TRABALHO PARA A DISCIPLINA DE   
TÉCNICAS DE PROGRAMAÇÃO DO CURSO DE   
ENGENHARIA DE COMPUTAÇÃO DA UTFPR:   
*DESCENT INTO MADNESS*

Lucas Eduardo Bonancio Skora, Thiago de Mendonça Mildemberger

lucasskora@utfpr.edu.br, thimil@utfpr.edu.br

Disciplina: **Técnicas de Programação – CSE20**/S71 – 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*, no formato de um jogo de plataforma, para fins de aprendizado de técnicas de engenharia de *software*, além de programação orientada a objetos em C++. Para tal, desenvolveu-se o jogo *Descent into Madness*, no qual o jogador principal controla um explorador que busca escapar de um templo antigo repleto de monstros. Para o desenvolvimento do jogo foram considerados os requisitos propostos pelo professor e elaborada modelagem (análise e projeto) usando como recurso o Diagrama de Classes em Linguagem de Modelagem Unificada (*Unified Modeling Language* – *UML*). Subsequentemente, o projeto foi desenvolvido em linguagem de programação C++, que contemplou os conceitos usuais de Orientação a Objetos como Classe, Objeto e Relacionamento, bem como alguns conceitos 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 partidas do jogo demonstraram sua funcionalidade conforme os requisitos e a modelagem. Por fim, destaca-se que o projeto em questão permitiu cumprir o objetivo de aumentar a experiência e o conhecimento dos discentes.

**Palavras-chave ou Expressões-chave:** Paradigma Orientado a Objetos; C++; Desenvolvimento de jogos digitais.

**Abstract** -

*Programing Techniques is a subject that requires the development of software in the shape of a platform videogame with the objective of learning software engineering techniques, as well as object-oriented programming in C++. For that reason, the game Descent into Madness was developed. In this game, the main player controls an explorer trying to escape from an ancient temple full of monsters. All proposed requirements were considered to develop the game, which was modeled (analysis and project) using the Unified Modeling Language Class Diagram (UML). Subsequently, the software was developed using C++ programming language and usual Object Orientation concepts such as classes, objects, relationships, as well as some advanced concepts, such as abstract classes, polymorphism, templates, object persistence through files, operator overload, and the Standard Template Library (STL). After implementing, testing and playing the game, the developers showed its functionality according to requirements and modeling. At last, the project met the desired goals of improving the students’ knowledge and experience.*

**Keywords or Key-expressions:** Object Oriented Paradigm; C++; Videogame development.

# INTRODUÇÃO

O presente documento busca relatar o desenvolvimento de um projeto para a disciplina de Técnicas de Programação, matéria obrigatória do curso de Engenharia da Computação na UTFPR. Seu objetivo é aplicar, avaliar e expandir os conhecimentos adquiridos durante as aulas dessa disciplina, nomeadamente, o Paradigma Orientado a Objetos, a linguagem de programação C++ e uma introdução à Engenharia de Software. Esse documento, o artefato de software que ele descreve e o diagrama de classes UML produzido são baseados em modelos disponíveis no site do professor da disciplina [1].

Com esses objetivos, foi desenvolvido um jogo de plataforma com tema e mecânicas escolhidas pelos discentes e aprovadas pelo docente, visando seguir os requisitos previamente determinados, que serão descritos em detalhes no presente documento.

Para fazer isso, os alunos buscaram seguir o ciclo clássico de engenharia de software, ou seja, com a orientação do professor e do monitor da disciplina, estudaram os requisitos necessários, modelaram o software através de um diagrama de classes UML, implementaram o programa na linguagem C++ (com uso da biblioteca SFML) e testaram o programa tão extensivamente quanto possível.

As seções a seguir explicam o jogo, sua implementação e os conceitos utilizados.

# O JOGO

O jogo *Descent into Madness* é um jogo de plataforma em terceira pessoa para um ou dois jogadores. Quando o programa é executado, o usuário visualiza o menu principal. Existe também o menu de pausa, que pode ser acessado apertando a tecla ESC dentro de uma fase.

