# TRABALHO PARA A DISCIPLINA DE TÉCNICAS DE PROGRAMAÇÃO DO CURSO DE ENGENHARIA DE COMPUTAÇÃO DA UTFPR: "O CÉU TÁ CAINDO"

**Lucas Gonçalves Pinto, Matheus dos Santos Pedrozo de Lima**
lucasgonsalves18@hotmail.com, matheus.pedrozo@gmail.com

Disciplina: **Técnicas de Programação – CSE20** / S17 – 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 propõe o desenvolvimento de um software no formato de um jogo de plataforma com o intuito de desenvolver os tópicos de programação orientado a objeto em C++ ministrados na disciplina.Para tal, neste trabalho foi produzido o jogo "O céu tá caindo" onde o jogador enfrenta inimigos em um dado cenário.O jogo é constituído de três fases que se diferenciam por gerações aleatórias das mesmas. Para a realização do jogo foram abordados os requisitos textualmente propostos e elaborado modelagem(análise e projeto) usando como recurso o Diagrama de Classes em Linguagem de Modelagem Unificada (Unified Modeling Language - UML) usando como base um diagrama previamente proposto.Subsequentemente, em linguagem de programação C++, realizou-se o desenvolvimento que contemplou os conceitos usuais de Orientação a Objetos como Classes, Objeto e Relacionamento, bem como alguns conceitos avançados como Classe Abstrata, Polimorfismo, Gabaritos e Biblioteca Padrão de Gabaritos (Standard Template Library - STL).

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

**Abstract** - The subject "Técnicas de Programação" proposes the development of a software in the format of a platform game aiming the expansion of the programing topics object oriented in C++ as during classes. For this purpose, a game called "O céu tá caindo" was made, which consists of one or two players confronting enemies in a certain scenario. The game has three stages that can never be the same due the random generation program structure. For the accomplishment of the game the requirements given by the teacher were elaborated in the Unified Modeling Language (UML) using a given diagram as a starting point. Thereafter, in the C++ language, a development was made containing the usual concepts of object Oriented like Classes, Object and Relationship, and some advanced concepts like Abstract Class, Polyphormism, Templates and Standard Template Library.

**Key-words or Key-expressions :** Paper Model to the Academic Work of Programming Course, Academic Work Related to C++ Implementation, Object Oriented programing, Game, Chicken Little.

## INTRODUÇÃO

Trabalho referente a disciplina Técnicas de Programação, que consiste em um software em forma de jogo de plataforma com o objetivo de desenvolver os conteúdos ministrados na disciplina Técnicas de Programação referente à Orientação a Objeto em C++.

O trabalho tem como objetivo a prática e por conseguinte o aperfeiçoamento das técnicas de orientação a objeto em C++ e a utilização de diagramas de classe tais como UML (Unified Modeling Language - UML).

Assim, nas próximas seções serão abordados explicações da forma como foi implementado o software, o desenvolvimento orientado a objeto, quais conceitos foram abordados, diagrama UML e por fim uma conclusão sobre o projeto.

## EXPLICAÇÃO DO JOGO EM SI

No jogo “O céu tá caindo”, o jogador ao executar o programa será direcionado a um menu, no qual lhe será dado as seguintes opções: “Jogar” (iniciar o menu de fases), “CO-OP” (jogar em dupla), “Ranking” e “Exit” (sair do jogo). Ao escolher a opção “Jogar”, o usuário será redirecionado a um menu que contém as fases, e poderá escolher qual irá jogar, sendo estas, a “Fase 1” , “Fase 2” e “Fase 3”.


Figura 1. Menu.


Figura 2. Menu de fases

Escolhida a fase, o usuário poderá fazer entrada de comandos para controlar o personagem, fazendo-o andar e pular. Em cada fase o jogador irá se deparar com instâncias geradas aleatoriamente e colocadas na fase da mesma forma. Assim, as fases 1 e 2 se diferenciam, pois cada uma é criada de uma forma diferente da outra, e em cada uma contém dois tipos de inimigos e três tipos de objetos que fazem interações com o personagem.

