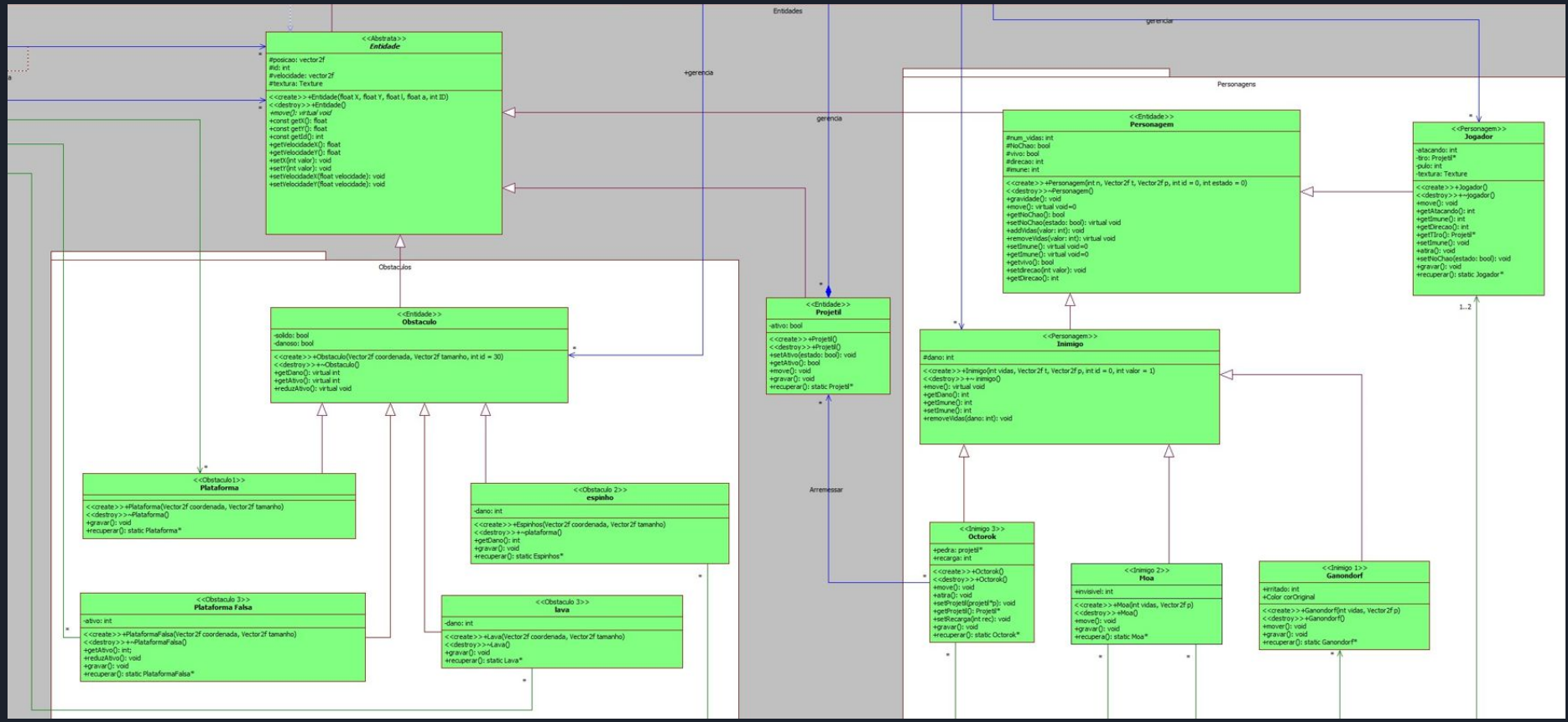
Requisito 2

Requisito 10

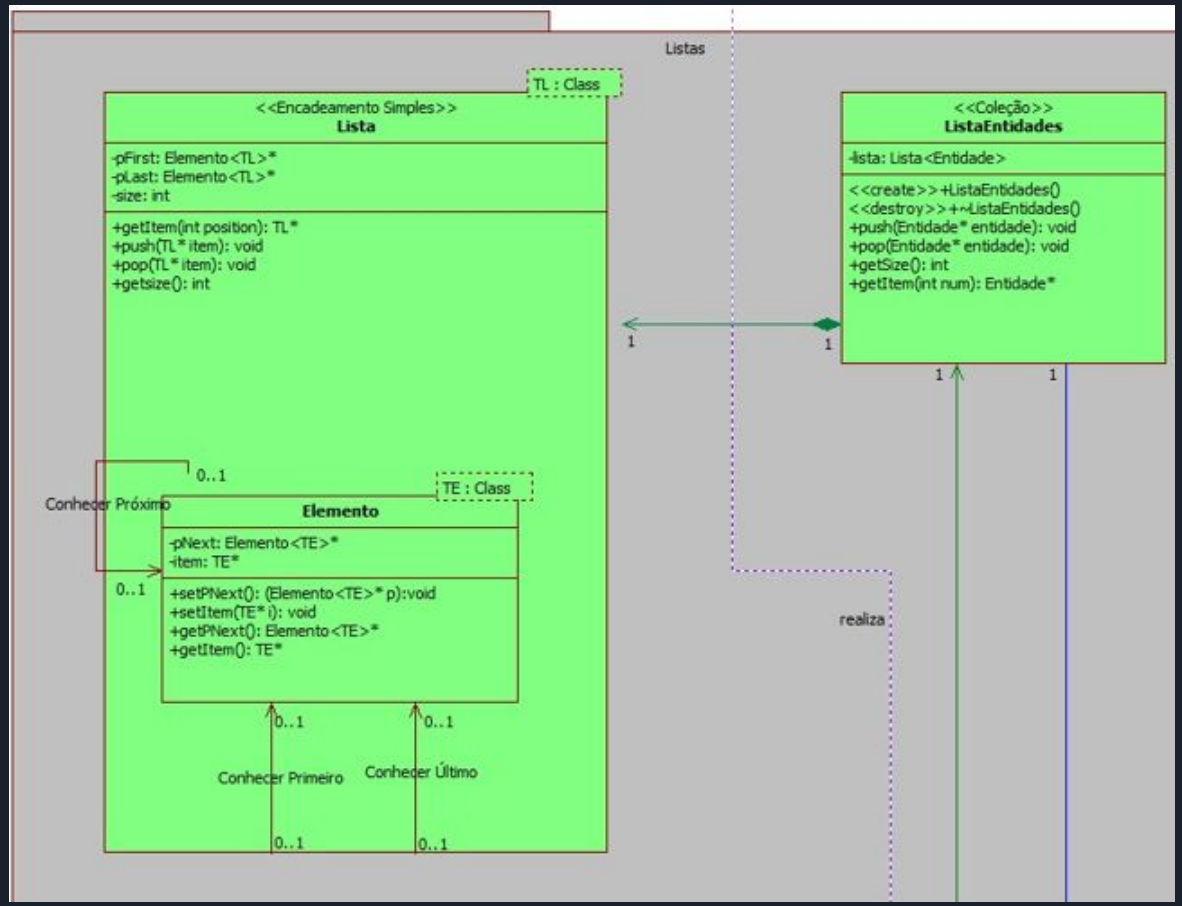
UML



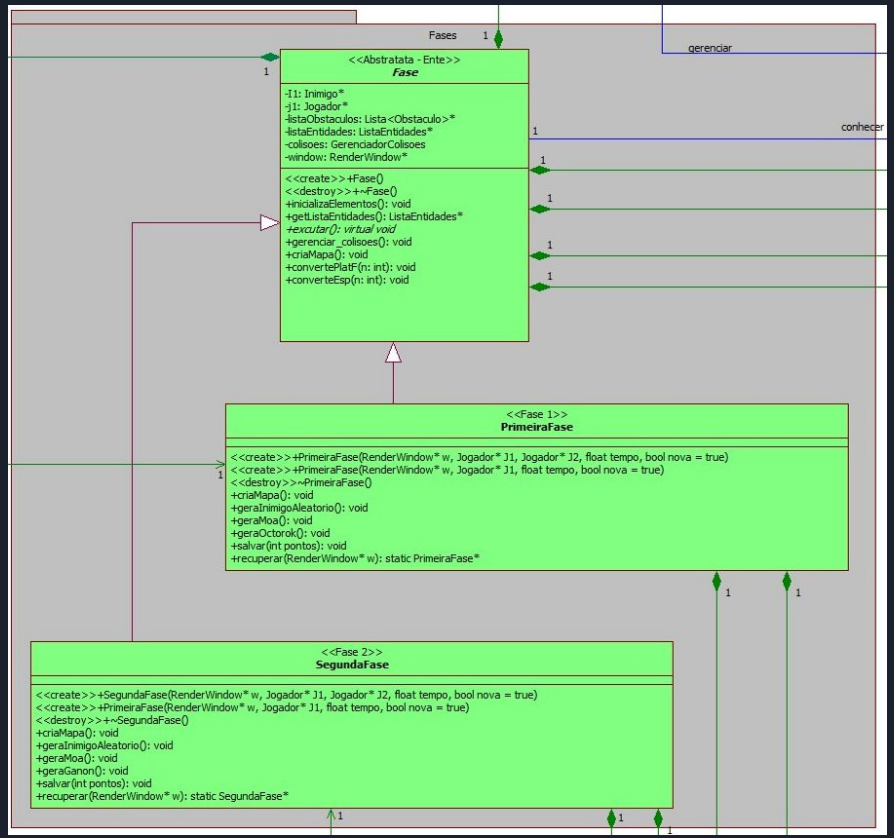
Entidade


Entidades
<<Abstrata>>
Entidade
#posicao: vector2f
#id: int
#velocidade: vector2f
#textura: Texture
<<create>>+Entidade(float X, float Y, float l, float a, int ID)
<<destroy>>+Entidade()
+move(): virtual void
+const getX(): float
+const getY(): float
+const getId(): int
+getVelocidadeX(): float
+getVelocidadeY(): float
+setX(int valor): void
+setY(int valor): void
+setVelocidadeX(float velocidade): void
+setVelocidadeY(float velocidade): void
+gerencia
gerencia
Personagens
<<Entidade>>
Personagem
#num_vidas: int
#NoChao: bool
#vivo: bool
#direcao: int
#imune: int
<<create>>+Personagem(int n, Vector2f t, Vector2f p, int id = 0, int estado = 0)
<<destroy>>~Personagem()
+gravidade(): void