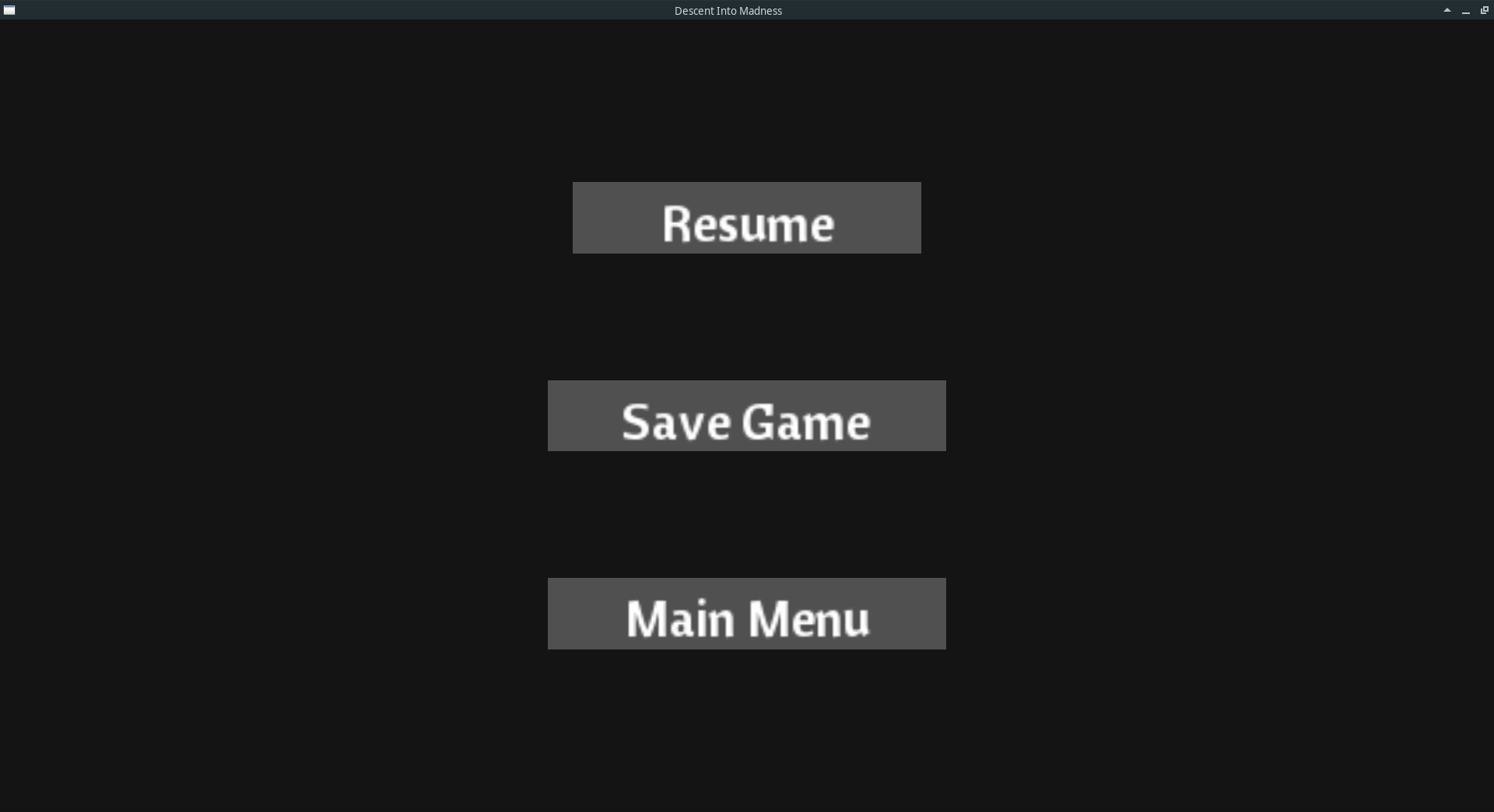


Figura 1: Menu principal. Figure 2: Menu de pausa.

O jogador deve chegar ao final de cada fase sem tocar nenhum inimigo, obstáculo ou projétil, que significam morte instantânea e retorno ao início. O jogador que conseguir terminar o jogo com menos mortes será aquele com melhor classificação.

Cada vez que o jogador joga uma nova fase, a posição e número dos inimigos e a existência de certos obstáculos mudam aleatoriamente, de forma que cada jogada é diferente.

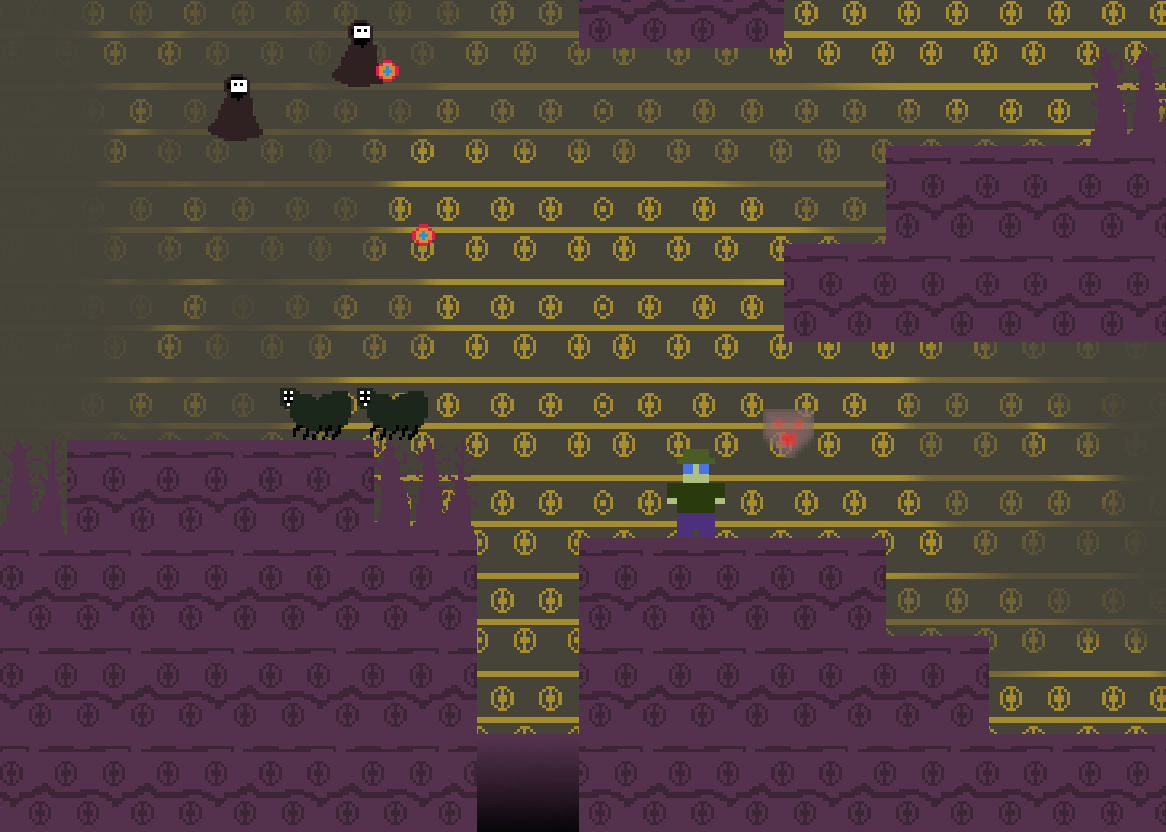
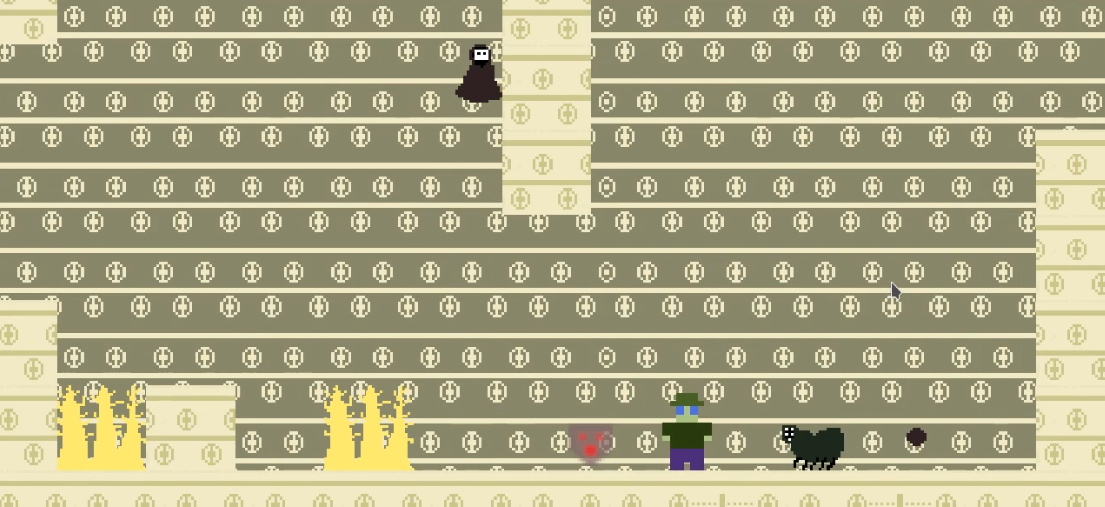
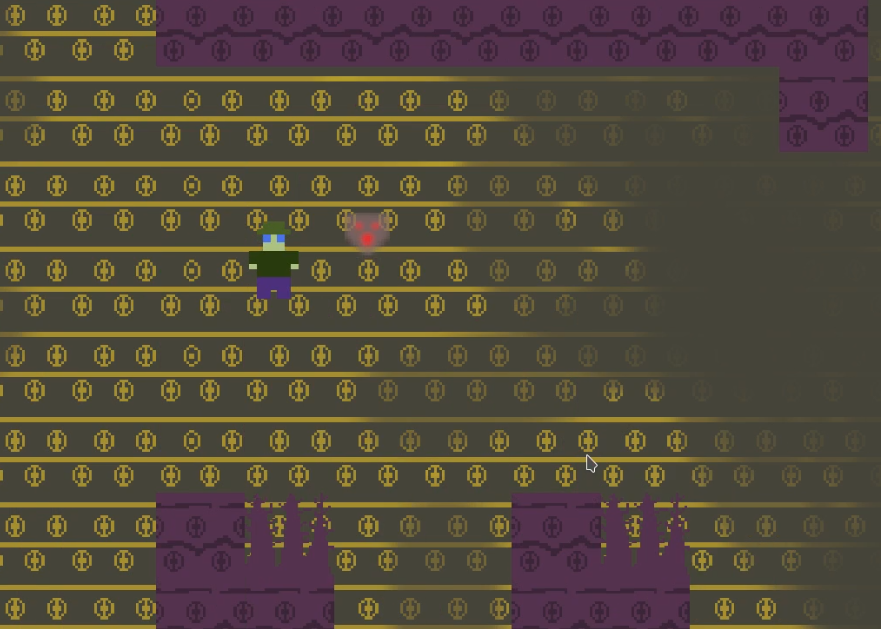


