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*, particularmente de programação orientada a objetos em C++. Para tal, neste trabalho, escolheu-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 textualmente propostos e elaborada modelagem (análise e projeto) usando como recurso o Diagrama de Classes em Linguagem de Modelagem Unificada (*Unified Modeling Language* – *UML*), tendo como base um diagrama 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 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 elaborado. Por fim, salienta-se que o desenvolvimento em questão permitiu cumprir o objetivo de aprendizado visado.

**Palavras-chave ou Expressões-chave** (máximo quatro, não excedendo três linhas)**:** Paradigma Orientado à Objetos; C++; Desenvolvimento de jogos digitais.

**Abstract** - *This document shows a model for the manuscript to the academic work of Técnicas de Programação as well as it presents general instructions about this academic work. With respect to the abstract contents, it must provide a general explanation about the work. Precisely, the abstract must shortly present the work motivation and context, its study object (a platform game), its development process, and the obtained results. An instance of abstract would be*:

**. . .**

**Key-words or Key-expressions** (maximum four, not exceeding three lines)**:** Paper Model to the Academic Work of Programming Course, Academic Work Related to C++ Implementation, Internal Rules for Work Elaboration, Examples of Elements for the Work of a Programming Course.

# 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 à 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 esse objetivo, foi desenvolvido um jogo de plataforma com tema e mecânicas escolhidas pelos discentes e aprovadas pelo doscente, visando seguir os requisitos previamente determinados, que serão descritos a fundo no corpo do presente documento.

Para fazer isso, os alunos buscaram seguir o ciclo clássico de engenharia de software, ou

