TRABALHO PARA A DISCIPLINA DE TÉCNICAS DE PROGRAMAÇÃO DO CURSO DE ENGENHARIA DE COMPUTAÇÃO DA UTFPR: *EXCALIBUR*

Gustavo Henrique Zeni e Ianca Polizelo

gus\_ghz@live.com, iancapolizelo@alunos.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

### 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* de plataforma, no formato de um jogo, 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 Excalibur, onde o jogador enfrenta inimigos em um determinado cenário. O jogo tem duas fases que se diferenciam por cenário e dificuldade ao enfrentar os inimigos (eles possuem mais vida) para o jogador. Para o desenvolvimento do jogo levantou-se textualmente os requisitos e os elementos gráficos que foram usados para compor a parte visual do jogo. Também foi elaborado uma modelagem (análise e projeto) usando o recurso de Diagrama de Classes e Diagrama de Atividades em *Unified Modeling Language* (*UML*). Subsequentemente, em linguagem de programação C++, realizou-se o desenvolvimento que contemplou os conceitos usuais de Orientação a Objetos, bem como alguns conceitos avançados como Classe Abstrata, Polimorfismo, Persistências de Objetos por Arquivos e uso do operador *this*. Depois da implementação, os testes e uso do jogo demonstraram sua funcionalidade conforme os requisitos e o projeto. Por fim, salienta-se que o projeto permitiu cumprir o objetivo de aprendizado visado.

**Palavras-chave ou Expressões-chave:** Artigo-Relatório para o Trabalho em Fundamentos de Programação 2, Trabalho Acadêmico Voltado à Implementação em C++, Trabalho Acadêmico Voltado para Implementação de Jogo em C++.

**Abstract** - This document is a report to the academic work of *Técnicas de Programação* as well as it presents the tools used to create the game. This game was motivated by the work of *Técnicas de Programação* to explore the applicability of object orientation programming concepts in C ++. The game takes place in a fantasy context, where the main character, the player, must face monsters equipped with a special sword called *Excalibur*. The necessary requirements for the accomplishment of the work, as well as the graphic tools, were written verbatim. A Class Diagram was also made in *UML*. After the implementation, tests were performed demonstrating their functionality according to the requirements.

**Key-words or Key-expressions:** Article-Report for Work on Programming Fundamentals 2, Academic Work Related to C++ Implementation, Academic Work Toward Implementation of Game in C ++.

# INTRODUÇÃO

Este documento apresenta o processo de realização do trabalho da disciplina *Técnicas de Programação*, lecionada pelo professor Jean Simão. Com o objetivo de aprofundar e aplicar conceitos de *Programação Orientada a Objetos,* objeto de estudo das aulas, foi proposto a realização de um *Software* em linguagem C++. O trabalho é proposto, principalmente, visando ampliar a aplicação dos conceitos aprendidos em aula, assim como a buscar novos conceitos, complementando todo o conteúdo abordado ao longo da disciplina.

O objeto de estudo do trabalho é um jogo de plataforma, onde são requisitados ao menos duas fases e três tipos de inimigos, os quais o jogador deverá enfrentar. O nome escolhido para esse projeto foi *Excalibur*, lembrando a lendária espada do Rei Arthur [1]. O jogo se passa em um contexto de fantasia, onde o jogador é transformado em um pinguim ao encontrar a espada lendária. Para poder voltar para sua forma original, deverá enfrentar monstros até encontrar o último chefe. Quando finalmente conseguir derrotar o poderoso monstro, voltará a sua forma original e a espada voltará a se esconder para que um novo desafiante a possa encontrar.

Para a realização deste *Software*, foi utilizado o *Modelo em Cascata*, o primeiro modelo de ciclo de vida de Engenharia de Software. Este consiste basicamente em um modelo linear, onde cada passo deve ser completado antes que o próximo possa ser iniciado. Primeiramente foi feita a análise de requisitos do jogo, levantando quais eram os elementos necessários para cada parte do sistema. Em seguida foi realizado o diagrama de classes em *UML,* modelando o jogo em si. Logo após, começou a implementação propriamente dita, seguindo o modelo previamente elaborado. E por fim, foi realizada uma série de testes, verificando assim se o projeto estava funcionando conforme deveria.