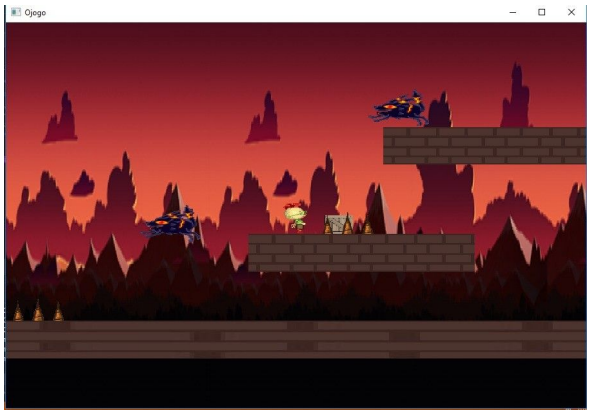


Figura 3. Fase 1.

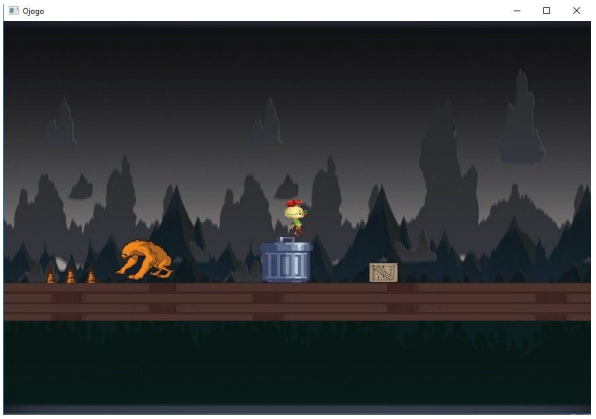


Figura 4. Fase 2.

Na “Fase 3” é a onde se encontra apenas  o “Chefão”, esta fase, em exceção às demais, as instâncias não são geradas aleatoriamente.

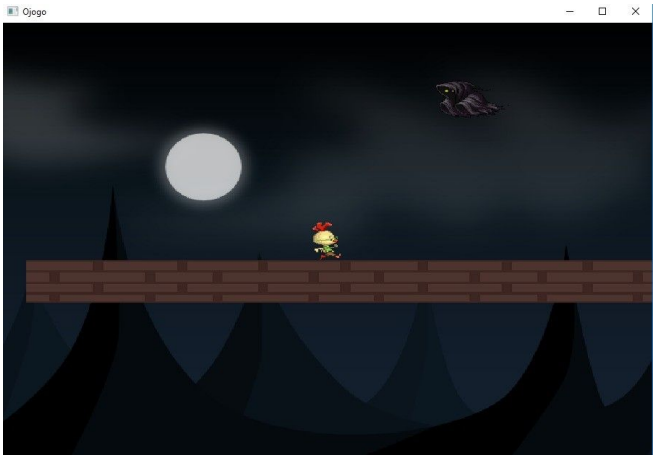


Figura 5. Fase 3, chefão.

Durante a execução das fases, a qualquer momento o usuário poderá pausar o jogo, fazendo isso, todas as instâncias iram permanecer no local anterior à ação e lhe será dado as opções de “Resume” (voltar ao jogo), ”Save”, “Back to Menu” (retorna ao menu inicial) e “Exit” (sair do programa).


Figura 6. Menu de pausa.

No entanto não há sistema de morte e por conseguinte o jogo não possui fim.