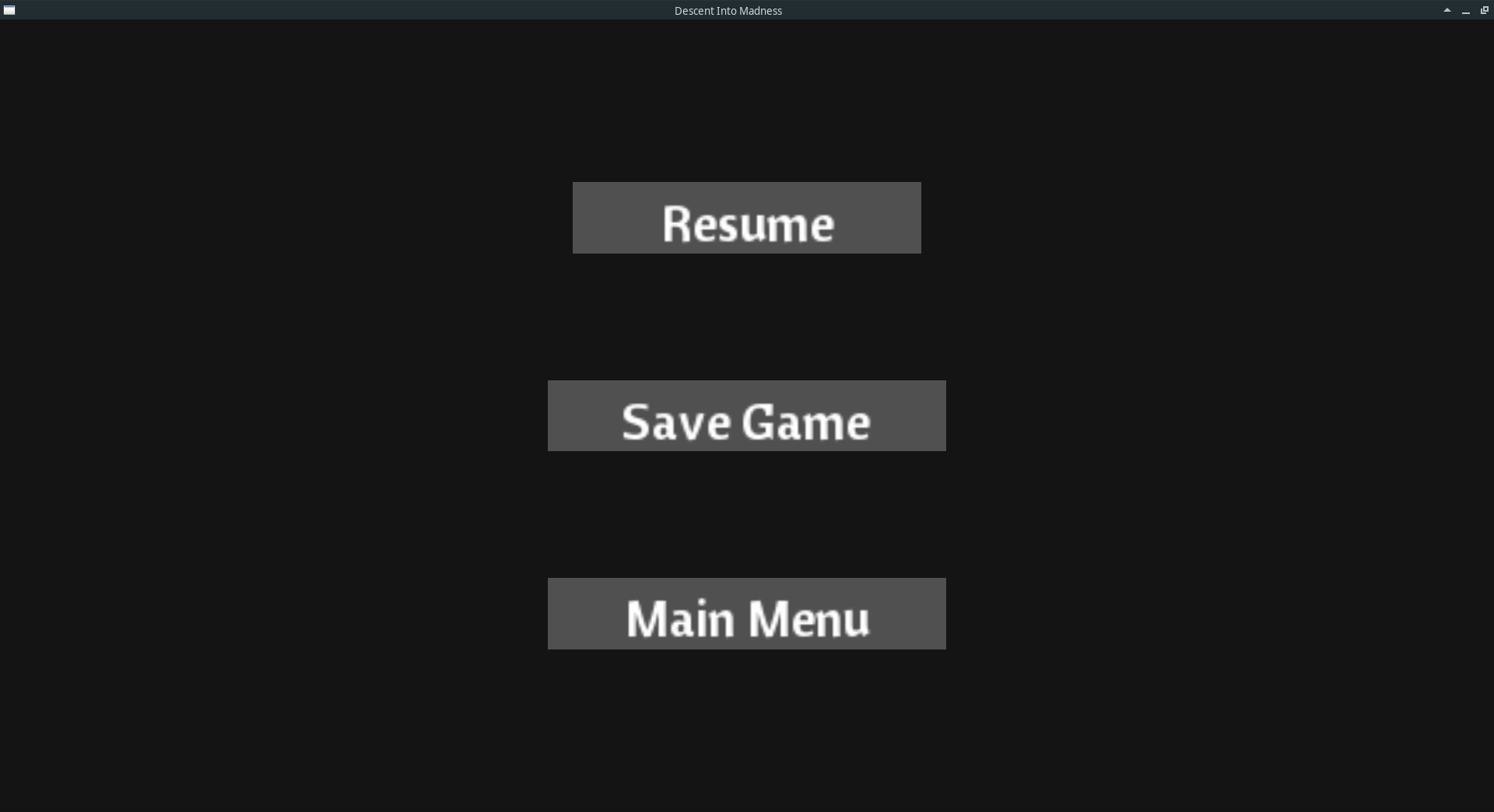
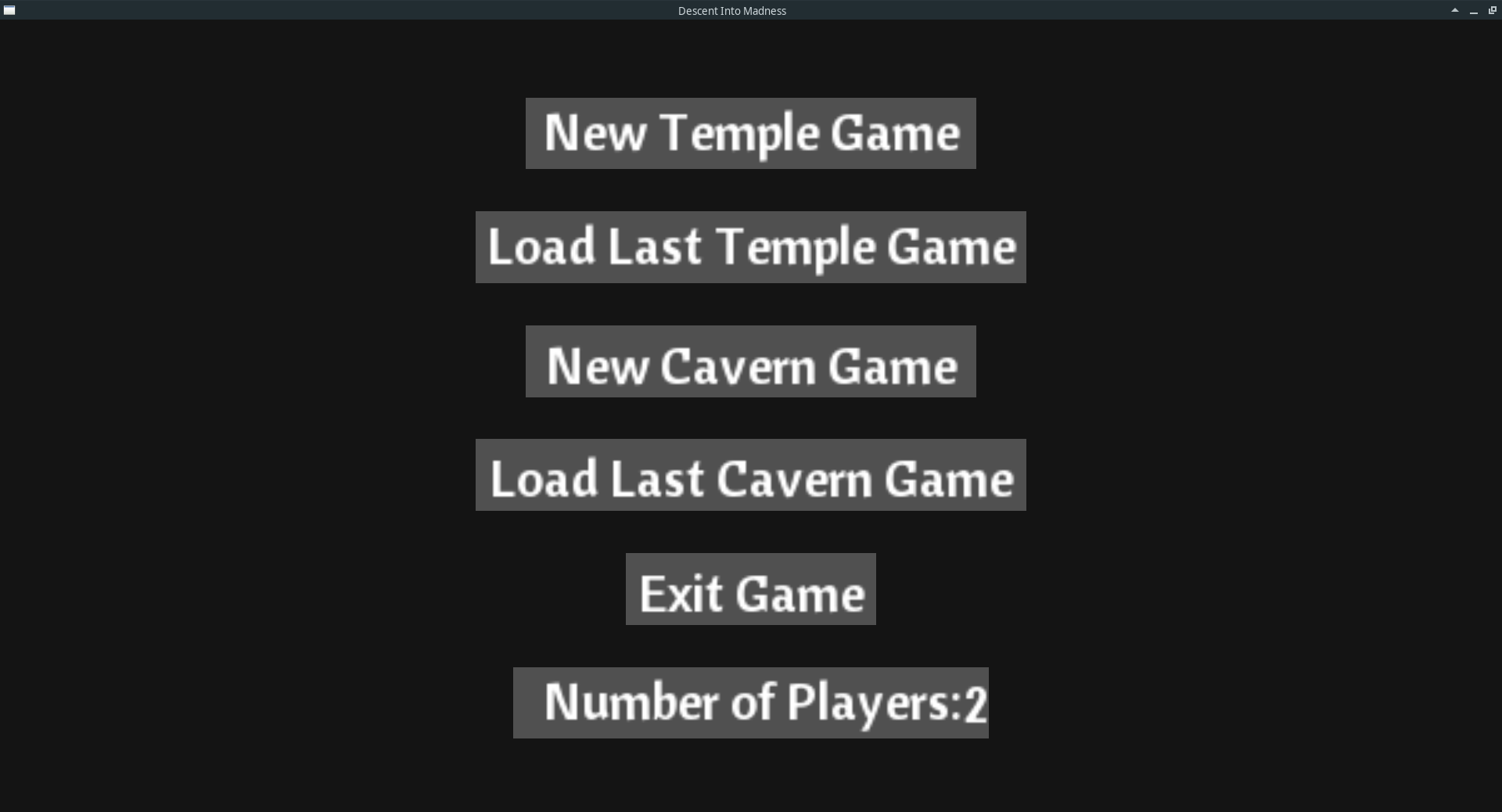
seja, com a orientação do professor e do monitor da disciplina, os discentes estudaram os requisitos à atingir, modelaram o software através de um diagrama de classes UML, implementaram o programa modelado através da linguagem C++ (com uso de algumas bibliotecas externas), e testaram o programa tão extensivamente quanto possível.