+move(): virtual void=0
+getNoChao(): bool
+setNoChao(estado: bool): virtual void
+addVidas(valor: int): void
+removeVidas(valor: int): virtual void
+setImune(): virtual void=0
+getImune(): virtual void=0
+getvivo(): bool
+setdirecao(int valor): void
+getDirecao(): int
<<Personagem>>
Jogador
-atacando: int
-tiro: Projetil*
-pulo: int
-textura: Texture
<<create>>+Jogador()
<<destroy>>+~Jogador()
+move(): void
+getAtacando(): int
+getImune(): int
+getDirecao(): int
+getTiro(): Projetil*
+setImune(): void
+atira(): void
+setNoChao(estado: bool): void
+gravar(): void
+recuperar(): static Jogador*
1..2
Obstaculos
<<Entidade>>
Obstaculo
-solido: bool
-danoso: bool
<<create>>+Obstaculo(Vector2f coordenada, Vector2f tamanho, int id = 30)
<<destroy>>+~Obstaculo()
+getDano(): virtual int
+getAtivo(): virtual int
+reduzAtivo(): virtual void
<<Entidade>>
Projetil
-ativo: bool
<<create>>+Projetil()
<<destroy>>+Projetil()
+setAtivo(estado: bool): void
+getAtivo(): bool
+move(): void
+gravar(): void
+recuperar(): static Projetil*
<<Personagem>>
Inimigo
#dano: int
<<create>>+Inimigo(int vidas, Vector2f t, Vector2f p, int id = 0, int valor = 1)
<<destroy>>+~inimigo()
+move(): virtual void
+getDano(): int
+getImune(): int
+setImune(): int
+removeVidas(dano: int): void
Arremessar
<<Obstaculo 1>>
Plataforma
<<create>>+Plataforma(Vector2f coordenada, Vector2f tamanho)
<<destroy>>+~Plataforma()
+gravar(): void
+recuperar(): static Plataforma*
<<Obstaculo 2>>
espinho
-dano: int
<<create>>+Espinhos(Vector2f coordenada, Vector2f tamanho)
<<destroy>>+~plataforma()
+getDano(): int
+gravar(): void
+recuperar(): static Espinhos*
<<Obstaculo 3>>
Plataforma Falsa
-ativo: int
<<create>>+PlataformaFalsa(Vector2f coordenada, Vector2f tamanho)
<<destroy>>+~PlataformaFalsa()
+getAtivo(): int;
+reduzAtivo(): void
+gravar(): void
+recuperar(): static PlataformaFalsa*
<<Obstaculo 3>>
lava
-dano: int
<<create>>+Lava(Vector2f coordenada, Vector2f tamanho)
<<destroy>>~Lava()
+gravar(): void
+recuperar(): static Lava*
<<Inimigo 3>>
Octorok
+pedra: projetil*
+recarga: int
<<create>>+Octorok()
<<destroy>>+Octorok()
+move(): void
+atira(): void
+setProjetil(projetil*p): void
+getProjetil(): Projetil*
+setRecarga(int rec): void
+gravar(): void
+recuperar(): static Octorok*
<<Inimigo 2>>
Moa
+invisivel: int
<<create>>+Moa(int vidas, Vector2f p)
<<destroy>>+~Moa()
+move(): void
+gravar(): void
+recuperar(): static Moa*
<<Inimigo 1>>
Ganondorf
+irritado: int
+Color corOriginal
<<create>>+Ganondorf(int vidas, Vector2f p)
<<destroy>>+~Ganondorf()
+mover(): void
+gravar(): void
+recuperar(): static Ganondorf*