Figura 3: Seção da fase 1 (templo) Figura 4: Seção da fase 2 (caverna).



Figuras 5 e 6: duas configurações possíveis de uma seção da fase Caverna.

O primeiro jogador pode se mover para os lados e pular. O segundo jogador, que tem a função de proteger o primeiro dos projéteis atirados, gira em torno do primeiro jogador.

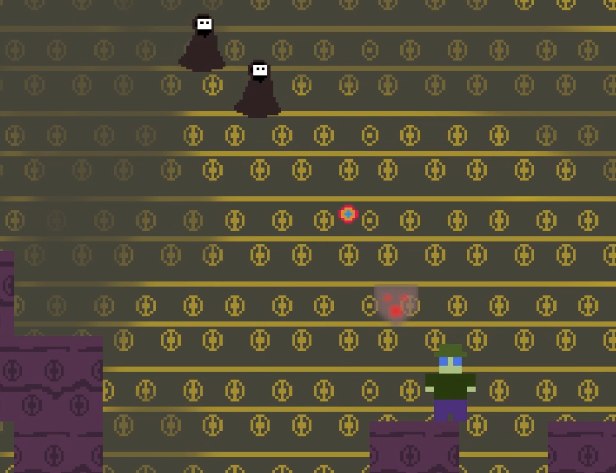
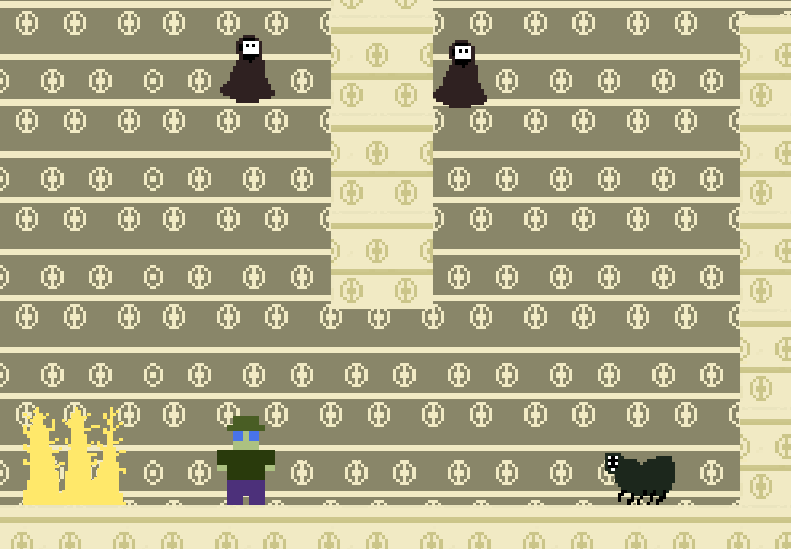


Figura 7: Jogador 1 sozinho. Figura 8: Jogador 2 defendendo o jogador 1 de um feitiço.

O chefão (*The Chained*), que espera o jogador no final da fase 2, é quase impossível de passar sem antes pegar o item *The Mirror of Hastur*, que torna o chefão impotente.