## 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 Menu e seu respectivo objeto. |
| 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 Jogador cujos objetos são agregados em jogo, podendo ser um ou dois efetivamente. |
| 3 | Disponibilizar ao menos duas fases que podem ser jogadas sequencialmente ou selecionadas. | Requisito previsto inicialmente e realizado. | Requisito cumprido via classes Jogo no qual um objeto de Stage é agregado em Jogo, e sendo criado em execução com os devidos parâmetros para especificar a fase. |
| 4 | Ter três tipos distintos de inimigos (o que pode incluir 'Chefão', vide abaixo). | Requisito previsto inicialmente e realizado. | Requisito cumprido via classe Inimigo e suas derivadas, cada um possuindo uma característica distinta de movimento e representação gráfica. |
| 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 classe stage que possui uma função que gera uma quantidade limitada e aleatória de inimigos e sendo chamada na construtora das classes Stage1 e Stage2. |
| 6 | Ter inimigo "Chefão" na última fase | Requisito previsto inicialmente e realizado | Realizado através da classe Enemy_Boss e possuindo uma característica de movimento própria. Sendo o único inimigo instanciado na fase 3 através da construtora da mesma. |
| 7 | Ter três tipos de obstáculos. | Requisito previsto inicialmente e realizado. | Requisito cumprido inclusive via Classes Obstacle, Obstacle_A, Obstacle_B e Obstacle_C. Cada qual possui uma característica única, seja dar dano, ser imóvel ou móvel. |
| 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 a luz de uma função de construção Aleatória quantidades e posições na classe Stage1. |
| 9 | Ter representação gráfica de cada instância. | Requisito previsto inicialmente e não concluído. | Cada instância que necessita de representação gráfica possui um Sprite para si, porém não |

| | | | |
|---|---|---|---|
| | | | foi possível concluir animações como a de morte de instâncias ou de período ocioso dos players. |
| 10 | Ter em cada fase um cenário de jogo com os obstáculos. | Requisito previsto inicialmente e realizado. | Requisito cumprido inclusive via construtoras de fases e funções de criação de instâncias disponivel na classe Stage. |
| 11 | Gerenciar colisões entre jogador e inimigos. | Requisito previsto inicialmente e realizado. | Requisito cumprido inclusive via classe Collider e inimigos baseadas em vectors e função de checagem de colisão na classe Stage. |
| 12 | Gerenciar colisões entre jogador e obstáculos. | Requisito previsto inicialmente e realizado. | Requisito cumprido inclusive via classe Collider, lista de plataformas e obstáculos baseadas em vetor, assim como o uso da função de checagem de colisão na classe Stage. |
| 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. | Requisito NÃO realizado. |
| 14 | Permitir Pausar o Jogo | Requisito previsto inicialmente e realizado | Requisito cumprido inclusive via agregação da classe MenuPausa na classe Stage, sendo chamado a partir da função de execução de Stage. |
| 15 | Permitir Salvar Jogada. | Requisito previsto não realizado | Requisito NÃO realizado. |

Isto feito a explicação do desenvolvimento segue devendo-se:

- A classe Stage Agrega listas de obstáculos, inimigos, plataformas e entidades para poder fazer o gerenciamento da fase. Não o bastante possui uma lista de entidade agregada que por si só agrega uma classe List que é associada por dependência à classe Entidade.
- O projeto como todo possui duas classes que se destacam pelo seu uso, comparativamente a um cérebro e um coração. sendo elas a classe Entidade e a classe Stage. Entidade por ser uma classe abstrata ela tem um poder de gerir outros objetos por meio de listas, comandado então o funcionamento de funções gerais como a atualização dos atributos de suas classes derivadas. Já a classe Stage ela tem por função agregar listas de entidades e gerenciar a fase, ou seja, ela permite a funcionalidade da abstração da classe entidade.

- Para obter os requisitos de número aleatórios de instâncias foi gerado aleatoriamente um número de instâncias a serem criadas a partir da função “rand” da biblioteca “stdio.h” e implementado uma função que a partir das funcionalidades disponíveis na biblioteca padrão do c++ “random”, for possível gerar posições um pouco mais “aleatórias” para passar por parâmetros para as construtoras de seus respectivos objetos.
- Os conceitos interdisciplinares usados teve grande atuação na implementação de funções de colisão. Por meio da geometria analitica foi possivel identificar se um objeto se sobrepõe sobre o outro, então, realizando suas devidas reações a estes eventos. Bem como uma parte muito básica da física foi usa, que é a gravidade. Apesar de ser um conceito simples, ela aplica certa dinâmica no projeto pela sua consequência de gerar o aumento da velocidade vertical de objetos afetados por ela.

