TRABALHO PARA A DISCIPLINA DE   
TÉCNICAS DE PROGRAMAÇÃO DO CURSO DE   
ENGENHARIA DE COMPUTAÇÃO DA UTFPR:   
*Scape from Cemitery*

Douglas Yamasaki Cunha, Segundo Autor

dcunha@alunos.utfpr.edu.br,pedrotangerino@hotmail.com

Disciplina: **Técnicas de Programação – CSE20** / S73 – Prof. Dr. Jean M. Simão

**Departamento Acadêmico de Informática – DAINF** - Campus de Curitiba

Curso Bacharelado em: Engenharia da Computação / Sistemas de Informação

### Universidade Tecnológica Federal do Paraná - UTFPR

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

**Resumo** - Um dos requisitos para aprovação na disciplina de Técnicas de Programação é o desenvolvimento de um *software*, obrigatoriamente no formato de um jogo, para a prática de conceitos e técnicas aprendidos em aula, sobre engenharia de *software* e programação orientada a objetos. O nome escolhido para o jogo é *Scape from Cemitery*, e consiste em um jogo de plataforma, *side-scroller*, no qual o jogador enfrenta inimigos em dificuldade crescente. O desenvolvimento deste projeto deu-se com base nos requisitos propostos. Como etapa do desenvolvimento, foi utilizada como recurso a modelagem através de um Diagrama de Classes em Linguagem de Modelagem Unificada (em inglês, *UML: Unified Modeling Language*), a partir de um exemplo proposto. Com isto, foi realizada a implementação do jogo em si na linguagem de programação C++, fazendo uso de seus recursos de Orientação a Objeto, em particular, os conceitos desta modalidade de programação abordados em aula, como Classe, Objeto, e Relacionamento, além de conceitos avançados (principalmente associados ao conceito de Herança). como Classes Abstratas, Polimorfismo, Gabaritos (*templates*), Persistências de Objetos por Arquivos, Sobrecarga de Operadores, a Biblioteca Padrão de Gabaritos (em inglês, *STL: Standard Template Library*), alguns padrões de objetos, como por exemplo, *Composite*, uma derivação dos conceitos de *State* e *Singleton*. Finalmente, após a implementação, verificou-se a funcionalidade (bem como o atendimento dos requisitos) do *software* a partir de testes e do uso do mesmo. Ademais, é importante notar que o objetivo final, o aprendizado das técnicas e conceitos mencionados, foi cumprido.

**Palavras-chave ou Expressões-chave:** Relatório de Desenvolvimento do Projeto para a disciplina de Técnicas de Programação, Programação Orientada a Objetos, Implementação em C++.

# INTRODUÇÃO

O contexto deste trabalho é estabelecido pela disciplina de Técnicas de Programação, que aborda como conceito fundamental a Programação Orientada a Objetos. O trabalho é uma forma de aplicar os conceitos teóricos aprendidos em aula, e com o material didático, para efetivamente desenvolver um projeto, o qual deve atender a requisitos relacionados ao aprendizado.

O objeto desta aplicação e da implementação, o *software* em formato de jogo, será desenvolvido em etapas pela equipe. Isto envolverá o desenvolvimento de diversas versões do Diagrama de Classes, assim como várias seções da própria implementação computacional. Com isso, haverá também o estudo das técnicas a serem utilizadas, conforme o objetivo do trabalho.

Quanto ao método de Engenharia de *Software*, foram observados os requisitos, para em seguida estabelecer como se daria a modelagem do *software* que os atendesse. Conforme mencionado no texto acima, isto se deu através do diagrama de classes em *UML*. Após estes dois estágios do ciclo é que houve efetivamente a implementação do jogo em C++ orientado a objetos, e o subsequente uso do software, para fins de testá-lo.

O jogo consiste no controle de um personagem arqueiro, controlável, com três tipos de ataques. O jogador enfrentará inimigos do tipo dragão, fantasmas, e um *warlock* (*boss*), tendo ainda que lidar com três tipos de obstáculos (como qualquer jogo plataforma).