As seções subsequentes mostrarão mais detalhadamente o desenvolvimento do jogo, contendo tabelas de conceitos usados e não usados da linguagem C++ junto do paradigma Orientado a Objetos, requisitos previstos pelo professor para o jogo propriamente dito e divisão do trabalho entre os membros da equipe. Também mostrará uma explicação mais detalhada sobre o jogo e sua implementação. Estarão presentes uma série de imagens e diagramas usados no desenvolvimento do *Software*, tornando a explicação mais clara e abrangente.

# EXPLICAÇÃO DO JOGO EM SI

O jogo se inicia com o jogador já segurando a espada lendária e com sua forma transformada em um pinguim, devido à maldição de Excalibur. Com o objetivo de se livrar da maldição e ter sua forma recuperada, ele deverá enfrentar uma série de monstros que tentam o impedir de chegar até o Guardião da Espada. Quando chegar até o final da segunda fase e o Guardião ele encontrar, poderá tentar derrotá-lo para ser novamente um humano.

O jogo contará com duas fases, sendo diferenciadas entre si pela ambientação e pela quantidade de inimigos que cada uma possuirá.

A primeira fase se passa em uma floresta, onde existirão dois tipos de inimigos, morcegos e gosmas. O dano que o jogador deverá infligir contra esses elementos para os matar não será grande, deixando assim a dificuldade da fase baixa. Existirão também três tipos de obstáculos de deverão ser enfrentados, sendo eles pedras e buracos.

A segunda fase será dentro de uma caverna, onde o Guardião mora. Dentro dela existirão morcegos e esqueletos, inimigos um pouco mais fortes dos que foram encontrados na primeira fase. Para chegar até o chefe da fase, além de ter de matar os inimigos, o jogador deverá enfrentar mais três tipos de obstáculos, o quais são: espinhos no chão e lava.

Após enfrentar e derrotar o Guardião, a espada será tomada e a maldição será revertida, sendo assim, o jogador não mais terá a forma de um pinguim e voltará a ser um humano.

# DESENVOLVIMENTO DO JOGO NA VERSÃO ORIENTADA A OBJETOS

Nesta seção será realizada a explicação do desenvolvimento do *software* usando programação orientada a objetos em linguagem C++, para tal, será utilizada algumas tabelas e imagens, especificando cada coisa realizada ao longo de todo o processo.

Primeiramente, será listada os requisitos funcionais para o jogo e suas situações, se foram ou não realizadas. Após a finalização do jogo foi cumprido 86,7% dos requisitos funcionais previsto inicialmente.

Tabela 1. Lista de Requisitos do Jogo e suas Situações.

|  |  |  |
| --- | --- | --- |
| N. | Requisitos Funcionais | Situação |
| 1 | Apresentar menu de opções aos usuários do Jogo | Requisito previsto inicialmente e realizado |
| 2 | Permitir um ou dois jogadores aos usuários do Jogo | Requisito previsto inicialmente e realizado |
| 3 | Disponibilizar ao menos duas fases que podem ser jogadas sequencialmente ou selecionadas. | Requisito previsto inicialmente e realizado |
| 4 | Ter três tipos distintos de inimigos (o que pode incluir ‘Chefão’, vide abaixo) | Requisito previsto inicialmente e realizado |
| 5 | Ter a cada fase ao menos dois tipos de inimigos com número aleatório de instâncias, podendo ser várias instâncias. | Requisito previsto inicialmente e realizado |
| 6 | Ter inimigo “Chefão” na última fase | Requisito previsto e realizado |
| 7 | Ter três tipos de obstáculos. | Requisito previsto inicialmente e realizado |
| 8 | Ter em cada fase entre um e três tipos de obstáculos com número aleatório de instâncias (i.e., objetos). | Requisito previsto inicialmente e realizado |
| 9 | Ter representação gráfica de cada instância. | Requisito previsto inicialmente e realizado |
| 10 | Ter em cada fase um cenário de jogo com os obstáculos. | Requisito previsto inicialmente e realizado |
| 11 | Gerenciar colisões entre jogador e inimigos. | Requisito previsto inicialmente e realizado |
| 12 | Gerenciar colisões entre jogador e obstáculos. | Requisito previsto inicialmente e realizado |
| 13 | Permitir cadastrar/salvar dados do usuário, manter pontuação durante jogo, salvar pontuação e gerar lista de pontuação (*ranking*). | Requisito previsto inicialmente e não realizado |
| 14 | Permitir Pausar o Jogo | Requisito previsto inicialmente e realizado |
| 15 | Permitir Salvar Jogada. | Requisito previsto e não realizado |