As seguintes seções desse trabalho tratarão de explicar o jogo em si, sua implementação, os conceitos (do POO, da disciplina ou até de outras áreas) utilizados, contendo finalmente considerações pessoais dos discentes, agradecimentos e referências.

# EXPLICAÇÃO DO JOGO EM SI

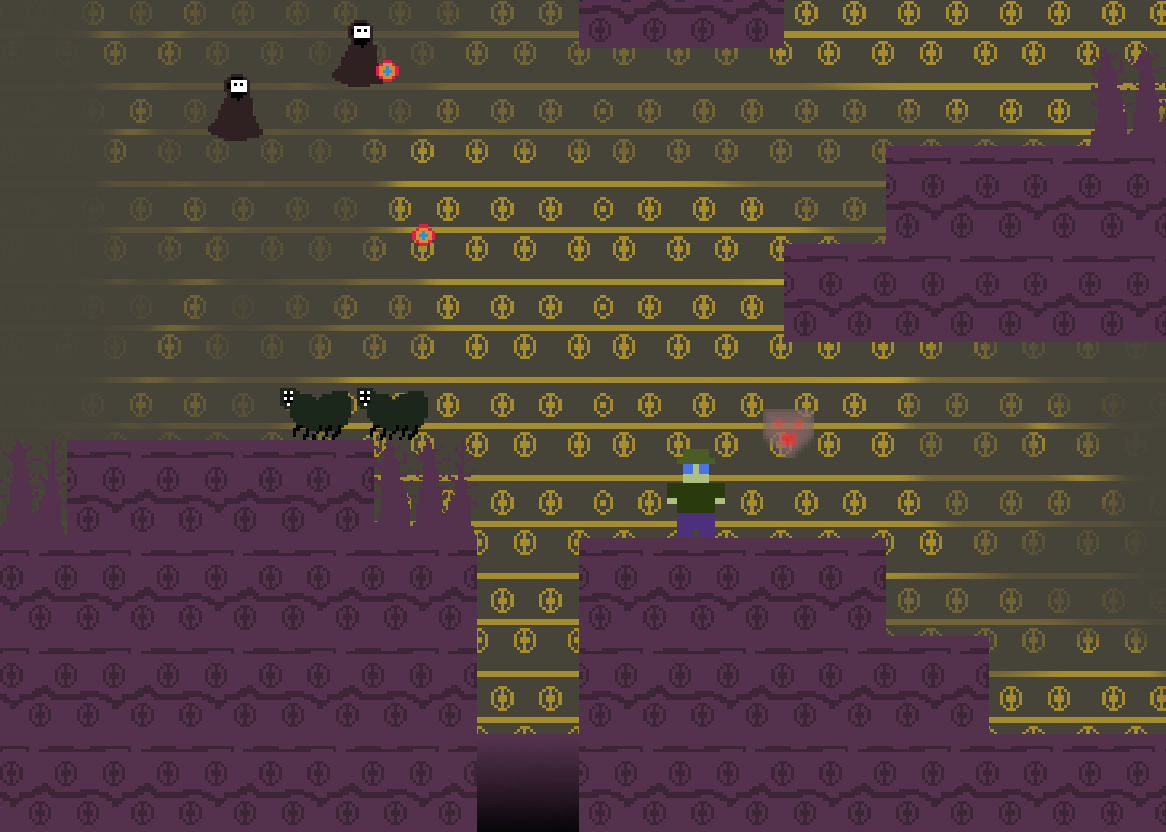
O jogo Descent Into Madness é um jogo de plataforma em terceira pessoa que permite um ou dois jogadores. Quando o programa é executado, o usuário se depara com um menu principal, onde pode escolher jogar a primeira ou a segunda fase do início ou a partir de um ponto salvo em outra sessão, escolher o número de jogadores ou sair do jogo. O outro menu existente no jogo é o menu de pause, que oferece as opções de voltar a jogar, sair do jogo ou salvar a fase, e pode ser acessado apertando a tecla ESC dentro de uma fase.