# EXPLICAÇÃO DO JOGO EM SI

Na tela inicial do jogo (exibida logo após a execução), é exibido um dos menus, o menu principal. Cada menu é composto por botões clicáveis, que servirão tanto para passar para outros menus (com opções mais especializadas), quanto para executar outros tipos de funções específicas.

No menu principal temos os botões *quit game* (o qual fechará a janela do jogo), *load game* (função específica de carregar jogada), *highscore*, que irá levar até outro menu, para que se exiba o *ranking* (melhores 15 pontuações já obtidas, para os modos de dois jogadores ou de somente um).

De volta ao menu principal, temos ainda os botões *2 player* e *1 player*, os quais são responsáveis por ativar a presença do 2 jogador, ou jogar apenas como jogador solo, respectivamente. Após selecionar o botão *1 Player* ou *2 Player* será selecionado um menu para que se registre o dado dos jogadores (nome).

O *input* é realizado com a simples digitação dos nomes dos jogadores (*players*) e apertando a tecla “enter” para confirmar o nome digitado. Após a confirmação do nome de ambos os jogadores (caso o jogador 2 esteja ativo), o jogo seguirá para a próxima tela, que consiste no menu de seleção de fases. Aqui, se poderá escolher a primeira fase (e em seguida jogar a segunda, caso a primeira seja vencida), ou ir direto para a segunda fase (*boss stage*).

Após selecionada uma das fases, o jogo carregará.

Os controles do personagem são os seguintes:

* Jogador 1:

w, a, d (pular, esquerda, e direita, respectivamente), x , c , v (ataque 1, ataque 2, e ataque 3, respectivamente);

* Jogador 2:

i, j, l (pular, esquerda, e direita, respectivamente), b, n, m (ataque 1, ataque 2, e ataque 3, respectivamente).

Há também a possibilidade de se pausar o jogo (com a tecla “p”), transitando para a tela de *pause* que segue a seguir :

Esta tela contém três botões: o primeiro retorna ao jogo, o segundo cumpre a função de salvar a jogada, e o terceiro descarta a jogada atual e retorna ao menu principal.

O ataque 1 corresponde a um tiro único, com grande precisão e rapidez.

O ataque 2 corresponde a um tiro triplo, mas com precisão e velocidade medianas.

O ataque 3 corresponde a um tiro quíntuplo, atirado de forma semi vertical, sofrendo grandes perdas de precisão e velocidade.

O jogador conta com um HUD indicativo, contendo o seu nome, quantidade de pontos, e vidas representadas por frascos de poção. Quando vazios, representam ausência de pontos de vida, e quando cheios, o contrário.

O objetivo da primeira fase é simplesmente chegar ao final dela. Nesta fase , o jogador enfrentará inimigos fantasmas, que o perseguem jogador, e conseguem atravessar paredes (no caso de dois jogadores, eles irão perseguir algum dos jogadores, escolhido aleatoriamente).

Caso o jogador derrote um fantasma , outro nascerá logo em seguida.

Também será enfrentado um dragão esqueleto, cujo comportamento é relativamente mais estacionário, e que possui a habilidade de atirar bolas de fogo em direção ao jogador mais próximo.

Ainda deverá enfrentar obstáculos como espinhos e minas.

O jogador receberá dano ao encostar em algum destes objetos (cada um tomará 1 das vidas do jogador), com a mina explodindo e desaparecendo do seu lugar após ser ativada.

A progressão do nível é bastante linear (sem bifurcações), e ao chegar no final (passando de um certo ponto do mapa), o próximo nível será carregado e posto para ser jogado.

O segundo nível conta com a presença dos já citados fantasmas, o chefe final (não imediatamente presente), e também de minas espalhadas no objetivo central da fase.

Esta fase contém uma mecânica única (no contexto do jogo), onde o jogador deve destruir (basta andar por cima) todas as lápides que se encontram espalhadas pela fase para que o boss seja finalmente revelado.