Para a modelagem do software foi realizado um diagrama de classes em UML, conforme a figura abaixo:

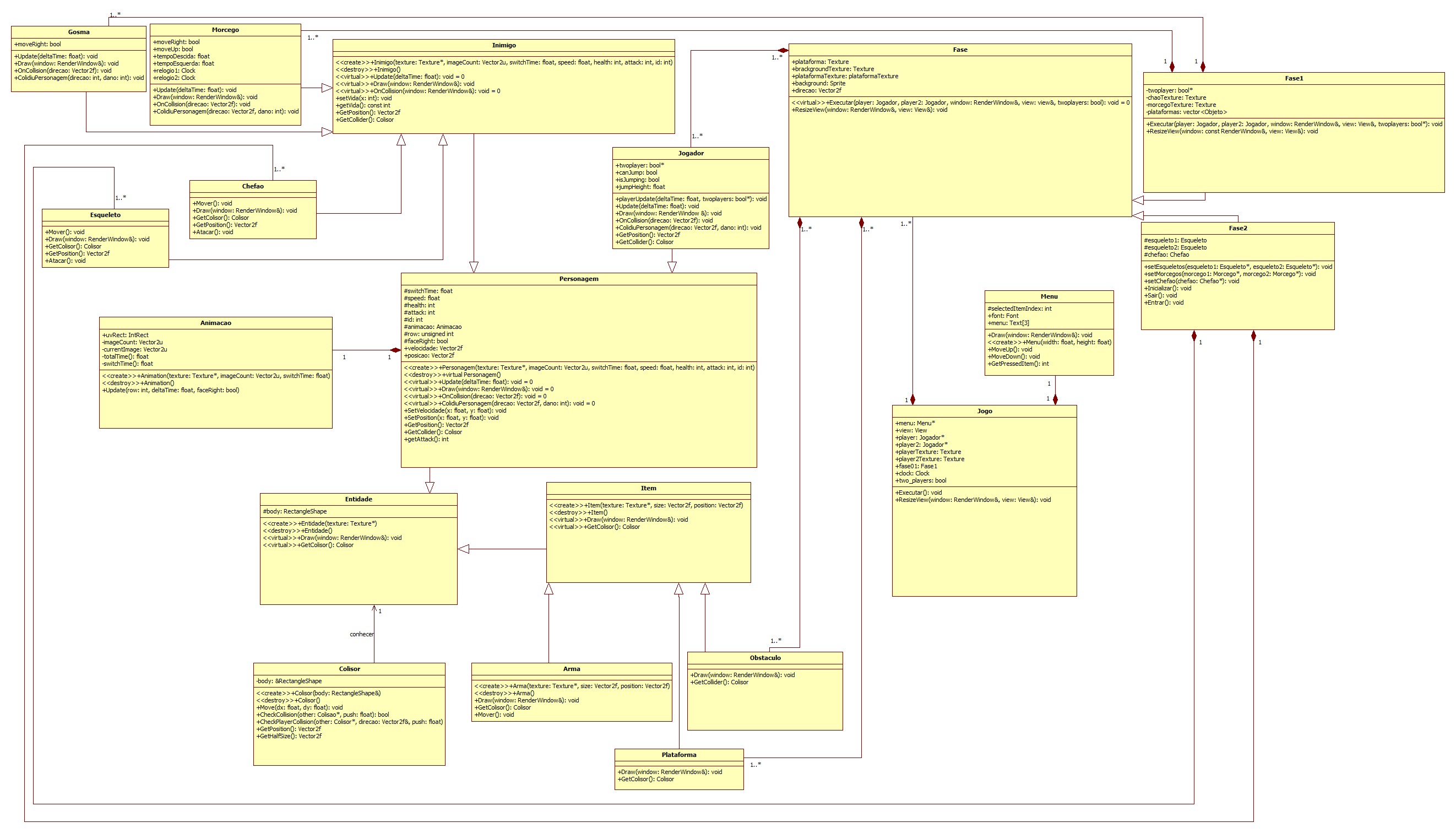


Figura 1. Diagrama de classes em UML do projeto.