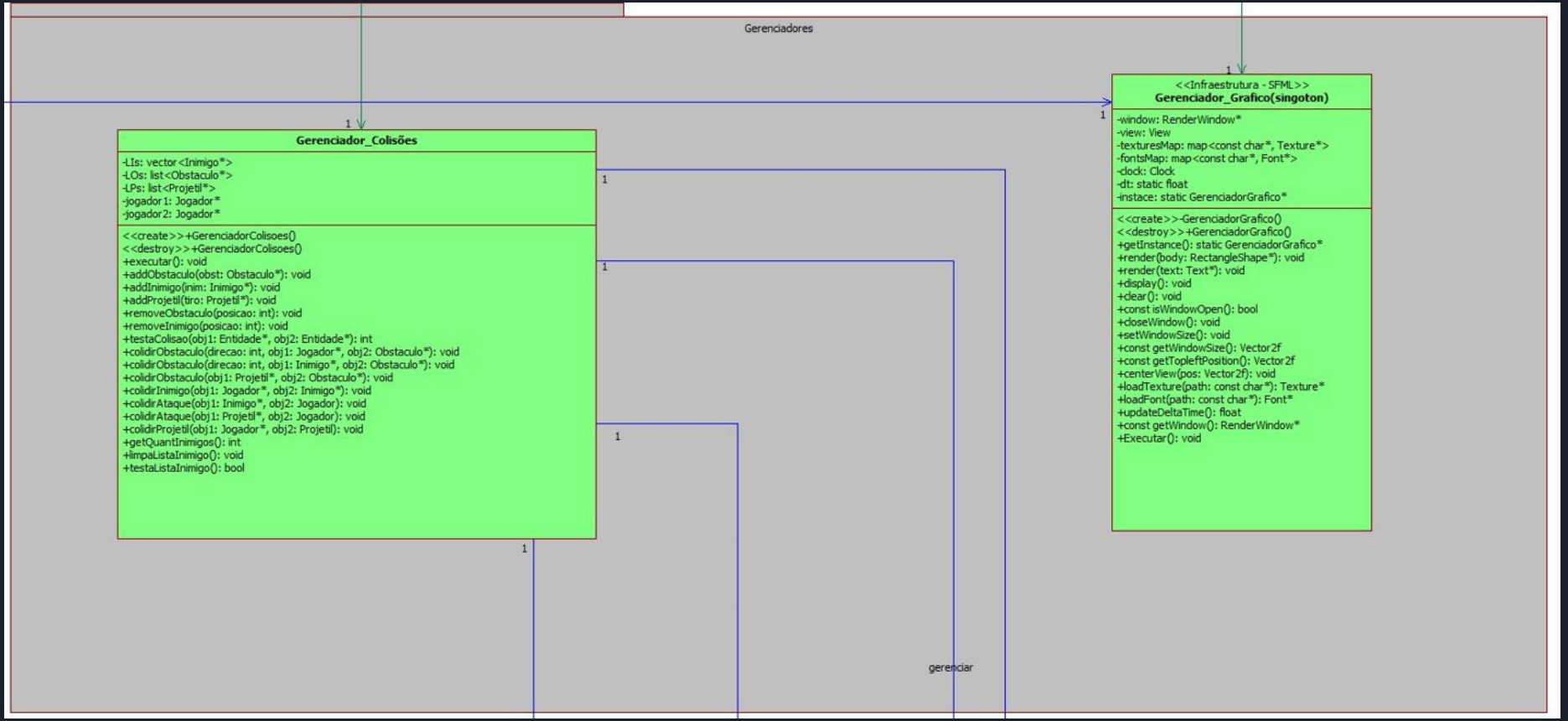
Lista De Entidades



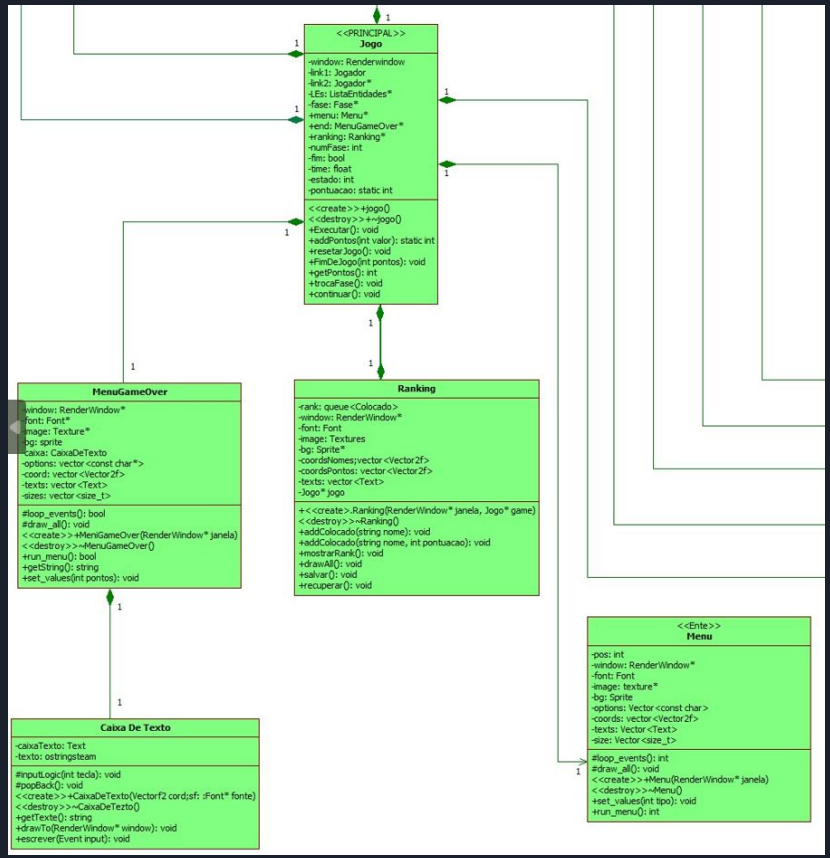
Fase


Fases
1
gerenciar
1
<<Abstratata - Ente>>
Fase
-I1: Inimigo*
-j1: Jogador*
-listaObstaculos: Lista<Obstaculo>*
-listaEntidades: ListaEntidades*
-colisoes: GerenciadorColisoes
-window: RenderWindow*
<<create>>+Fase()
<<destroy>>+~Fase()
+inicializaElementos(): void
+getListaEntidades(): ListaEntidades*
+executar(): virtual void
+gerenciar_colisoes(): void
+criaMapa(): void
+convertePlatF(n: int): void
+converteEsp(n: int): void
conhecer
1
<<Fase 1>>
PrimeiraFase
<<create>>+PrimeiraFase(RenderWindow* w, Jogador* J1, Jogador* J2, float tempo, bool nova = true)
<<create>>+PrimeiraFase(RenderWindow* w, Jogador* J1, float tempo, bool nova = true)
<<destroy>>~PrimeiraFase()
+criaMapa(): void
+geraInimigoAleatorio(): void
+geraMoa(): void
+geraOctorok(): void
+salvar(int pontos): void
+recuperar(RenderWindow* w): static PrimeiraFase*
<<Fase 2>>
SegundaFase
<<create>>+SegundaFase(RenderWindow* w, Jogador* J1, Jogador* J2, float tempo, bool nova = true)
<<create>>+PrimeiraFase(RenderWindow* w, Jogador* J1, float tempo, bool nova = true)
<<destroy>>+~SegundaFase()
+criaMapa(): void
+geraInimigoAleatorio(): void
+geraMoa(): void
+geraGanon(): void
+salvar(int pontos): void
+recuperar(RenderWindow* w): static SegundaFase*

Gerenciadores



Menus


<<PRINCIPAL>>
Jogo
-window: Renderwindow
-link1: Jogador
-link2: Jogador*
-LEs: ListaEntidades*
-fase: Fase*
+menu: Menu*
+end: MenuGameOver*
+ranking: Ranking*
-numFase: int
-fim: bool
-tme: float
-estado: int
-pontuacao: static int