O chefe possui 3 ataques: um rápido, um de média velocidade, e outro lento. Ele correrá de qualquer um dos jogadores, e após atingir certa distância, irá conjurar acima da posição de ambos os jogadores suas magias.

Para derrotá-lo, é necessário desviar de suas magias, andando enquanto a animação de cada conjuração é feita, para não tomar dano letal, e atirar em suas brechas.

Todas as magias são animadas e contam com começo, pico e fim, e somente à partir do pico trarão alguma ameaça ao jogador.

Caso a vida de ambos os jogadores (ou somente um) seja reduzida a zero, o nível será dado automaticamente como perdido.

Caso o jogador derrote o chefe, aparecerá uma tela informando que o jogador venceu.

Em ambos os casos , a pontuação dos jogadores envolvidos naquela jogada será automaticamente salva ao *ranking*.

# 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 MenuHandler e suas respectivas classes auxiliares, formando uma derivação do Padrão State. |
| 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 inclusive via classe Player cujos objetos são agregados em Principal, e utilizados pela classe Level e suas derivações, a partir da escolha através do menu. |
| 3 | Disponibilizar ao menos duas fases que podem ser jogadas sequencialmente ou selecionadas. | Requisito previsto inicialmente e realizado. | Requisito foi realizado através da classe Level (abstrata) derivando-se em FirstStage e SecondStage (ambas as fases agregadas na principal, a qual utiliza apenas uma por polimorfismo). |
| 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. | No pacote Entidade Abstrata , há uma classe chamada “Enemie” , a qual é abstrata e serve como base para os inimigos presentes no pacote Entidade Concreta, Ghost, Dragon e Warlock. |
| 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 inclusive via criação e instancionamento em sua respectiva lista de inimigos ,única para cada fase. |
| 6 | Ter inimigo “Chefão” na última fase | Requisito previsto inicialmente e realizado. | Requisito cumprido via classe derivada de Enemie presente no pacote Entidade Concretas chamada de Warlock. |
| 7 | Ter três tipos de obstáculos. | Requisito previsto inicialmente e realizado. | Requisito cumprido inclusive via derivação da classe Obstacle (presente no pacote Entidade Abstrata). As classes derivadas são : Mine, Spike, e Tombstone. Todas elas presentes no pacote Entidade Concreta. |
| 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 inclusive via criação e instancionamento em sua respectiva lista de obstáculos, única para cada fase. |
| 9 | Ter representação gráfica de cada instância. | Requisito previsto inicialmente e realizado. | Requisito cumprido inclusive via namespace Models. Este namespace contém várias classes terminadas com a palavra “Models” ,que são Singletons individuais contendo sua(s) respectiva(s) texturas. Ex.: ArcherModel contém o Singleton do Jogador. |
| 10 | Ter em cada fase um cenário de jogo com os obstáculos. | Requisito previsto inicialmente e realizado. | Requisito cumprido inclusive via agregação dos objetos pertencentes a fase em suas respectivas listas. |
| 11 | Gerenciar colisões entre jogador e inimigos, bem como seus projéteis (em havendo). | Requisito previsto inicialmente e realizado. | Requisito cumprido inclusive via classe Body (presente no pacote Abstrato) e Collider (cada fase agrega um Objeto do tipo Collider e este efetua colisões com base em Body) |
| 12 | Gerenciar colisões entre jogador e obstáculos. | Requisito previsto inicialmente e realizado. | Requisito cumprido inclusive via classe Body (presente no pacote Abstrato) e Collider (cada fase agrega um Objeto do tipo Collider e este efetua colisões com base em Body) |
| 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:  Cadastrar - Classe MenuNaming.  Salvar dados - função savePoints em Level.  Gerar Ranking - Classe MenuRanking |
| 14 | Permitir Pausar o Jogo | Requisito previsto inicialmente e realizado | Requisito cumprido inclusive via alteração do loop principal de atualização do jogo para a atualização do menu pause (Formado pela classe MenuPause). |
| 15 | Permitir Salvar Jogada. | Requisito previsto inicialmente e realizado. | Requisito realizado pela classe LevelStateHandler. Cada classe derivada de Level agrega um objeto do tipo LevelStateHandler. |

A estrutura da classe principal é composta pela agregação de dois objetos derivados de Level. Um objeto representando o First Stage (primeiro nível) e o segundo representando o SecondStage (segundo nível). Esses dois objetos representam os níveis em si. Serão utilizadas funções internas para carregar suas agregações e composições formando assim as fases.