As principais classes do software são: personagem, jogo, fase, item e colisor[A]. Pode-se ter uma visão geral do jogo analisando-as melhor. A primeira classe citada é a base para se construir os elementos animados do jogo, sendo eles jogador e inimigos. Estes possuem um corpo onde o formato é um retângulo e, dentro dele, é colocada a animação de cada entidade. A forma de retângulo foi escolhida para facilitar no tratamento de colisões, que será explicado posteriormente. A segunda é a classe principal do programa, sendo utilizada para executar, propriamente dito, o jogo. A terceira é a base para as duas fases criadas no projeto, sendo que será nela que os elementos gráficos são criados. E a quarta é a classe mãe dos obstáculos e plataformas, também tendo um corpo retangular.

A classe colisor se trata de uma classe Mediator [2][B], usada para gerenciar a colisão entre os personagens e os obstáculos durante o jogo. Foi escolhido encapsular a comunicação entre esses objetos afim de diminuir o acoplamento dentro do trabalho, assim os objetos que se relacionam entre si não precisam se conhecer diretamente. Esse relacionamento é realizado através de um mediador, no caso o colisor. A principal vantagem desse padrão de projeto utilizado é a centralização da comunicação, tornando sua manutenção muito mais simples.

Para verificar a colisão foi utilizado o conceito de calcular distâncias entre pontos. Cada entidade é tratada como um retângulo e então é calculado a distância entre os centros delas. Caso essa distância seja menor que a soma do centro até a borda de cada retângulo, então elas estão colidindo, ou seja, uma forma está sobreposta a outra.

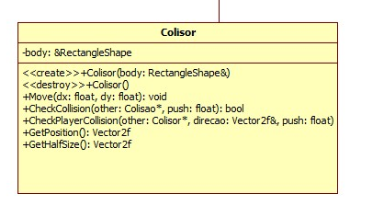


Figura 2. Classe Colisor em UML ampliada.

Não apenas os personagens tiveram a base do corpo em formato retangular, como todas as partes gráficas do jogo, como plataformas, cenários e inimigos, utilizando do mesmo conceito para calcular a colisão entre esses outros objetos.

Foi utilizada a biblioteca gráfica *SFML (Simple and Fast Multimedia Libray)* [3][C], por ser de simples utilização comparada a outras bibliotecas como *Allegro* [4][D] e por ser rápida. Grande parte da implementação foi realizada utilizando a documentação no próprio site da *SFML*, assim como tutoriais publicados no *Youtube* (principalmente para o colisor).

# TABELA DE CONCEITOS UTILIZADOS E NÃO UTILIZADOS

Esta seção mostra uma tabela de conceitos utilizados e não utilizados, e uma tabela com a justificativa para o uso ou não uso de cada conceito aprendido em aula. Após a finalização do software foi cumprido 75,8% dos conceitos teóricos listados abaixo.