-

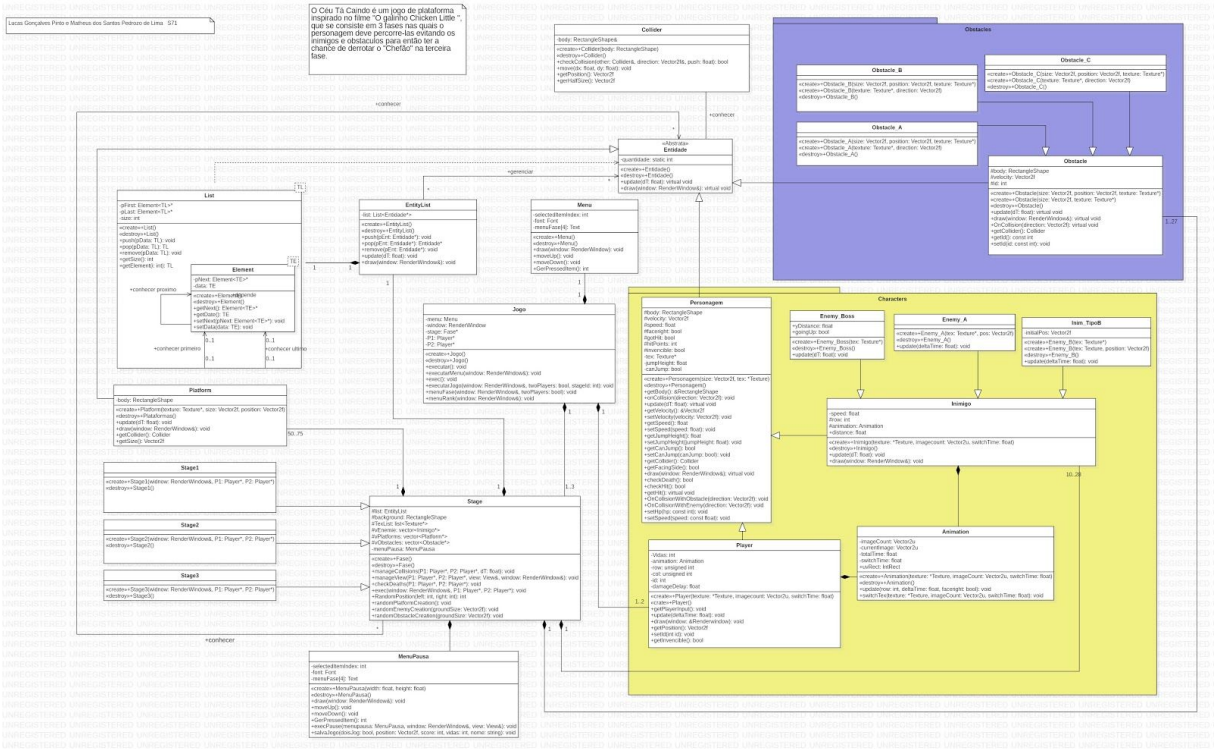

Figura 2. Diagrama de Classes de base em UML.

## TABELA DE CONCEITOS UTILIZADOS E NÃO UTILIZADOS

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

| N. | Conceitos | Uso | Onde / O quê |
|---|---|---|---|
| 1 | **Elementares:** | | |
| | - Classes, objetos. &<br>- Atributos (privados), variáveis e constantes. & | Sim | Todos .h e .cpp |