Figuras 9, 10 e 11: Chefão atacando o jogador, jogador pegando o item e chefão inerte frente ao jogador com o item.

Ao terminar a última fase, o jogador entra no menu de classificação, onde pode escrever seu nome e ver a pontuação dos outros jogadores.

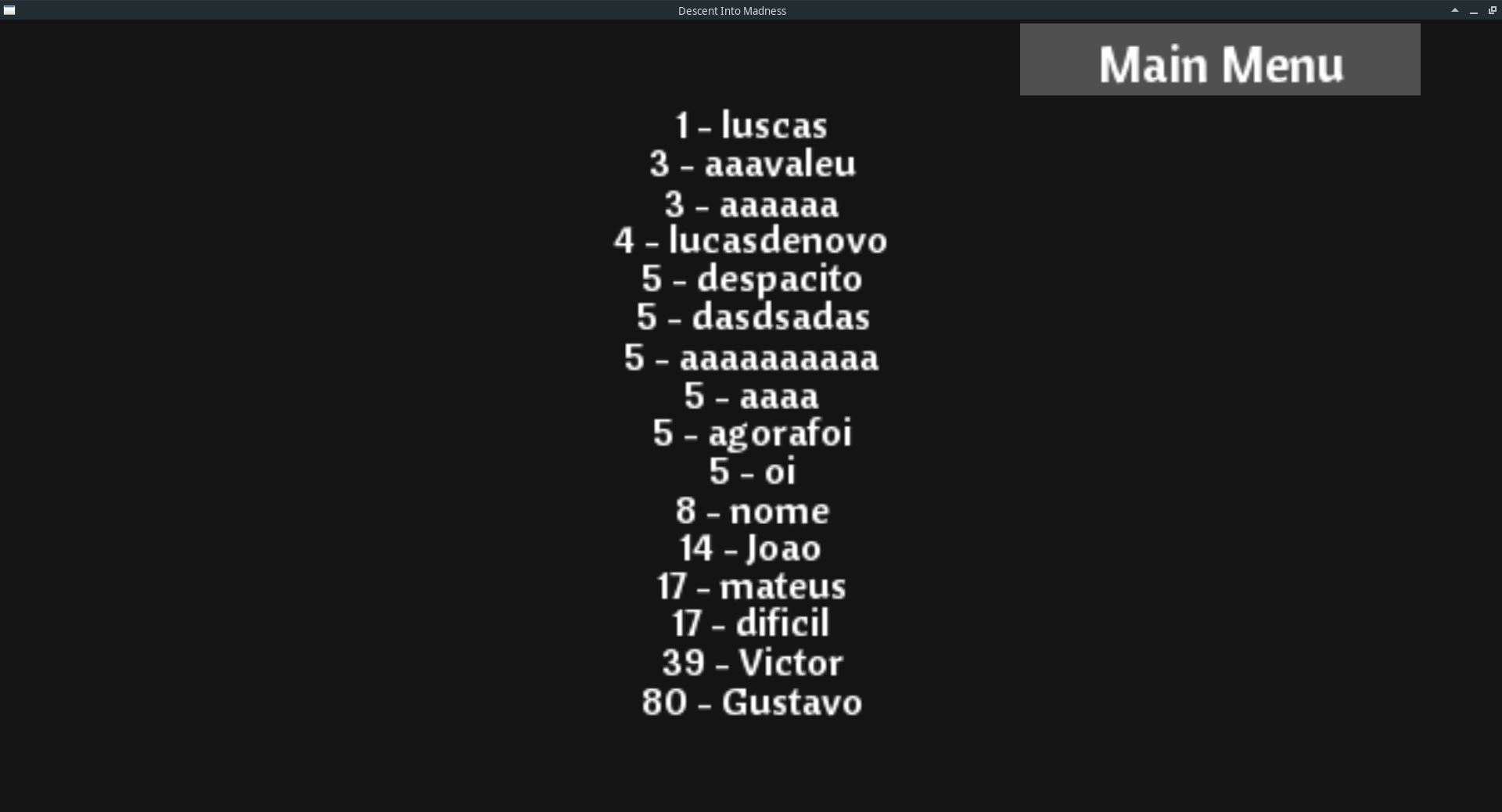


Figura 12: Menu de classificação dos jogadores.

# DESENVOLVIMENTO DO JOGO NA VERSÃO ORIENTADA A OBJETOS

Tabela 1. Lista de requisitos do jogo e suas situações.