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N.** | | Conceitos | Uso | Onde |
|  | **1** | **Elementares:** | | |
|  | - Classes, objetos. | Sim | Todos .h/.cpp |
|  | - Atributos (privados), variáveis e constantes. | Sim | Todos .h/.cpp |
|  | - Métodos (com e sem retorno). | Sim | Todos .h/.cpp |
|  | - Métodos (com retorno *const* e parâmetro *const*). | Sim | Todos .h/.cpp |
|  | - Construtores (sem/com parâmetros) e destrutores. | Sim | Todos .h/.cpp |
|  | - Classe Principal. | Sim | Main.cpp & Jogo.h/.cpp |
|  | - Divisão em .h e .cpp. | Sim | No projeto |
|  | **2** | **Relações de:** | | |
|  | - Associação. | Sim | Colisor.h/.cpp & Personagem.h/.cpp |
|  | - Agregação via associação. | Sim | Colisor.h/.cpp & Personagem.h/.cpp |
|  | - Agregação propriamente dita. | Sim | Personagem.h/.cpp & Jogo.h/.cpp & Fase1.h/.cpp & Fase2.h/.cpp & Fase.h/.cpp |
|  | - Herança elementar. | Sim | Jogador.h/.cpp & Inimigo.h/.cpp & Fase1.h/.cpp & Fase2.h/.cpp |
|  | - Herança em diversos níveis. | Sim | Morcego.h/.cpp & Gosma.h/.cpp & Esqueleto.h/.cpp & Chefao.h/.cpp & Obstaculo.h/.cpp & Plataforma.h/.cpp |
|  | - Herança múltipla. | Não |  |
|  | **3** | **Ponteiros, generalizações e exceções** | | |
|  | - Operador *this* | Sim | Colisor.cpp & Animacao.cpp |
|  | - Alocação de memória (*new* & *delete*) | Sim |  |
|  | - Gabaritos/*Templates* criada/adaptados pelos autores (e.g. Listas Encadeadas via Templates) | Não |  |
|  | - Uso de Tratamento de Exceções (*try catch*) | Não |  |
|  | **4** | **Sobrecarga de:** | | |
|  | - Construtoras e Métodos. | Não |  |
|  | - Operadores (2 tipos de operadores pelo menos) | Não |  |
|  | **Persistência de Objetos** | | |
|  | - Texto via Arquivos de Fluxo | Não |  |
|  | - Binário | Não |  |
|  | **5** | **Virtualidade:** |  |  |
|  | - Métodos Virtuais. | Sim | Personagem.h/.cpp & Inimigo.h/.cpp & Fase.h/.cpp & Item.h/.cpp |
|  | - Polimorfismo | Sim | Personagem.h/.cpp & Inimigo.h/.cpp & Fase.h/.cpp & Item.h/.cpp |
|  | - Métodos Virtuais Puros / Classes Abstratas | Sim | Personagem.h/.cpp & Inimigo.h/.cpp & Fase.h/.cpp & Item.h/.cpp |
|  | - Coesão e Desacoplamento | Sim | Todos .h/.cpp |
|  | **6** | **Engenharia de Software** | | |
|  | **- Levantamento de Requisitos Textualmente e Tabelado** | Sim | Relatório\_Excalibur.docs/.pdf |
|  | **- Diagrama de Classes em *UML*** | Sim | Excalibur.uml |
|  | **7** | Biblioteca Gráfica | | |
|  | - Funcionalidades Elementares. | Sim | Todos.h/.cpp |
|  | - Funcionalidades Avançadas como:   * tratamento de colisões * eventos * clock (contador) * métodos de entrada (teclado) * métodos de saída (window e view) | Sim | Todos.h/.cpp |
|  | **8** | **Interdisciplinaridades por meio da utilização de Conceitos de Matemática, Física etc** | | |
|  | - Ensino Médio(especificar quais Conceitos aqui):   * calcular distância entre dois pontos * verificar a distância de qual eixo é menor (indica se a colisão ocorreu primeiro verticalmente ou horizontalmente) | Sim | Colisor.h/.cpp |
|  | - Ensino Superior*:*   * Torricelli (para calcular a aceleração do jogador) | Sim | Jogador.h/.cpp |
|  | **Organizadores e Estáticos** | | |
|  | Espaço de Nomes (*Namespace*) criada pelos autores.  E/ou Classes aninhadas. | Não |  |
|  | Atributos estáticos e chamadas estáticas de métodos. | Sim | Jogo.h/.cpp |
|  | **9** | **Standard Template Library *(STL)* e String OO** | | |
|  | *Vector da STL* (p/ objetos ou ponteiros de objetos de classes definidos pelos autores) | Sim | Fase1.h/.cpp & Fase2.h/.cpp |
|  | A classe Pré-definida *String* ou equivalente. | Não |  |
|  | **10** | **Uso de Conceito Avançado no tocante a Orientação a Objetos.** | | |
|  | **Ou Padrões de Projeto:** GOF  **Ou Programação orientada a eventos e visual:** Objetos gráficos como formulários, botões etc (Listar apenas os utilizados) | Sim | Colisor.h/.cpp |