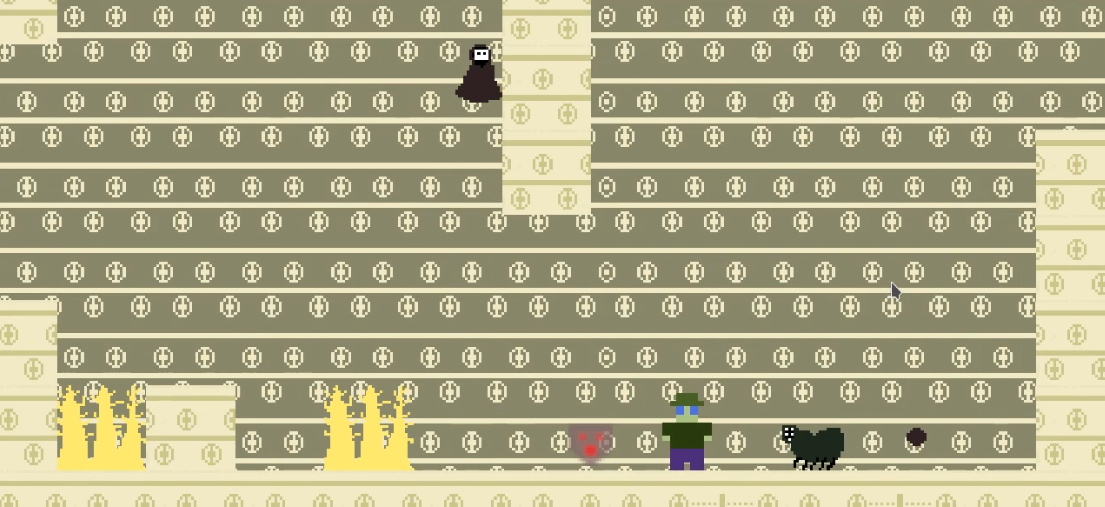
<<Colocar menus com mais botão>>



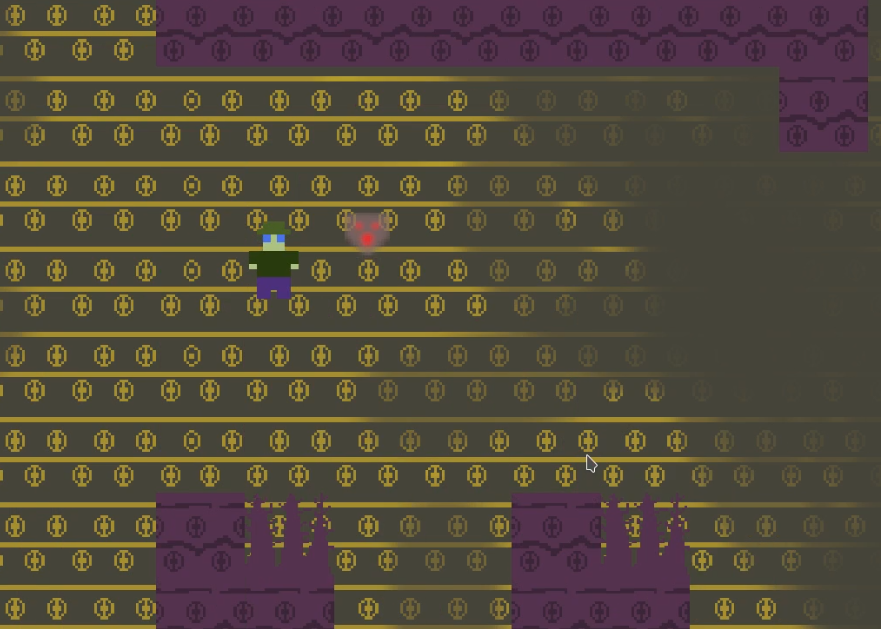
Figuras 1 e 2 : Menu Principal e Menu de Pausa, respectivamente.

O objetivo do jogo é chegar ao final da fase, marcada pelos tiles em degradê na horizontal. O jogador deve fazer isso sem tocar nenhum inimigo, obstáculo ou projétil, que significam morte instantânea e retorno ao início da fase. O jogador que conseguir terminar o jogo com menos mortes terá uma classificação melhor na classificação de melhores jogadores.

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

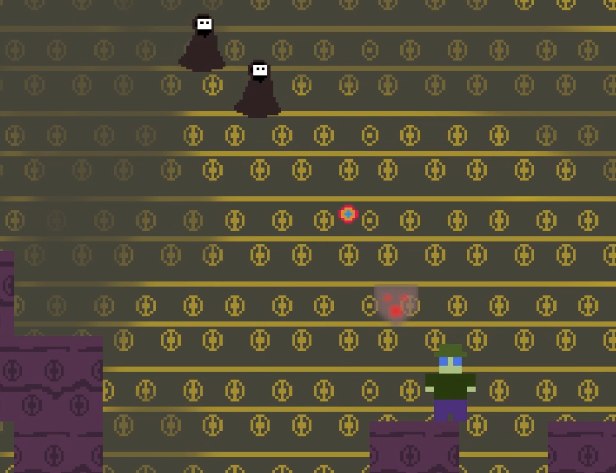
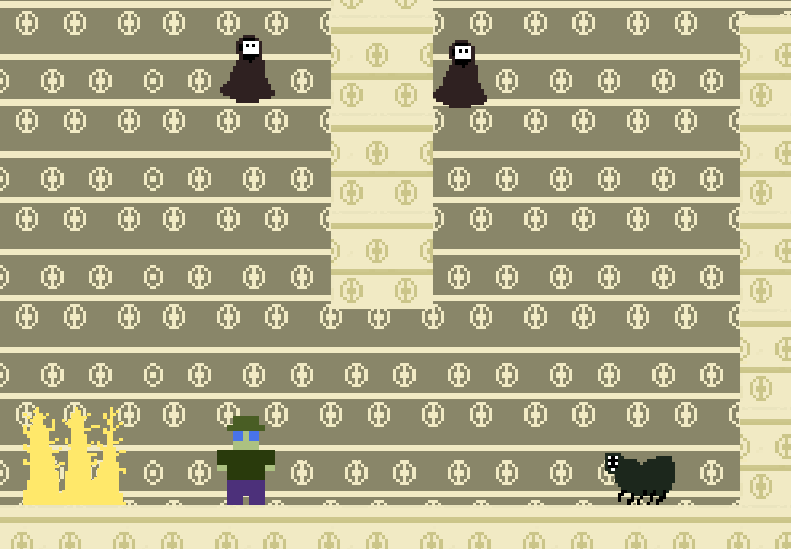


Figuras 3 e 4 : Seções da fase 1 (templo) e 2 (caverna), respectivamente, com seus inimigos e obstáculos.