|  |  |  |  |
| --- | --- | --- | --- |
| N. | Requisitos funcionais | Situação | Implementação |
| 1 | Apresentar menu de opções aos usuários do Jogo. | Requisito previsto inicialmente e realizado. | Cumprido via classe MainMenu e Button e seus respectivos objetos. |
| 2 | Permitir um ou dois jogadores aos usuários do Jogo, sendo que no último caso seria para que os dois joguem de maneira concomitante. | Requisito previsto inicialmente e realizado. | Cumprido via objetos das classes TheUndying, ThePenitent e MainMenu. |
| 3 | Disponibilizar ao menos duas fases que podem ser jogadas sequencialmente ou selecionadas. | Requisito previsto inicialmente e realizado. | Cumprido via pacote Levels e pelo objeto da classe MainMenu, onde o usuário pode escolher a fase. |
| 4 | Ter três tipos distintos de inimigos (o que pode incluir ‘Chefão’, vide abaixo), sendo que pelo menos um dos inimigos deve ser capaz de lançar projetil contra o(s) jogador(es). | Requisito previsto inicialmente e realizado. | Requisito cumprido através dos pacotes Enemies e Projectiles, ambos contidos no pacote Entities. |
| 5 | Ter a cada fase ao menos dois tipos de inimigos com número aleatório de instâncias, podendo ser várias instâncias e sendo pelo menos 5 instâncias por tipo. | Requisito previsto inicialmente e realizado. | Cumprido a partir do pacote Enemies, dos objetos de classes derivadas da classe Level e dos objetos da classe TileManager. |
| 6 | Ter inimigo “Chefão” na última fase | Requisito previsto inicialmente e realizado. | Cumprido a partir da classe TheChained e seu objeto. |
| 7 | Ter três tipos de obstáculos. | Requisito previsto inicialmente e realizado. | Cumprido através do pacote TileSystem. |
| 8 | Ter em cada fase ao menos dois tipos de obstáculos com número aleatório de instâncias (i.e., objetos) sendo pelo menos 5 instâncias por tipo. | Requisito previsto inicialmente e realizado. | Cumprido através dos objetos da classe Tile e BulletObstacle, além de objetos da classe TileManager[[1]](#footnote-2) |
| 9 | Ter representação gráfica de cada instância. | Requisito previsto inicialmente e realizado. | Cumprido através do objeto da classe GraphicsManager. |
| 10 | Ter em cada fase um cenário de jogo com os obstáculos. | Requisito previsto inicialmente e realizado. | Cumprido através do uso do pacote TileSystem e pelos objetos de classes derivadas de Level. |
| 11 | Gerenciar colisões entre jogador e inimigos, bem como seus projéteis (em havendo). | Requisito previsto inicialmente e realizado. | Cumprido pelo uso de objetos das classes CollisionManager e TileManager, gerenciando objetos de classes derivadas de PhysicalEntity e Tile. |
| 12 | Gerenciar colisões entre jogador e obstáculos. | Requisito previsto inicialmente e realizado. | Idem item anterior. |
| 13 | Permitir cadastrar/salvar dados do usuário, manter pontuação durante jogo, salvar pontuação e gerar lista de pontuação (*ranking*). | Requisito previsto inicialmente e realizado. | Requisito cumprido via objeto da classe TheUndying (jogador 1) e pacote Menus. |
| 14 | Permitir Pausar o Jogo | Requisito previsto inicialmente e realizado | Requisito cumprido através dos objetos das classes EventManager e PauseMenu. |
| 15 | Permitir Salvar Jogada. | Requisito previsto inicialmente e realizado. | Requisito cumprido através do objeto da classe PauseMenu e pacote Mementos. |

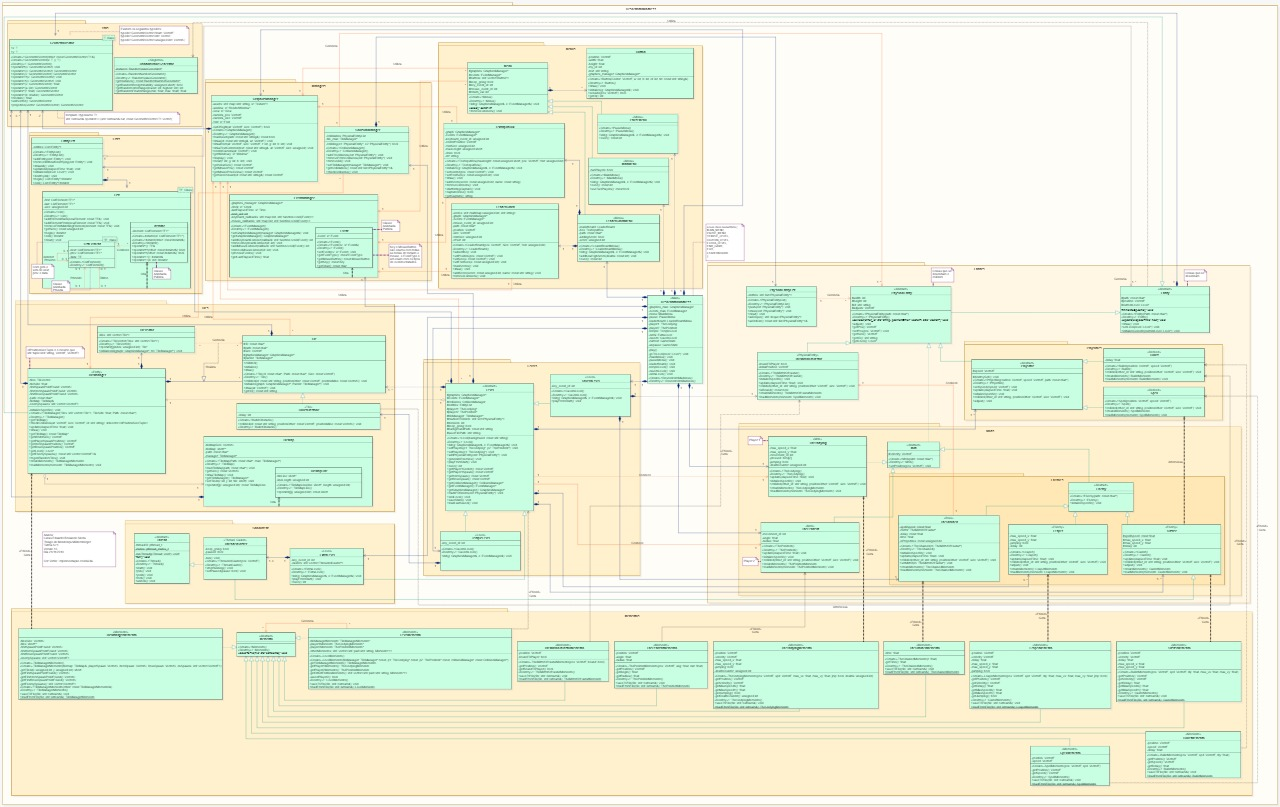


Figura 13: Diagrama de Classes UML.[[2]](#footnote-3)

A classe principal do jogo é DescentIntoMadness, que recebe seu nome. O objeto dessa classe agrega as fases, menus e jogadores, além dos objetos das classes EventManager, CollisionManager e GraphicsManager. Ela funciona, em resumo, como uma máquina de estados, executando uma fase ou menu de acordo com o valor da variável current. Para manter a coerência da explicação com a organização real do programa, a explicação das demais classes se dará em blocos dedicados a cada pacote.

GraphicsManager é a classe responsável por todos os elementos gráficos do jogo, interagindo diretamente com a biblioteca gráfica utilizada SFML. Ela tem métodos para carregar (loadAsset) e imprimir imagens, retângulos e texto, de forma que nenhum objeto precisa interagir realmente com a SFML para ser desenhado.