A principal também agrega dois objetos do tipo Player, representando os jogadores. Esses dois objetos são os dois jogadores propriamente ditos, e serão utilizados e moldados de acordo com as fases. Quando se passa um jogador para uma fase , ela automaticamente o molda em seus parâmetros.A função principal (executar contida na classe principal) , contém dois loops que se alternam , o loop primário do jogo e o loop do menu.

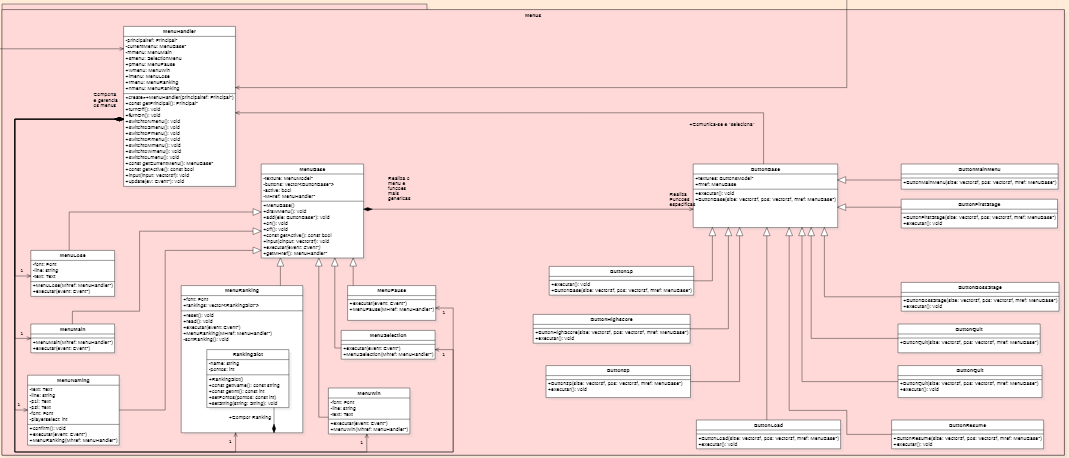


Figura 1. UML pasta “Menu”

O menu é realizado por uma agregação de um objeto do tipo MenuHandler , composto por uma referência a um menu (MenuBase , utiliza-se polimorfismo para executar cada função). E todos os menus que compõem o jogo , cada menu tem uma função específica e cada um é composto por um conjunto de operações que se resume em comunicar-se com a principal e alterar o estado do jogo.

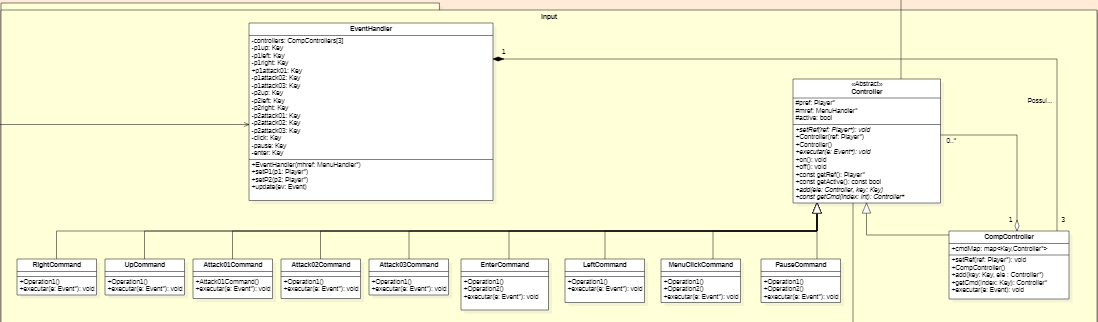


Figura 2. UML pasta “Input”

A principal também agrega um objeto do tipo EventHandler , o qual em cada atualização do seu loop processa os eventos através de um composite. Cada nó folha , representado à esquerda , contem um comando e interpretação unico , que através de ponteiros na sua classe advindos de sua classe abstrata base realizam a comunicação do input via evento.