<<create>>+jogo()
<<destroy>>+~jogo()
+Executar(): void
+addPontos(int valor): static int
+resetarJogo(): void
+FimDeJogo(int pontos): void
+getPontos(): int
+trocaFase(): void
+continuar(): void
MenuGameOver
-window: RenderWindow*
-font: Font*
-image: Texture*
-bg: sprite
-caixa: CaixaDeTexto
-options: vector<const char*>
-coord: vector<Vector2f>
-texts: vector<Text>
-sizes: vector<size_t>
#loop_events(): bool
#draw_all(): void
<<create>>+MenuGameOver(RenderWindow* janela)
<<destroy>>~MenuGameOver()
+run_menu(): bool
+getString(): string
+set_values(int pontos): void
Ranking
-rank: queue<Colocado>
-window: RenderWindow*
-font: Font
-image: Textures
-bg: Sprite*
-coordsNomes;vector<Vector2f>
-coordsPontos: vector<Vector2f>
-texts: vector<Text>
-Jogo* jogo
+<<create>.Ranking(RenderWindow* janela, Jogo* game)
<<destroy>>~Ranking()
+addColocado(string nome): void
+addColocado(string nome, int pontuacao): void
+mostrarRank(): void
+drawAll(): void
+salvar(): void
+recuperar(): void
<<Ente>>
Menu
-pos: int
-window: RenderWindow*
-font: Font
-image: texture*
-bg: Sprite
-options: Vector<const char>
-coords: vector<Vector2f>
-texts: Vector<Text>
-size: Vector<size_t>
#loop_events(): int
#draw_all(): void
<<create>>+Menu(RenderWindow* janela)
<<destroy>>~Menu()
+set_values(int tipo): void
+run_menu(): int
Caixa De Texto
-caixaTexto: Text
-texto: ostringsteam
#inputLogic(int tecla): void
#popBack(): void
<<create>>+CaixaDeTexto(Vectorf2 cord;sf: :Font* fonte)
<<destroy>>~CaixaDeTezto()
+getTexte(): string
+drawTo(RenderWindow* window): void
+escrever(Event input): void