| | | | | |
|---|---|---|---|---|
| | | - Métodos (com e sem retorno). | | |
| | | - Métodos (com retorno *const* e parâmetro *const*). &<br>- Construtores (sem/com parâmetros) e destrutores | Sim | Todos .h e .cpp |
| | | - Classe Principal. | Sim | Main.cpp & Jogo..h/.cpp |
| | | - Divisão em .h e .cpp. | Sim | No desenvolvimento como um todo. |
| | **2** | **Relações de:** | | |
| | | - Associação direcional. &<br>- Associação bidirecional. | Sim | Associações direcionais contemplados em entre as classe: EntityList, List, Collider e Stage para com a classe Entidade |
| | | - Agregação via associação. &<br>- Agregação propriamente dita. | Sim | Agregação das classes: Menu, Stage e Player na classe Jogo; Agregação de MenuPausa, Obstacles, Platform, Inimigos e EntityList na classe Stage; Classe List em EntityList; E por fim Agregação de animator em Player e Inimigo. |
| | | - Herança elementar. &<br>- Herança em diversos níveis. | Sim | Heranças contempladas nas classes: Personagem, Platform, Obstacle, Obstacle_A, Obstacle_B, Obstacle_C, Player, Inimigo, Enemy_A, Enemy_B, Enemy_Boss , Stage, Stage1, Stage2 and Stage3 |
| | | - Herança múltipla. | Não | |
| | **3** | **Ponteiros, generalizações e exceções** | | |
| | | - Operador *this*. | Sim | List.h, Animation.cpp, Personagem.cpp, Player.cpp, Obstacle.h, |
| | | - Alocação de memória (*new* & *delete*). | Sim | Jogo.cpp, Stage.cpp, Stage1.cpp, Stage2.cpp, Stage3.cpp, |
| | | - Gabaritos/*Templates* criada/adaptados pelos autores (e.g. Listas Encadeadas via *Templates*). | Sim | Lista simplesmente encadeada adaptada por Matheus em List.h |
| | | - Uso de Tratamento de Exceções (*try catch*). | Não | |
| | **4** | **Sobrecarga de:** | | |
| | | - Construtoras e Métodos. | Sim | Construtoras das seguintes classes sobrecarregadas: Enemy_B, Obstacle, Obstacle_A, Obstacle_B, Obstacle_B |

| | | | | |
|---|---|---|---|---|
| | | - Operadores (2 tipos de operadores pelo menos). | Não | |
| | | **Persistência de Objetos (via arquivo de texto ou binário)** | | |
| | | - Persistência de Objetos. | não | |
| | | - Persistência de Relacionamento de Objetos. | não | |
| | **5** | **Virtualidade:** | | |
| | | - Métodos Virtuais. | sim | Personagem.cpp |
| | | - Polimorfismo | sim | Nas funções "updates" das classes: Enemy_A, Enemy_B, Enemy_C, Obstacles e Player |
| | | - Métodos Virtuais Puros / Classes Abstratas | Sim | Entidade.h/.cpp |
| | | - Coesão e Desacoplamento | Sim | No projeto como todo por meio de generalização de Classe e funções usadas. |
| | **6** | **Organizadores e Estáticos** | | |
| | | - Espaço de Nomes (*Namespace*) criada pelos autores. | Sim | todos .h/.cpp para definir o escopo do código criado pelo autor |
| | | - Classes aninhadas (*Nested*) criada pelos autores. | Sim | List.h |
| | | - Atributos estáticos e métodos estáticos. | Não | |
| | | - Uso extensivo de constante (*const*) parâmetro, retorno, método... | Não | |
| | **7** | **Standard Template Library *(STL)* e String OO** | | |
| | | - A classe Pré-definida *String* ou equivalente. &<br>- *Vector* e/ou *List* da *STL* (p/ objetos ou ponteiros de objetos de classes definidos pelos autores) | Sim | Uso das classes Vector e List em Stage.h/.cpp |
| | | - Pilha, Fila, Bifila, Fila de Prioridade, Conjunto, Multi-Conjunto, Mapa **OU** Multi-Mapa. | Não | |
| | | **Programação concorrente** | | |
| | | - *Threads* (Linhas de Execução) no âmbito da Orientação a Objetos, utilizando Posix, C-Run-Time **OU** Win32API ou afins. | Não | |
| | | - *Threads* (Linhas de Execução) no âmbito da Orientação a Objetos com uso de Mutex, Semáforos, **OU** Troca de mensagens. | Não | |
| | **8** | **Biblioteca Gráfica / Visual** | | |
| | | - Funcionalidades Elementares. &<br>- Funcionalidades Avançadas como:<br>● tratamento de colisões<br>● duplo *buffer* | Sim | Funcionalidades básicas como criação de formas geométricas, uso de sprites para animação, e gerenciamento gráfico durante execução. Ademais, o tratamento de colisões foi feito com o uso de funções básicas da Biblioteca juntamente com um algoritmo de colisão do tipo (AABB) |