Sua completa estruturacao se da em 3 niveis , sendo a primeira camada relacionada aos comandos do menu , a segunda aos comandos de movimento do jogador e a terceira os de ataque.

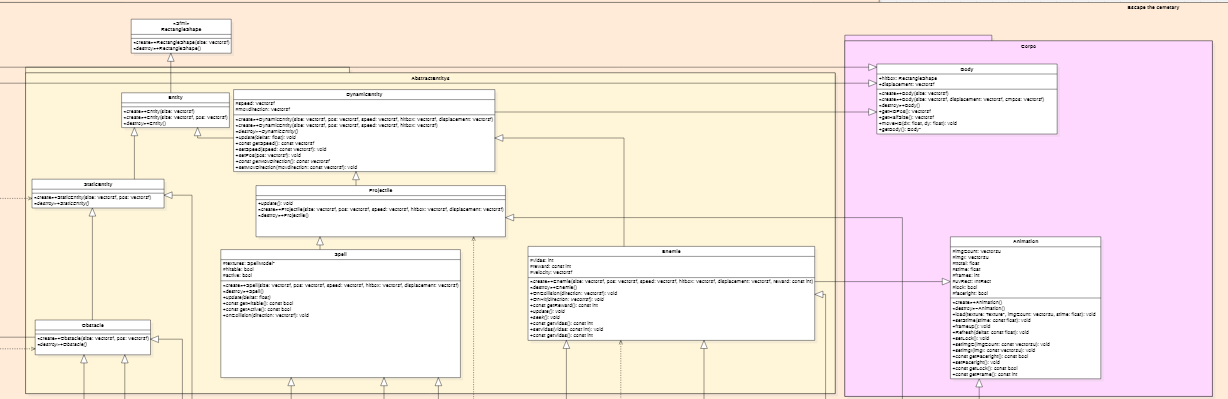


Figura 3. UML pasta “AbstractEntity” e “Corpo”

A estruturação das entidades presentes no jogo começa com a classe Entity , uma classe derivada de RectangleShape , esta classe vem do próprio SFML e contém várias funções auxiliares para desenho , texturização , etc.. (estabelece aqui uma premissa que todas as classes derivadas de Entity , serão objetos desenhados). Após isso , herda-se duas classes : DynamicEntity e StaticEntity , essas representam entidades com movimento e sem , respectivamente. A partir dessas , são formadas várias outras classes abstratas a fim de especialização das entidades e o seu agrupamento em listas. Há ainda a pasta corpo contendo as classes Animation e Body , a primeira é responsável por controlar as animações e recortes das Sprites , enquanto a segunda, armazena o corpo do objeto. Para o projeto é importante a separação da representação gráfica e do seu corpo visto que a maioria das texturas utilizadas via Sprite Sheet nao ha correspondencia certeira entre a imagem e o corpo a se utilizar nas colisões. Partindo dessas classes citadas , deriva-se as classes bases para os inimigos (Enemie) e para os obstáculos (Obstacle) e também para os projéteis (Projectile) e também a classe Spell , derivada de Projectile , o qual no futuro irá compor os ataques do chefe.