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

O primeiro jogador (The Undying) é controlado com as teclas A e D, que o movimentam para a esquerda e direita, e o botão de espaço, que o faz pular. O segundo jogador (The Penitent), que tem a função de proteger o primeiro dos projéteis atirados, é controlado com a roda do mouse, e está sempre em volta do jogador 1.



Figuras 7 e 8: Jogador 1 sozinho e 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, aparece na tela um diálogo onde o jogador pode escrever seu nome, que é salvo junto com sua pontuação na classificação mostrada em seguida.

<<Colocar fotos>>

Figuras 12 e 13 : Diálogo para escrita do nome do jogador e 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. | Requisito 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. | Requisito cumprido via pacote Players e objeto da classe MainMenu. |
| 3 | Disponibilizar ao menos duas fases que podem ser jogadas sequencialmente ou selecionadas. | Requisito previsto inicialmente e realizado. | Requisito 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. | Requisito 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 | Requisito cumprido à partir da classe TheChained e seu objeto. |
| 7 | Ter três tipos de obstáculos. | Requisito previsto inicialmente e realizado. | Requisito 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. | Requisito cumprido através dos objetos da classe HoleObstacle, BulletObstacle e das derivações de SpikeObstacle, além de objetos da classe TileManager[[1]](#footnote-2) |
| 9 | Ter representação gráfica de cada instância. | Requisito previsto inicialmente e realizado. | Requisito cumprido através do objeto da classe GraphicsManager. |
| 10 | Ter em cada fase um cenário de jogo com os obstáculos. | Requisito previsto inicialmente e realizado. | Requisito 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 projeteis (em havendo). | Requisito previsto inicialmente e realizado. | Requisito 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). Salvar a pontuação e gerar ranking ainda não implementado. |
| 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. |

<<Colocar diagrama final>>

Figura 14: Diagrama de Classes UML.

A classe principal do jogo é DescentIntoMadness, que recebe seu nome. O objeto dessa classe agrega as fases, menus e jogadores(que serão explicados subsequentemente), 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, e é a única classe incluída diretamente no pacote (namespace) principal DescentIntoMadness. 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.

<<Que ordem faz mais sentido: principal, managers, Levels, Entities ou o contrário ( o resto acho que é depois mesmo, por ser mais acessório)>>

<<Imagem>>

Figura 15: Diagrama de classes UML – Foco no pacote Managers.

GraphicsManager é a classe responsável por todos os elementos gráficos do jogo, interagindo diretamente com a biblioteca gráfica utilizada SFML. Para desenhar alguma coisa na tela, um objeto com acesso à essa classe pode pedir em sua inicialização que se carregue uma imagem (método loadAsset), e depois simplesmente pedir que ela seja desenhada quando quiser, usando o próprio caminho para a imagem na chamada de método (método draw), pois todas os objetos sf::Texture são armazenados em um mapa (atributo assets) onde a chave é o próprio caminho para a imagem. Essa classe também é capaz de escrever na tela (método drawText) e desenhar retângulos (método drawRect).

A classe EventManager é responsável por lidar com todos os eventos relacionados ao mouse e teclado. 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 utilizar outras bibliotecas com funcionalidades similares.Todas os objetos que precisam responder a movimentos ou cliques do mouse e teclado simplesmente chamam o método relevante (addKeyboardListener ou addMouseListener) passando como parâmetro uma função sem parâmetro de retorno void, e se o evento acontecer, essa função será chamada.