| | | | | |
|---|---|---|---|---|
| | | - Programação orientada a evento em algum ambiente gráfico.<br>**OU**<br>- *RAD – Rapid Application Development* (Objetos gráficos como formulários, botões etc). | Não | |
| | | **Interdisciplinaridades por meio da utilização de Conceitos de Matemática e/ou Física.** | | |
| | | - Ensino Médio. | Sim | Matemática e Física básica no tratamento de movimento de objetos, como definição de velocidade e aceleração gravitacional |
| | | - Ensino Superior. | Sim | Uso de geometria análitca na decomposição de diversos vetores e análise de interação de objetos planos. |
| | **9** | **Engenharia de Software** | | |
| | | - Compreensão, melhoria e rastreabilidade de cumprimento de requisitos. & | Não | |
| | | - Diagrama de Classes em *UML*. | Sim | Diagrama produzido três vezes a luz dos requisitos propostos e ajustes necessários. |
| | | - Uso efetivo (quiçá) intensivo de padrões de projeto (particularmente GOF). | Não | |
| | | - Testes a luz da Tabela de Requisitos e do Diagrama de Classes. | Sim | Foram feitos testes durante a produção do programa, visto que, o diagrama de classes foi produzido antes da confecção do código. |
| | **10** | **Execução de Projeto** | | |
| | | - Controle de versão de modelos e códigos automatizado (via SVN e/ou afins) **OU** manual (via cópias manuais). &<br>- Uso de alguma forma de cópia de segurança (backup). | Sim | Backups manuais dos arquivos e salvos tanto em computadores diferentes, quanto na nuvem. |
| | | - Reuniões com o professor para acompanhamento do andamento do projeto. | Sim | Uma reunião |
| | | - Reuniões com monitor da disciplina para acompanhamento do andamento do projeto. | Não | |
| | | - Revisão do trabalho escrito de outra equipe e vice-versa. | Não | |

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

| **No.** | **Conceitos** | ***Listar apenas os utilizados* Situação** |
|---|---|---|
| 1 | **Elementares** | Classe e Objetos foram utilizado porque na orientação de objetos é uma ótima prática de desacoplamento de |

| | | |
|---|---|---|
| | | objetos que possuem características semelhantes, organização e para generalização do programa,<br>Atributos foram utilizados porque são eles que permitem uma grande diversificação de objetos. |
| 2 | **Relações** | Associação foi utilizado porque para certos objetos terem uma execução correta de acordo com os requisitos ela necessita conhecer atributos de um objeto, porém não deve alterá-los.<br>Agregação foi utilizado porque certos objetos precisam conceitualmente ser constituídos de outros que possuem suas próprias funcionalidades para execução.<br>A herança foi usada amplamente para generalizar classes que têm características fundamentais muito parecidas, porém existem diferenças que não podem ser analisadas por fazerem parte de um espectro alternativo de situações. |
| 3 | **Ponteiros, generalizações e exceções** | Ponteiros foram amplamente utilizados pela facilidade em estruturar os dados do programa.<br>New e delete teve uso para criar instâncias e removê-las para que não tenha um uso de memória desnecessário.<br>O operador “this” foi usado por não precisar dar nomes diferentes a elementos que representam o mesmo, apenas possuindo um diferente escopo.<br>Uma Lista template foi usada com o fim de ter um controle mais pessoal sobre os objetos que estamos tratando, além de agregar o aprendizado da generalização. |
| 4 | **Sobrecarga de métodos** | Sobrecarga de Construtoras foi usado para garantir que todas classes possuem ao menos uma construtora padrão vazia, ademais, para que possa ter objetos com características mais específicas.<br>Sobrecarga de operadores não foi utilizado por não observarmos necessidade para o projeto. |
| 5 | **Virtualidade** | A virtualidade teve uso para garantir que objetos similares por herança tenho comportamentos diferentes com uma mesma chamada de função.<br>A classe Entidade foi criada como abstrata pois ela pode controlar todas suas derivadas que por mais de diferente, terão funções iguais que realizarão funcionalidades diferentes uma das outras.<br>A coesão e desacoplamento é o lema da matéria e o uso dela foi para garantir que o código sempre esteja fácil de alterar e corrigir erros. |
| 6 | **Organizadores e Estáticos** | Namespaces foi utilizado para demarcar qual parte do código foi produzido por indivíduo da dupla. |