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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **N.** | | Conceitos | | Uso | | Onde / O quê | |
|  | **1** | | **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. | |
|  | **2** | | **Relações de:** | | | | | |
|  | - Associação direcional. &  - Associação bidirecional. | | Sim | | No desenvolvimento como um todo. | |
|  | - Agregação via associação. &  - Agregação propriamente dita. | |  | | No desenvolvimento como um todo. | |
|  | - Herança elementar. &  - Herança em diversos níveis. | | Sim | | Listas , Menus , Controllers , Stages, Entidades... | |
|  | - Herança múltipla. | | Sim | | DynamicEntity.h,Obstacle.h,Enemie.h,Player.h,Platform.h | |
|  | **3** | | **Ponteiros, generalizações e exceções** | | | | | |
|  | - Operador *this*. | | Sim | | No geral, para sets e também referenciamento próprios nos menus (namespaces Menus, compostos pelos .hs e .cpps iniciados com a palavra Menu) | |
|  | - Alocação de memória (*new* & *delete*). | | Sim | | em todos os instanciamentos em listas, como em Player.cpp e Warlock.cpp assim como em FirstStage.cpp , SecondStage.cpp, LevelStateHandler.cpp, | |
|  | - Gabaritos/*Templates* criada/adaptados pelos autores (e.g. Listas Encadeadas via *Templates*). | | Sim | | Classe GenericList. GenericList.h e cpp. | |
|  | - Uso de Tratamento de Exceções (*try catch*). | | Sim | | Classe CompositeController.  CompositeController.h e cpp. | |
|  | **4** | | **Sobrecarga de:** | | | | | |
|  | - Construtoras e Métodos. | | Sim | | DynamicEntity.h e Entity.h  (construtora) e (métodos) Collider e LevelStateHandler .hs , .cpps | |
|  | - Operadores (2 tipos de operadores pelo menos). | | Sim | | GenericList.h | |
|  | **Persistência de Objetos (via arquivo de texto ou binário)** | | | | | |
|  | - Persistência de Objetos. | | Sim | | via texto , Principal.cpp (relativos a suas agregações).  via texto , LevelStateHandler.cpp (salva os objetos relativos ao nível). | |
|  | - Persistência de Relacionamento de Objetos. | | Sim | | via texto , LevelStateHandler.cpp e algumas funcoes auxiliares como getBoss() em SecondStage | |
|  | **5** | | **Virtualidade:** | |  | |  | |
|  | - Métodos Virtuais. | | Sim | | Controller.h,DynamicEntity.h | |
|  | - Polimorfismo | | Sim | | Principal.cpp (levelref), EventHandler.cpp,MenuHandler.cpp,Collider.cpp,Level.cpp(update & draw), LevelStateHandler.cpp | |
|  | - Métodos Virtuais Puros / Classes Abstratas | | Sim | | Level.h,Enemie.h,DynamicEntity.h,ButtonBase.h,MenuBase.h,Spell.h,Projectile.h. | |
|  | - Coesão e Desacoplamento | | Sim | | Projeto como todo , Destaque para separamento dos eventos via Composite (EventHandler e classes associadas) e dos menus/botões separando em funções mais especificas (MenuHandler e classes associadas). | |
|  | **6** | | **Organizadores e Estáticos** | | | | | |
|  | - Espaço de Nomes (*Namespace*) criada pelos autores. | | Sim | | Namespaces: AbstractEntity ,Models , Buttons , Menus , CompController. | |
|  | - Classes aninhadas (*Nested*) criada pelos autores. | | Sim | | GenericList.h,MenuRanking.h e cpp | |
|  | - Atributos estáticos e métodos estáticos. | | Sim | | Todas as classes (.hs e .cpps) terminadas em Model. | |
|  | - Uso extensivo de constante (*const*) parâmetro, retorno, método... | | Sim | | Projeto Como 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 | | Vector utilizado nos menus (parametrizados com buttonbase veja MenuBase.h) e list no level.h , parametrizada com Vector2f.  Também foi utilizada a classe string da stl e ainda string da sfml. Ambas utilizadas para gerenciamento de texto no MenuRanking.h e .cpp , em Level.cpp para o hud. Player.h contém um atributo do tipo string representando o nome, | |
|  | - Pilha, Fila, Bifila, Fila de Prioridade, Conjunto, Multi-Conjunto, Mapa **OU** Multi-Mapa. | | Sim | | map na classe CompositeController | |
|  | **Programação concorrente** | | | | | |
|  | *- Threads* (Linhas de Execução) no âmbito da Orientação a Objetos, utilizando Posix, C-Run-Time **OU** Win32API ou afins*.* | | *Nao* | |  | |
|  | *- Threads* (Linhas de Execução) no âmbito da Orientação a Objetoscom uso de Mutex, Semáforos, **OU** Troca de mensagens. | | *Nao* | |  | |
|  | **8** | | **Biblioteca Gráfica / Visual** | | | | | |
|  | - Funcionalidades Elementares. &  - Funcionalidades Avançadas como:   * tratamento de colisões * duplo *buffer* | | Sim | | Tratamento de colisões (Collider.cpp) , alocação de texturas (arquivos com nomes terminados em Model), o sfml por padrão utiliza duplo buffer. | |
|  | - Programação orientada e evento em algum ambiente gráfico.  **OU**  *- RAD – Rapid Application Development* (Objetos gráficos como formulários, botões etc). | | Sim | | O sistema de controle via composite trabalha diretamente com os eventos do sfml. | |
|  | **Interdisciplinaridades por meio da utilização de Conceitos de Matemática e/ou Física.** | | | | | |
|  | - Ensino Médio. | | *Sim* | | Gravidade. | |
|  | - Ensino Superior*.* | | *Sim* | | Rotação das flechas (Arrow.cpp), colisão (checkCollision em Collider.cpp). InsertionSort em MenuRanking.cpp | |
|  | **9** | | **Engenharia de Software** | | | | | |
|  | - Compreensão, melhoria e rastreabilidade de cumprimento de requisitos. & | | Sim | |  | |
|  | - Diagrama de Classes em *UML*. | | Sim | |  | |
|  | - Uso efetivo (quiçá) intensivo de padrões de projeto (particularmente GOF). | | Sim | | Composite (EventHandler.h e .cpp e associadas) , derivação de State (MenuHandler.h e .cpp e associadas) e Singletons (finais em Models.h e cpp). | |
|  | - Testes a luz da Tabela de Requisitos e do Diagrama de Classes. | | Sim | |  | |
|  | **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* | | Controle feito pelo github. | |
|  |  | | - Reuniões com o professor para acompanhamento do andamento do projeto. | | *Não* | | Apenas feita uma reunião em sala de aula. | |
|  |  | | - Reuniões com monitor da disciplina para acompanhamento do andamento do projeto. | | *Não* | | Feito apenas uma reunião presencial e retirada de dúvidas no email do mesmo. | |
|  |  | | - Revisão do trabalho escrito de outra equipe e vice-versa. | | *Não* | |  | |