Tabela 3. Lista de Justificativas para Conceitos Utiliza e Não Utilizados no Trabalho.

|  |  |  |
| --- | --- | --- |
| No. | Conceitos | *Listar apenas os utilizados* Situação |
| 1 | **Elementares** | *Classe e Objetos* foram utilizados, pois são essenciais para o paradigma *Orientado a Objetos*.  *Atributos, Variáveis e Constantes* foram utilizados pelos mesmo motivos que *Classes e Objetos*.  *Métodos (com e sem retorno)* foram utilizados, pois é necessário funções indicando o que cada classe e objeto irá realizar.  *Métodos (com retorno const e parâmetro const)* foram utilizados, pois era necessário funções que não alterassem os parâmetros ou que o retorno não fosse alterado por demais funções.  *Construtores (sem/com parâmetros) e destrutores* foram utilizadores, pois eram necessários que as variáveis de determinados objetos fossem inicializadas ou destruídas.  *Classe Principal* foi utilizada, pois é preciso uma classe para organizar e executar todo o jogo. Porém foi utilizada com outro nome (Jogo.h/.cpp).  *Divisão em .h e .cpp* foi utilizado visando a melhor organização do projeto. |
| 2 | **Relações** | *Associação* foi utilizado porque a classe Personagem precisava conhecer a classe Colisor para que fosse possível verificar a colisão entre personagens e obstáculos.  *Agregação via associação* foi utilizado pelo mesmo motivo de associação.  *Agregação propriamente dita* foi utiliza, pois era preciso que certas classes tivessem objetos de outras classes, como no caso da classe Jogo que agrega Menu.  *Herança elementar* foi utilizada, porque era preciso que diversas classes herdassem atributos e métodos de classes mais básicas e abstratas.  *Herança em diversos níveis* foi utilizada, pois era preciso que uma classe fosse derivada de outra classe que também era uma derivada.  *Herança múltipla* não foi utilizada, pois nenhuma das classes do jogo era derivada de duas ou mais classes ao mesmo tempo. |
| 3 | **Ponteiros, generalizações e exceções** | *Operador this* foi utilizado, pois era necessário apontar para o objeto que chamava determinada função, mesmo não sabendo qual objeto seria.  *Alocação de memória (new & delete)*, pois era preciso alocar e desacolar objetos para a maximização do uso da memória.  *Gabaritos/Templates* não foram utilizados, porque não se julgou necessário para a construção do software.  *Tratamento de exceções*... |
| 4 | **Sobrecarga de:** | *Construtoras e Métodos* não foi utilizada, pois não foi preciso que nenhuma construtora ou método fosse sobrecarregado.  *Operadores* não foi utilizada pelo mesmo motivo que construtoras e métodos não foram sobrecarregados. |
| 4 | **Persistência de Objetos** | *Texto via Arquivos de Fluxo* não foi utilizada, pois os dados do jogo não foram salvos  *Binário* não foi utilizada pelo mesmo motivo que texto via arquivos de fluxo. |
| 5 | **Virtualidade:** | *Métodos Virtuais* foram utilizados, porque assim não era necessário ficar repetindo a implementação de funções que seriam as mesmas nas diversas classes derivadas de uma única classe abstrata.  *Polimorfismo* foi utilizado, pois existiam várias funções em classes diferentes que precisavam ter o mesmo nome.  *Métodos Virtuais Puros/Classes Abstratas* foram usadas para que a mesma função tivesse diferentes implementações em suas classes derivadas.  *Coesão e Desacoplamento* foi utilizado para que ficasse mais fácil fazer manutenção em determinada classe sem ter que mexer em outras classes/objetos. |
| 6 | **Engenharia de Software** | *Levantamento de Requisitos Textualmente e Tabelado* foi utilizado para facilitar na hora de montar o diagrama de classes, tornando o trabalho mais rápido e com menos erros.  *Diagrama de Classes em UML* foi realizado, pois facilita na montagem e implementação do software em si, minimizando os erros. |
| 7 | **Biblioteca Gráfica** | *Funcionalidades Elementares* foram utilizadas para poder mostrar graficamente no *display* os elementos do jogo como jogador, inimigos, obstáculos, etc.  *Funcionalidades Avançadas* como tratamento de colisões foi utilizada, pois era precisa fazer com que o jogador morresse quando algum inimigo colidisse com ele, ou que o inimigo morresse quando o jogador pudesse sobre ele. |
| 8 | **Interdisciplinaridades por meio da utilização de Conceitos de Matemática, Física, etc** | *Conceitos do ensino médio* foram utilizados, pois era preciso calcular distância entre dois pontos para verificar uma colisão.  *Conceitos do ensino superior* foram utilizados, porque era preciso calcular a aceleração dos personagens. |
| 8 | **Organizadores e Estáticos** | *Namespace* não foi utilizado, pois...  *Atributos estáticos e chamadas estáticas de métodos* foram utilizados, porque era necessário um atributo que não mudasse de valor ao longo do jogo. |
| 9 | **Standard Template Library(STL) e String OO** | *Vector* foi utilizado para facilitar na inclusão e remoção das entidades dentro do jogo como inimigos e obstáculos.  *A Classe String* não foi utilizada, pois não foi preciso salvar nomes. |
| 10 | **Uso de conceitos avançados.** | Foi utilizado *Padrões de Projeto*, mais especificamente *Mediator*, pois era necessário centralizar a comunicação entre diversos objetos em relação ao tratamento de colisões.  Programação orientada a eventos e visual foi utilizada, pois a biblioteca gráfica SFML trabalha principalmente em cima de eventos. |