A classe EventManager é responsável por lidar com todos os eventos relacionados ao mouse e teclado, chamando as funções “inscritas” pelos objetos sempre que um evento ocorre. Vale notar que essa classe e a descrita no parágrafo anterior são as únicas que tem alguma relação direta com a biblioteca gráfica escolhida, o que significa que seria fácil transformá-las em classes abstratas e trocar a biblioteca gráfica.

A classe CollisionManager gerencia as colisões das entidades físicas entre si e com o mapa. Quando seu método checkCollisions é chamado, ela verifica quais entidades colidiram com quais outras comparando suas posições e tamanhos e então verifica se estão colidindo com o mapa chamando o método checkCollisions da classe TileManager. Se houver colisões, os métodos collided da PhysicalEntity e/ou Tile relevante é chamado.

O pacote Entities comporta todos os objetos que fazem parte dos níveis, interagem entre si e precisam ser desenhados. Isso inclui os jogadores, projéteis, inimigos, o item e o TileManager, todas estendendo a classe Entity, que possui posição, uma string com o caminho para uma imagem e consegue se desenhar.

A classe PhysicalEntity, também abstrata, é estendida por entidades que precisam colidir com as outras, possuindo getters e setters para sua posição e tamanho e um método virtual puro (collided) que é chamado quando o objeto colide com outro.

Em resumo, os objetos das classes Level gerenciam as entidades que compõe uma fase. Eles são responsáveis por criar (salvo os objetos dos players, que são incluídos diretamente na classe principal por exigirem consistência de uma fase para outra), inicializar e chamar os métodos de desenho e atualização de suas entidades, além de salvá-las/carregá-las de arquivos através do pacote Mementos, que será abordado posteriormente.

Os menus mantêm desenhadas uma lista de botões e reagem apropriadamente quando algum deles é clicado através do EventManager e do GraphicsManager. LeaderBoardMenu mantem um objeto da classe TextInputBox, que permite ao usuário escrever seu nome para cadastrar sua pontuação, e um da classe LeaderBoard, que carrega as pontuações de um arquivo e as mostra na tela ordenadas.

O pacote utils é composto das classes RandomValueGenerator, um singleton que gera valores aleatórios e GeometricVector, gabarito que agrupa dois valores do mesmo tipo como coordenadas x e y. O pacote Lists é composto por um gabarito de lista duplamente encadeada com iterador e uma lista desacoplada de entidades.

O pacote TileSystem gira em torno da classe TileManager, que é, como dito anteriormente, derivada de Entity. Seus objetos agregados principais são um TileMap, que gerencia uma matriz representando o mapa de tiles; e um TileVector, que armazena vários Tiles acessíveis por índice. O mapa de Tiles, o TileManager, percorre o mapa e chama a função desenhar do tile armazenado em TileVector com o índice indicado pelo TileMap e, para verificar colisões, transforma as coordenadas e tamanho de cada entidade em coordenadas do TileMap e verifica se existe algum tile naquela posição.

O pacote Mementos é responsável pela persistência de objetos e seus relacionamentos. Para isso, todo objeto que tem que ter seu estado salvo é provido de um método para gerar um objeto de uma classe que estende Memento e um para restaurar o estado salvo. O salvamento e posterior carregamento são feitos pelos objetos de classes que estendem Level através de objetos da classe LevelMemento que salvam as informações contidas em cada Memento em uma linha de um arquivo.

O pacote Concurrent engloba a classe Thread, baseada na implementação do professor Jean Marcelo Simão[1], ThreadedCaster, que extende Caster (um dos inimigos do jogo) e Thread, e ExtraLevel, uma classe que estende Level e demonstra o funcionamento do ThreadedCaster.