# DISCUSSÃO E CONCLUSÕES

A composicao deste trabalho (Jogo em C++) representa uma grandiosa tarefa , tanto em tamanho quanto em valor de aprendizado. Se utilizado uma análise superficial do problema e de suas principais interações , sem uma devida modelagem , compreensão dos requisitos ( assim como se programa em materiais da universidade anteriores a esta) e técnicas relacionadas a orientação de objeto; A tarefa escala para um patamar de dificuldade muito superior e de forma desnecessária. A facilidade de interpretação e entendimento espacial do programa ,via UML, que a orientação de objetos proporciona é imensurável em relação ao seu custo inicial de uma implementação mais técnica , complexa e árdua. Diferente da programação habitual, a complexidade escalada da orientação a objetos visa esclarecer detalhes e deixar a programação mais fluida e mais clara. A principal vantagem observada no desenvolvimento deste trabalho foi : a possibilidade de se mudar partes do código, via encapsulamento , sem mexer em grandes blocos de comandos.

# DIVISÃO DO TRABALHO

Tabela 4. Lista de Atividades e Responsáveis.

|  |  |
| --- | --- |
| Atividades. | Responsáveis |
| Levantamento de Requisitos | Douglas e Pedro |
| Diagrama de Classes | Douglas e Pedro |
| Programação em C++ | Douglas |
| Escrita & Revisão do Trabalho | Douglas |