# COMPARAÇÃO ENTRE DESENVOLVIMENTOS

Após a realização deste software o conhecimento sobre outro paradigma de programação além do procedimental ampliou a visão sobre os diversos métodos de se arquitetar um programa. Apesar de ter sido apenas o primeiro projeto realizado seguindo essa linha de raciocino, foi o suficiente para que fossem levantadas algumas comparações.

Na programação procedimental todas as funções, inicializações e atributos se dão no mesmo lugar, dificultando enormemente a verificação de erros e manutenção do software. Já em programação orientada a objetos cada parte do programa é tratada como sendo separada, podendo ser desacoplada, o que aumenta a facilidade na solução de erros e na manutenção do código em si.

Outra grande vantagem da orientação a objetos é que a reutilização de implementações fica muito mais viável, pois todas as partes do programa são construídas da forma mais genérica possível, podendo se encaixar em vários lugares diferentes.

Uma boa analogia para entender este tipo de paradigma é a manutenção de um automóvel. Quando alguma parte do veículo está com defeito é simples e rápido tirar a peça com defeito e procurar erros apenas nela, sem influenciar no resto de todo o projeto. Assim que o erro é encontrado, é rapidamente corrigido e colocado novamente no local. Assim acontece também com a programação orientada a objetos.

# DISCUSSÃO E CONCLUSÕES

A grande vantagem deste projeto foi a aplicação prática dos conceitos aprendidos em aula de um modo visual, tornando o paradigma orientado a objetos de muito mais fácil compreensão. Para o desenvolvimento de todo o jogo, foi preciso utilizar todos os conceitos teóricos da matéria assim como pesquisar as diversas funcionalidades da biblioteca gráfica escolhida, ampliando as habilidades de autodidatismo como um todo, habilidade de extrema importância não apenas na vida acadêmica como na profissional.

Para conseguir realizar o trabalho de forma tranquila e cumprindo todos os requisitos de conceitos era preciso montar um cronograma de atividades, seguindo-o rigorosamente e antecipadamente. Caso esse passo não fosse cumprido, o aproveitamento dos conceitos aprendidos em aula não seria 100% alcançado. Isto foi uma das principais aprendizagens que o software conseguiu proporcionar para os realizadores, criando nesses um novo modo de pensar seguindo esse padrão.

Foi preciso também o estudo de engenharias de software para que o projeto fosse realizado de forma mais dinâmica e com menos margem para erros, processo não utilizado antes em trabalhos de disciplinas anteriores. O estudo foi de grande importância, tanto para este software como para os demais que um dia serão projetados pelos membros da dupla.