| | | |
|---|---|---|
| | | Uma classe aninhada foi utilizada na lista template porque não há motivo de outro objeto ter acesso ao seu conteúdo sem ser por meio do gerenciamento proporcionado pela lista.<br>O uso de const não foi previsto em diagrama de classes préviamente, portanto durante a implementação foi escasso o uso do mesmo. |
| 7 | **Standard Template Library *(STL)* e String OO** | Foram usadas classes pré-definidas Vector e List para poder diversificar o tipo de estrutura de dados usados no projeto e por ser de fácil uso. |
| 8 | **Biblioteca Gráfica / Visual** | O uso da biblioteca gráfica 'SFML' foi escolhido por parecer uma biblioteca relativamente fácil de se lidar e por ser orientada a objetos, seus métodos foram usados estensivamentes por se mostrarem muito úteis na organização dos dados.<br>A programação orientada a eventos não foi utilizada por não ter sido préviamente previsto para realização do projeto. |
| 9 | **Engenharia de Software** | Padrões de Projeto não foram utilizados por não terem sido previstos pelas dupla préviamente antes do início da implementação do código. |
| 10 | **Execução de Projeto** | O uso de controle de versões foi muito usado por fornecer um modo seguro de ter salvo arquivos que podem ser utilizados caso tenha algum problema na confecção do projeto.<br>As visitas ao monitor e o professor não foram possíveis pois não cogitamos a possibilidade. |

## REFLEXÃO COMPARATIVA ENTRE DESENVOLVIMENTOS

A Partir da realização de um projeto utilizando a orientação a objetos é perceptível uma generalização muito maior que a procedimental, visto que utilizando conceitos de C++ Orientado a Objeto permite uma coesão e desacoplamento muito mais perceptível. Também faz se necessário um planejamento muito mais virtuoso e presente em todos os estágios do desenvolvimento de um projeto como este, desde conceitos, padrões de projeto e diagramas de classes.

## DISCUSSÃO E CONCLUSÕES

Com o desenvolvimento do trabalho foi possível absorver e relacionar conceitos do C++ Orientado a Objeto, Engenharia de Software, e de padrões de projeto. No entanto, com o decorrer do desenvolvimento, vários "obstáculos" surgiram, como planejar o desenvolvimento, divisão de tarefas, trabalhar em equipe e desenvolver os tópicos pré estabelecidos.

O resultados obtidos não foram totalmente satisfatórios, visto que, requisitos como salvar o jogo e ranking não foram possíveis de serem concluídos

Não obstante, o projeto serviu para aprimorar os conceitos de Fundamentos de Programação e Estrutura de Dados, além de permitir o aprendizado de bibliotecas gráficas em especial SFML.

## DIVISÃO DO TRABALHO

Tabela 4. Lista de Atividades e Responsáveis.

| Atividades. | Responsáveis |
|---|---|
| Levantamento de Requisitos | Lucas e Matheus |
| Diagramas de Classes | Lucas e Matheus |
| Programação em C++ | Mais Matheus que Lucas |
| Implementação de *Template* | Matheus |
| Implementação das classes prevista em UML | Mais Matheus que Lucas |
| Implementação de algoritmos de aleatoriedade | Matheus |
| Implementação de 2 jogadores | Matheus |
| Implementação de Menus | Lucas |
| Implementação de inimigos | Matheus |
| Implementação de colisão | Matheus |
| Implementação de representações gráficas | Lucas e Matheus |
| Implementação de Pausa | Lucas |
| Escrita do Trabalho | Lucas e Matheus |
| Revisão do Trabalho | Lucas e Matheus |

## REFERÊNCIAS UTILIZADAS NO DESENVOLVIMENTO


[A] Biblioteca <random> C++:
http://www.cplusplus.com/reference/random/
[B] Lista encadeada adaptada:
http://cncpp.divilabs.com/2013/12/c-program-code-to-implement-singly.html
[C] SFML Tutorial para iniciantes:
https://www.youtube.com/watch?v=axIgxBQVBg0
[D] Modelo para confecção de relatório:
http://www.dainf.ct.utfpr.edu.br/~jeansimao/Fundamentos2/TopicosTrab/Modelo_para_Trabalho_TecnicasProgramacao.pdf