Requisitos Conceituais

N.	Conceitos	Uso	Onde / O quê
1	**Elementares:**		
1.1	- Classes, objetos. & - Atributos (privados), variáveis e constantes. & - Métodos (com e sem retorno).	Sim	Todos .h e .cpp, como nas classes Jogador e Menu.
1.2	- Métodos (com retorno *const* e parâmetro *const*). & - Construtores (sem/com parâmetros) e destrutores	Sim	Todas as classes possuem construtoras e destrutoras, e algumas possuem métodos com const como a classe Ente.
1.3	- Classe Principal.	Sim	Classe Jogo instanciada na main.cpp
1.4	- Divisão em .h e .cpp.	Sim	No desenvolvimento como um todo, como na classe Fase.
2	**Relações de:**		
2.1	- Associação direcional. & - Associação bidirecional.	Sim	bidirecional: classe MenuGameOver com a Classe Jogo direcional: classe GerenciadorColisoes com a classe Jogador, por exemplo
2.2	- Agregação via associação. & - Agregação propriamente dita.	Sim	via associação: classe ListaEntidades com a classe Jogo propriamente dita: a classe Ranking com a classe CaixaDeTexto
2.3	- Herança elementar. & - Herança em diversos níveis.	Sim	Em alguns dos .h e .cpp, como nas classes Entidade que herda Ente e na Personagem que herda Entidade
2.4	- Herança múltipla.	Não	Requisito não realizado.
3	**Ponteiros, generalizações e exceções**		
3.1	- Operador *this* para fins de relacionamento bidirecional.	sim	No relacionamento entre MenuGameOver e Jogo
3.2	- Alocação de memória (*new* & *delete*).	sim	Em vários dos .h e .cpp, como na classe Jogo.
3.3	- Gabaritos/*Templates* criada/adaptados pelos autores (*e.g.*, Listas Encadeadas via *Templates*).	sim	Na classe Lista.
3.4	- Uso de Tratamento de Exceções (*try catch*).	não	Requisito não realizado.
	catch).		
4	**Sobrecarga de:**		
4.1	- Construtoras e Métodos.	sim	em várias .h e .cpp, como na classe PrimeiraFase.
4.2	- Operadores (2 tipos de operadores pelo menos – Quais?).	não	Requisito não realizado..
---	**Persistência de Objetos (via arquivo de texto ou binário)**		
4.3	- Persistência de Objetos.	sim	Em várias .h e .cpp, como na classe Jogador
4.4	- Persistência de Relacionamento de Objetos.	sim	Na classe Octorok, o relacionamento com o Projétil.
5	**Virtualidade:**		. . .
5.1	- Métodos Virtuais Usuais.	sim	Em várias classes, como na classe Inimigo.
5.2	- Polimorfismo.	sim	Em vários .cpp, feito através da ListaEntidades, em específico na Fase.
5.3	- Métodos Virtuais Puros / Classes Abstratas.	sim	Em várias classes, como na classe Fase.
5.4	- Coesão/Desacoplamento efetiva e intensa com o apoio de padrões de projeto.	sim	isso pode ser encontrado em vários lugares, um exemplo é a classe CaixaDeTexto que não foi criada dentro da classe MenuGameOver
6	**Organizadores e Estáticos**		
6.1	- Espaço de Nomes (*Namespace*) criado pelos autores.	sim	Em grande parte das classes, como na classe Entidade
6.2	- Classes aninhadas (*Nested*) criada pelos autores.	sim	Na classe Lista
6.3	- Atributos estáticos e métodos estáticos.	sim	Em várias classes, como na função recuperar da classe jogador
6.4	- Uso extensivo de constante (*const*) parâmetro, retorno, método...	sim	em entidades têm a função getX que é uma função constante
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	Na classe CaixaDeTexto