A classe CollisionManager gerencia as colisões das entidades físicas (objetos de classes que extendem PhysicalEntity, que serão explicadas em seguida) entre si e com o mapa. Todas as entidades físicas são adicionadas às colisões durante sua inicialização, e quando o método checkCollisions é chamado, ela verifica quais entidades estão colidindo com as outras comparando suas posições e tamanhos, e então verifica se estão colidindo com o mapa chamando o método checkCollisions da classe TileManager (que será explicada em seguida). Se houver colisões, o método collided da PhysicalEntity é chamado, parametrizado com uma string que identifica com o que houve colisão, o tamanho do outro objeto e sua posição.

<<colocar foto>>

Figura 15: Diagrama de classes UML – Foco no pacote Entities

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 sistema de tiles, que, embora extenda a classe Entity, é mais complexa, e por isso merece um pacote próprio e atenção especial posterior. Esse pacote é composto da classe abstrata Entity, que possui métodos virtuais puros draw, update e initializeSpecific. Em geral, a inicialização de uma entidade depende de chamadas à métodos de classes associadas a e acessíveis a partir do nível atual. Para garantir que o ponteiro que cada entidade tem para a fase seja inicializado, existe o método inicializeGeneric, que simplesmente inicializa o ponteiro para Level currentLevel e chama a função initializeSpecific.

A classe PhysicalEntity, também abstrata, é extendida 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. TileManager não extende essa classe pois mesmo sendo “uma entidade que precisa colidir”, o tratamento de suas colisões é muito mais complexo, precisando de atenção especial.

<<ISSO FAZ SENTIDO????>

Das classes derivadas de PhysicalEntity, ainda existem a classe Projectile, que é implementada por entidades que se movem retiliniamente com velocidade constante e que podem ser destruídas ao colidirem; e Mob, que é implementada por entidades que tem comportamento mais complexo, nomeadamente, jogadores e inimigos. O item *The Mirror of Hastur*, por colidir sem ter velocidade própria (ou parado ou acompanhando o player), extende PhysicalEntity diretamente.

<<colocar foto>>

Figura 16: Diagrama de classes UML – Foco no pacote Levels

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 abordada posteriormente.

<<colocar foto>>

Figura 17: Diagrama de classes UML – Foco no pacote Menu

Os menus mantém desenhadas uma lista de botões, e reagem apropriadamente quando algum deles é clicado, através do EventManager e do GraphicsManager.

<<colocar foto>>

Figura 18: Diagrama de classes UML – Foco no pacote Utils

O pacote utils é composto de classes criadas pouco relacionadas entre si, mas que tem a característica comum de serem utilizadas para apoiar as outras classes do projeto, como o singleton RandomValueGenerator, que gera booleanos, inteiros ou floats aleatórios com uma chamada de função; GeometricVector, gabarito que agrupa dois valores do mesmo tipo como coordenadas x e y, utilizado principalmente para posições, velocidades e coordenadas em matrizes; e List, gabarito para uma lista duplamente encadeada com iterador.

<<colocar foto>>

Figura 18: Diagrama de classes UML – foco no pacote TileSystem

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 carrega um arquivo de texto contendo números que representam ou espaços vazios (números negativos) ou índices de tiles (números não negativos) e o transforma em uma matriz alocada dinamicamente de short ints; e um TileVector, que armazena vários Tiles, classes que se comportam de forma similar a PhysicalEntity, mas não tem uma posição própria, tendo que receber onde se desenhar e sua posição quando há uma colisão. Para desenhar o mapa de Tiles, o TileManager percorre o mapa, chamando a função desenhar do tile armanzenado 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. Como é possível colidir com mais de um tile ao mesmo tempo, essa função retorna um vetor de triplas ordenadas com o código identificador do tipo de tile, sua posição e seu tamanho, com cada uma das triplas sendo passadas para o método colided de cada vez. Há apenas uma classe que estende Tile pois BulletObstacle é o único Tile que tem um comportamento diferente do padrão, todas os outros tiles podendo ser feitos simplesmente variando os parâmetros da construtora de Tile.

<<colocar foto>>

Figura 19: Diagrama de classes UML – foco no pacote Mementos.