# TABELA DE CONCEITOS UTILIZADOS E NÃO UTILIZADOS

Tabela 2. Lista de conceitos utilizados e não utilizados no trabalho.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Conceito | Uso | Onde/ O quê |  |
| 1 | **Elementares:** | | | |
| - Classes, objetos. &  - Atributos (privados), variáveis e constantes. &  - Métodos (com e sem retorno). | Sim | Todos .hpp e .cpp | |
| - Métodos (com retorno *const* e parâmetro *const*). &  - Construtores (sem/com parâmetros) e destrutores. | Sim | Todos .hpp e .cpp | |
| - Classe principal. | Sim | Main.cpp & DescentIntoMadness.hpp/.cpp | |
| - Divisão em .h e .cpp. | Sim | No desenvolvimento como um todo. | |
| 2 | **Relações de:** | | | |
| - Associação direcional. &  - Associação bidirecional. | Sim | Bidirecional entre CollisionManager e PhysicalEntity, direcional entre Level e GraphicsManager (por exemplo) | |
| - Agregação via associação. &  - Agregação propriamente dita. | Sim | Via associação entre Tile (e derivadas) e TileVector, propriamente dita entre Entity e GeometricVector. | |
| - Herança elementar. &  - Herança em diversos níveis. | Sim | Elementar entre Entity e TileManager, em múltiplos níveis entre Entity e Mob. | |
| - Herança múltipla. | Sim | Classe ThreadedCaster | |
| 3 | **Ponteiros, generalizações e exceções** | | | |
| - Operador *this*. | Sim | Em vários momentos. | |
| - Alocação de memória (*new* & *delete*). | Sim | Em vários momentos. | |
| - Gabaritos/*Templates* criada/adaptados pelos autores (e.g. Listas Encadeadas via *Templates*). | Sim | Classe List e GeometricVector. | |
| - Uso de Tratamento de Exceções (*try catch*). | Sim | Classe TileManager (durante o desenvolvimento). | |
| 4 | **Sobrecarga de:** | | | |
| - Construtoras e Métodos. | Sim | Classe GeometricVector (construtores) e Level (método bindPlayer). | |
| - Operadores (2 tipos de operadores pelo menos). | Sim | Classe GeometricVector. | |
| **Persistência de Objetos (via arquivo de texto ou binário)** | | | |
|  | - Persistência de Objetos. | Sim | Pacotes Mementos e Levels. | |
| - Persistência de Relacionamento de Objetos. | Sim | Idem item anterior. | |
| 5 | **Virtualidade:** |  |  | |
| - Métodos Virtuais. | Sim | Classe Tile, por exemplo. | |
| - Polimorfismo | Sim | Chamadas à draw, update e initializeGeneric de Entity na classe EntityList. | |
| - Métodos Virtuais Puros / Classes Abstratas | Sim | Classe Entity, por exemplo. | |
| - Coesão e Desacoplamento | Sim | No projeto como um todo. | |
| 6 | **Organizadores e Estáticos** | | | |
| - Espaço de Nomes (*Namespace*) criada pelos autores. | Sim | No projeto como um todo. | |
| - Classes aninhadas (*Nested*) criada pelos autores. | Sim | Classes List, TileMap e EventManager. | |
| - Atributos estáticos e métodos estáticos. | Sim | Classe RandomValueGenerator. | |
| - Uso extensivo de constante (*const*) parâmetro, retorno, método... | Sim | No projeto como um todo. | |
| 7 | Standard Template Library *(STL)* e String OO | | | |
| - 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 | Classes Entity e TileVector, por exemplo. | |
| - Pilha, Fila, Bifila, Fila de Prioridade, Conjunto, Multi-Conjunto, Mapa **OU** Multi-Mapa. | Sim | Classes Level (Conjunto), LeaderBoard (Multi-mapa) e EventManager (Mapas). | |
| **Programação concorrente** | | | |
| - Threads (Linhas de Execução) no âmbito da Orientação a Objetos, utilizando Posix, C-Run-Time **OU** Win32API ou afins. | Sim | Classe Thread, classe ThreadedCaster. | |
| - Threads (Linhas de Execução) no âmbito da Orientação a Objetos com uso de Mutex, Semáforos, **OU** Troca de mensagens. | Sim | Idem. | |
| 8 | **Biblioteca Gráfica / Visual** | | | |
| - Funcionalidades Elementares. &  - Funcionalidades Avançadas como:   * tratamento de colisões * duplo buffer | Sim | Carregar e desenhar imagens na tela, escrever texto e duplo buffer através da biblioteca SFML. | |
| - Programação orientada e evento em algum ambiente gráfico.  **OU**  - RAD – Rapid Application Development (Objetos gráficos como formulários, botões etc). | Sim | Classe EventManager e uso das funcionalidades de eventos da SFML. | |
| **Interdisciplinaridades por meio da utilização de Conceitos de Matemática e/ou Física.** | | | |
| - Ensino Médio. | Sim | Velocidade e aceleração, coordenadas cartesianas. | |
| - Ensino Superior. | Sim | Soma, subtração e multiplicação por escalar com vetores, versores. | |
| 9 | **Engenharia de Software** | | | |
| - Compreensão, melhoria e rastreabilidade de cumprimento de requisitos. | Sim | Reuniões com o professor e monitor. | |
| - Diagrama de classes em UML. | Sim | No desenvolvimento como um todo. | |
| - Uso efetivo (quiçá) intensivo de padrões de projeto (particularmente GOF). | Sim | Classe RandomValueGenerator (Singleton), pacote Mementos (Memento). | |
| - Testes a luz da Tabela de Requisitos e do Diagrama de Classes. | Sim | Reuniões com o professor e monitor, testes. | |
| 10 | **Execução de Projeto** | | | |
| - Controle de versão de modelos e códigos automatizado (via SVN e/ou afins) **OU** manual (via cópias manuais). &  - Uso de alguma forma de cópia de segurança (backup). | Sim | Repositório no Github. | |
| - Reuniões com o professor para acompanhamento do andamento do projeto. | Sim | 4: 29/10, 12/11, 19/11, 21/11 | |
| - Reuniões com monitor da disciplina para acompanhamento do andamento do projeto. | Sim | 9[[3]](#footnote-4): 21/10, 23/10, 25/10, 06/11, 8/11, 11/11, 13/11, 18/11, 22/11 | |
| - Revisão do trabalho escrito de outra equipe e vice-versa. | Sim | Revisão do trabalho da dupla do Franco Barpp Gomes e João Vítor Dotto Rissardi. | |

Tabela 3. Lista de justificativas para conceitos utilizados.

|  |  |  |
| --- | --- | --- |
| No. | Conceitos | Situação |
| 1 | **Elementares** | Classes, objetos, atributos, métodos e classe Principal foram utilizadas por serem partes integrais do POO. |
| 2 | **Relações** | Relações de associação, agregação e herança foram utilizadas por serem partes integrais do POO. |
| 3 | **Ponteiros, generalizações e exceções** | O operador this foi utilizado por possibilitar a passagem de ponteiros para objetos de classes associadas. O uso de memória dinâmica permitiu a variação do número de objetos existentes no programa em tempo de execução, Templates foram utilizados por aumentar a reusabilidade do código e blocos try/catch foram usados por facilitar o tratamento de erros difíceis de prever. |
| 4 | **Sobrecargas e persistências** | Sobrecargas de construtoras, métodos e operadores foram utilizados para aumentar a reusabilidade e intuitividade do código. |
| 5 | **Virtualidade** | Funções virtuais, classes abstratas e polimorfismo foram utilizados por serem partes integrais do POO, assim como a coesão e desacoplamento são indispensáveis na engenharia de software. |
| 6 | **Organizadores e estáticos** | Namespaces foram utilizados por ajudarem a identificar a qual parte do programa cada classe pertence, e para diferenciar classes com nomes similares (como List, que tem implementação dos discentes, da STL e da SFML).  O modificador const foi utilizado por aumentar a eficiência e clareza do código. Atributos e métodos estáticos foram utilizados para valores que deviam ser divididos entre todas as instâncias da classe e para métodos que deveriam poder ser chamados independentemente das instâncias da classe. |
| 7 | **STL, String e Programação Concorrência** | Os gabaritos e classes da STL foram utilizados por serem implementações prontas e eficientes de tipos abstratos de dados, acelerando consideravelmente o desenvolvimento. Programação concorrente foi utilizada para fins de aprendizado e pela conformação com a tabela de conceitos. |
| 8 | **Biblioteca Gráfica e Interdisciplinaridades** | A biblioteca gráfica SFML foi utilizada para garantir a compatibilidade entre sistemas operacionais e facilidade de implementação dos aspectos gráficos do programa. Conceitos de física e matemática foram utilizados para concretizar os modelos desejados de comportamento, como gravidade e direção de vetores. Programação orientada a eventos pela UML foi usada para tratar as interações do programa com o mouse e teclado. |
| 9 | **Engenharia de Software** | Requisitos, diagramas e testes foram contemplados por serem intrínsecos ao ciclo clássico de engenharia de software. O uso de padrões de projeto foi feito para melhorar a organização, coesão e desacoplamento do código, além de deixar as classes mais padronizadas. |
| 10 | **Execução de Projetos** | Controle de versão foi utilizado para garantir a organização e manter o histórico de toda versão do projeto. As reuniões com monitor/professor e revisão foram feitas para garantir a conformação aos requisitos e a qualidade do trabalho. |