7.2	- Pilha, Fila, Dinha, Fila de Prioridade, Conjunto, Multi-Conjunto, Mapa **OU** Multi-Mapa.	sim	Na classe Ranking foi usada uma fila
---	**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 realizado.
7.4	- *Threads* (Linhas de Execução) no âmbito da Orientação a Objetos com uso de Mutex, Semáforos, **OU** Troca de mensagens.	não	Requisito não realizado.
8	**Biblioteca Gráfica / Visual**		
8.1	- Funcionalidades Elementares. **&** - Funcionalidades Avançadas como: • tratamento de colisões • duplo *buffer*	sim	Em várias .h e .cpp, como na classe GerenciadorGrafico para exibir as Entidades
8.2	- Programação orientada e evento efetiva (com gerenciador apropriado de eventos inclusive) em algum ambiente gráfico. **OU** - *RAD – Rapid Application Development* (Objetos gráficos como formulários, botões etc).	não	Requisito não realizado.
---	**Interdisciplinaridades via utilização de Conceitos de Matemática Contínua e/ou Física.**		
8.3	- Ensino Médio Efetivamente.	sim	uso do conceito de vetores para a manipulação das posições e velocidades de todas as Entidades.
8.4	- Ensino Superior Efetivamente.	não	Especificar quais conceitos aqui.
9	**Engenharia de Software**		
9.1	- Compreensão, melhoria e rastreabilidade de cumprimento de requisitos. &	sim	Feito através do diagrama em UML e das tabelas de requisitos
9.2	- Diagrama de Classes em *UML*.	sim	Feito no starUML
9.3	- Uso efetivo e intensivo de padrões de projeto *GOF*, *i.e.*, mais de 5 padrões.	não	Requisito não realizado.
9.4	- Testes à luz da Tabela de Requisitos e do Diagrama de Classes.	sim	Feito ao longo do desenvolvimento para garantir o cumprimento dos requisitos
10	**Execução de Projeto**		
10.1	- Controle de versão de modelos e códigos automatizado (via github e/ou afins). & - Uso de alguma forma de cópia de segurança (*i.e.*, *backup*).	sim	via github : https://github.com/ErreraV/jogo-TecProg
10.2	- Reuniões com o professor para acompanhamento do andamento do projeto.	sim	4 reuniões dias 3/11, 10/11, 17/11 e 24/11
10.3	- Reuniões com monitor da disciplina para acompanhamento do andamento do projeto.	não	Requisito não realizado.
10.4	- Revisão do trabalho escrito de outra equipe e vice-versa.	sim	Ian Ishikawa e Pedro Neves
Total de conceitos apropriadamente utilizados. *(Cada grande tópico vale 10% do total de conceitos. Assim, por exemplo, caso se tenha feito metade de um tópico, então valeria 5%.)*			**77,5%** (setenta e sete e meio por cento).

conclusões

- apesar do ritmo lento no início do projeto, foi possível cumprir grande parte dos requisitos.
- com esse projeto foi possível sentir um pouco do que nos aguarda mais para frente no curso e na vida profissional
- com esse projeto foi possível aprender muitas coisas.
- esse jogo ainda tem muitas coisas que podem melhorar, entretanto fizemos o que foi possível com o que tinham a nossa disposição no cenário atual