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 Memento (nas entidades, createMemento) e um para restaurar o estado salvo (nas entidades, loadMemento). O salvamento e posterior carregamento são feitos pelos objetos de classes que estendem Level, através da classe LevelMemento. Os arquivos de salvamento são compostos de uma linha para cada memento salvo, para facilitar o carregamento. Para as entidades, o primeiro identificador é o próprio ID da classe, seguido da posição, velocidade, e outros atributos necessários.

# TABELA DE CONCEITOS UTILIZADOS E NÃO UTILIZADOS

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Uso | Onde / O quê | |  | |
| **Elementares:** | | | | |
| - Classes, objetos. &  - Atributos (privados), variáveis e constantes. &  - Métodos (com e sem retorno). | | Sim | | Todos .h e .cpp |
| - Métodos (com retorno *const* e parâmetro *const*). &  - Construtores (sem/com parâmetros) e destrutores | | Sim | | Todos .h e .cpp |
| - Classe Principal. | | Sim | | Main.cpp & Principal.h/.cpp |
| - Divisão em .h e .cpp. | | Sim | | No desenvolvimento como um todo. |
| **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 multíplos níveis entre Entity e Mob. |
| - Herança múltipla. | | Sim | | <<ONDE>> |
| **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 GraphicsManager |
| **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. |
| **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. |
| **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. |
| 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), 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. |
| **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; |
| **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. |
| **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[[2]](#footnote-3): 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 e **Não** Utilizados no Trabalho.

|  |  |  |
| --- | --- | --- |
| No. | Conceitos | *Listar apenas os utilizados* 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 foi feito por permitir 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 dificeís 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 módulo do programa cada classe pertence, e para diferenciar classes com nomes similares a implementadas por outros (como List, que tem implementação própria, na STL e na 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 da existência de instâncias da classe. |
| 7 | **STL, String e Programação Concorrência** | Os gabaritos e classes da STL foram utilizados por serem implementações prontas e eficiências de inúmeros tipos abstratos de dados, acelerando consideravelmente o desenvolvimento. Programação concorrente foi utilizada para fins de aprendizado e para haver 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. RAD foi utilizada para facilitar o desenvolvimento da interface de classificação de pontuação rapidamente. |
| 9 | **Engenharia de Software** | Requisitos, diagramas e testes foram feitos 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 registrar as responsabilidades de cada membro da equipe, além de manter o histórico de toda versão feita 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 seu 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, se pensa sempre no procedimento, na sequência de passos, que precisam ser executadas para algum objetivo ser atendido. No orientado à objetos, é possível pensar em que relacionamentos entre objetos que refletem as partes do problema à resolver precisam se comunicar, e como. Essa possibilidade de abstração mais refinada facilita bastante o processo de análise do problema em questão, e facilita muito a reutilização de código, pois é mais fácil um objeto reaparecer em vários 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 obtida do 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 desenvolveram sem pensar no projeto como um todo, o que leva a muito código ter de ser refeito muitas vezes. Posto isso, os alunos também sentiram a importância de se ater atentamente aos requisitos, pois perderam muito tempo tentando implementar coisas que não atendiam a nenhum deles, o que poderia ter comprometido a execução completa do trabalho se os discentes não tivessem percebido esse erro a tempo.

# CONSIDERAÇÕES PESSOAIS

Os discentes têm a opinião de que 12 páginas foi muito pouco para a confecção do presente documento, visto os requisitos que ele deve cumprir.

# 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 [[3]](#footnote-4) |
| Preparação da apresentação | Mais Thiago que Lucas |

# AGRADECIMENTOS

Agradecimentos a Franco Barpp Gomes, João Vítor Dotto Rissardi (colegas discentes) e 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, Acessado em 22/11/2019, às 08:52:

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

1. Devido a implementação dos obstáculos como parte do mapa de tiles, não existem várias instâncias de cada obstáculo, embora existam sim várias representações deles no jogo. [↑](#footnote-ref-2)
2. Embora o requisito especificasse um mínimo de 10 reuniões, foi percebido que, com apenas 9, os requisitos do trabalho já puderam ser atingidos. [↑](#footnote-ref-3)
3. Outros revisores contemplados na seção de agradecimentos. [↑](#footnote-ref-4)