Apesar do pouco tempo para a realização do jogo da disciplina devido a atrasos por parte dos integrantes da dupla, este projeto proporcionou grande aprendizagem não apenas para a disciplina de *Técnicas de Programação* como para todo o curso de *Engenharia de Computação* e para a vida profissional como um todo.

# DIVISÃO DO TRABALHO

Esta seção contém uma tabela salientando quem desenvolveu cada classe/módulo do *software* e realizou demais atividades como as de ‘engenharia de *software’*, a redação do trabalho escrito, a revisão da redação do trabalho e a preparação da apresentação do trabalho.

Tabela 4. Lista de Atividades e Responsáveis.

|  |  |
| --- | --- |
| Atividades. | Responsáveis |
| Levantamento de Requisitos | Gustavo e Ianca |
| Diagramas de Classes | Mais Ianca que Gustavo |
| Programação em C++ | Gustavo e Ianca |
| Implementação da classe Jogador | Gustavo e Ianca |
| Implementação da classe Animacao | Gustavo e Ianca |
| Implementação da classe Inimigo | Gustavo |
| Implementação da classe Menu | Ianca |
| Implementação da classe Colisor | Gustavo e Ianca |
| Implementação da classe Personagem | Mais Gustavo que Ianca |
| Implementação da classe Jogo | Mais Gustavo que Ianca |
| Implementação da classe Fase | Gustavo e Ianca |
| Implementação da classe Fase1 | Gustavo e Ianca |
| Implementação da classe Fase2 | Gustavo e Ianca |
| Implementação da classe Item | Gustavo e Ianca |
| Implementação da classe Gosma | Gustavo e Ianca |
| Implementação da classe Morcego | Gustavo e Ianca |
| Implementação da classe Esqueleto | Gustavo e Ianca |
| Implementação da classe Chefao | Gustavo e Ianca |
| Implementação da classe Arma | Gustavo e Ianca |
| Implementação da classe Plataforma | Gustavo e Ianca |
| Implementação da classe Obstaculo | Gustavo e Ianca |
| Implementação da classe Buraco | Gustavo e Ianca |
| Montagem da apresentação do Trabalho | Ianca |
| Escrita do Trabalho | Ianca |
| Revisão do Trabalho | Gustavo e Ianca |

# AGRADECIMENTOS

Para a realização deste trabalho é necessário agradecer principalmente ao professor da disciplina, Jean Simão, e ao monitor Vitor Corrêa, pela assistência e aconselhamento de ambas as partes. Também agradecemos a dupla composta por Mateus Freitas e Alexandre Nadolni por revisarem o relatório do trabalho e por dicas dadas ao longo do processo de realização do jogo. E, por último, mas não menos importante, damos os devidos agradecimentos a Paulo Bayer, por também ajudar na revisão da parte escrita do projeto.

# REFERÊNCIAS CITADAS NO TEXTO

[1] SUPERINTERESSANTE. Um rei como nunca houve: Arthur, Acessado em 13/06/2018:

https://super.abril.com.br/historia/um-rei-como-nunca-houve-arthur/

[2] BRIZENO, Marcos. Desenvolvimento de software #showmethecode, Acessado em 13/06/2018:

<https://brizeno.wordpress.com/category/padroes-de-projeto/mediator/>

[3] SFML. Simple and Fast Multimedia Library, Acessado em 13/06/2018:

<https://www.sfml-dev.org/>

[4] ALLEGRO. A game programming library, Acessado em 13/06/2018:

https://liballeg.org/

# REFERÊNCIAS UTILIZADAS NO DESENVOLVIMENTO

[A] VONCK, Hilze. Canal Hilze Vonck – Playlist: SFML 2.4 For Beginners, Acessado em 13/06/2018:

<https://www.youtube.com/watch?v=axIgxBQVBg0&list=PL21OsoBLPpMOO6zyVlxZ4S4hwkY_SLRW9>

[B] BRIZENO, Marcos. Desenvolvimento de software #showmethecode, Acessado em 13/06/2018:

https://brizeno.wordpress.com/category/padroes-de-projeto/mediator/

[C] SFML. Simple and Fast Multimedia Library, Acessado em 13/06/2018:

<https://www.sfml-dev.org/>

[D] ALLEGRO. A game programming library, Acessado em 13/06/2018:

https://liballeg.org/