# REFLEXÃO COMPARATIVA ENTRE DESENVOLVIMENTOS

Em primeiro lugar, é necessário lembrar que o paradigma orientado a objeto não é melhor que o procedimental. Ambos são, como o nome indica, paradigmas, ou seja, modelos, padrões que estruturam todo o processo de análise e implementação de um projeto de software, cada um com suas forças, fraquezas e aplicações respectivas.

Na percepção dos discentes, essas diferenças podem ser sintetizadas no fato que, no paradigma procedimental, pensa-se sempre no procedimento, na sequência de passos que precisam ser executadas para alcançar algum objetivo. Já no orientado a objetos, pensa-se em quais relacionamentos entre objetos que refletem as partes do problema a resolver precisam ocorrer. Essa possibilidade de abstração mais refinada facilita bastante o processo de análise e a reutilização de código, pois é mais fácil um objeto reaparecer em outros problemas do que todo um procedimento. É claro que existe um porém: essa facilidade aumenta o custo de memória e processamento para rodar o programa, o que, em casos onde a eficiência é crítica, pode impossibilitar o uso desse paradigma.

Um atributo do paradigma orientado a objetos que não possui equivalente dentro do procedimental é o polimorfismo. A possibilidade de chamar um método e executar uma de suas especializações abre portas para comportamentos e padrões de software extremamente complexos e refinados com esforço relativamente pequeno.

# DISCUSSÃO E CONCLUSÕES

O principal resultado obtido pelo processo de desenvolvimento é a percepção da importância de planejar um projeto antes de desenvolvê-lo. Os alunos, que já haviam desenvolvido outros projetos de software de complexidade menor do que o tema desse documento, demoraram muito mais tempo para fazê-lo, pois os executaram sem um planejamento prévio, criando um código que precisou ser mudado muitas vezes. Além disso, o estudo aprofundado do paradigma orientado a objetos possibilitou entendê-lo em todas as suas facetas e minúcias de uma forma que o estudo autônomo do assunto nunca provido. Isso posto, o aprendizado e a experiência adquirida pelos alunos durante a disciplina foram inestimável para sua formação como programadores e engenheiros.

# DIVISÃO DO TRABALHO

Tabela 4. Lista de Atividades e Responsáveis.

|  |  |
| --- | --- |
| Atividades. | Responsáveis |
| Levantamento de requisitos | Lucas e Thiago |
| Diagramas de classes | Lucas e Thiago |
| Programação em C++ | Lucas e Thiago |
| Implementação de *Template* | Lucas (Vector) e Thiago (List) |
| Implementação da persistência dos objetos | Thiago |
| Mecânicas de jogo | Mais Lucas que Thiago |
| Design das fases | Lucas |
| Arte | Lucas |
| Tratamento de colisões | Thiago |
| Tratamento de eventos | Thiago |
| Gerenciador gráfico | Thiago |
| Sistema de mapa de blocos (Tile Map) | Lucas |
| Inimigos, jogadores e item | Lucas e Thiago |
| Menus | Thiago |
| Uso de programação concorrente | Thiago |
| Classificação de jogadores | Lucas |
| Escrita do trabalho | Mais Lucas que Thiago |
| Revisão do trabalho | Lucas e Thiago[[4]](#footnote-5) |
| Preparação da apresentação | Mais Thiago que Lucas |

# AGRADECIMENTOS

Agradecimentos a Franco Barpp Gomes, João Vítor Dotto Rissardi (colegas discentes), Luciana Helena Bonancio (tradutora e revisora), revisores do trabalho e Felipe Alves (monitor da disciplina) pela assistência e supervisão prestadas durante o desenvolvimento.

# REFERÊNCIAS CITADAS NO TEXTO

[1] SIMÃO, J. M. Site das Disciplina de Técnicas de Programação, Curitiba – PR, Brasil, Acessado em 22/11/2019, às 08:46:

<http://www.dainf.ct.utfpr.edu.br/~jeansimao/Fundamentos2/Fundamentos2.htm>.

# REFERÊNCIAS UTILIZADAS NO DESENVOLVIMENTO

[A] SIMÃO, J. M. Site das Disciplina de Técnicas de Programação, Curitiba – PR, Brasil, Acessado em 22/11/2019, às 08:46:

<http://www.dainf.ct.utfpr.edu.br/~jeansimao/Fundamentos2/Fundamentos2.htm>.

[B] TIME C++ REFERENCE. Referência da linguagem de programação C++, Berkeley -California, EUA, Acessado em 22/11/2019, às 08:52:

<https://en.cppreference.com/w/>

[C] THE C++ RESOURCES NETWORK. Referência da linguagem de programação C++, Pasadena – California, EUA, Acessado em 24/11/2019, às 20:10:

<http://www.cplusplus.com/>

1. Devido a implementação dos obstáculos como parte do mapa de tiles, não existem várias instâncias de cada obstáculo, embora existam várias representações deles no jogo. [↑](#footnote-ref-2)
2. Devido a seu tamanho grande, é impossível colocar o diagrama UML em tamanho legível dentro desse documento. [↑](#footnote-ref-3)
3. Embora o requisito especificasse um mínimo de 10 reuniões, com apenas nove os requisitos do trabalho já puderam ser atingidos. [↑](#footnote-ref-4)
4. Outros revisores contemplados na seção de agradecimentos. [↑](#footnote-ref-5)