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

Victor Boechat Errera, Thiago Seiji MIyasawa
verrera@alunos.utfpr.edu.br, thiagomiyasawa@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 / Sistemas de Informação
Universidade Tecnológica Federal do Paraná - UTFPR
Avenida Sete de Setembro, 3165 - Curitiba/PR, Brasil - CEP 80230-901

Resumo – Visando o aprendizado de técnicas de engenharia de software e o uso da linguagem c++ para programação orientada o objetos, a disciplina Técnicas de Programação exige o desenvolvimento de um jogo de plataformas, para isto foi escolhido o Zelda++, baseando-se no jogo do NES Zelda 2: The adventure of Link, no qual o jogador percorre várias fases enfrentando inimigos em diversos cenários. O jogo tem 2 fases que se diferenciam pelo mapa e pelos inimigos presentes. O desenvolvimento do jogo foi baseado em uma lista de requisitos e um diagrama de classes genérico proposto previamente em Linguagem de Modelagem Unificada Unified Modeling Language - UML). O jogo foi feito na linguagem C++ e no seu desenvolvimento foram contemplados tantos conceitos usuais (Classe e Objeto) quanto conceitos avançados (Classe Abstrata, Polimorfismo, Persistência de Objetos por Arquivos e etc.) de Orientação a Objetos. Após a implementação, foram feitos testes pelos desenvolvedores que demonstraram sua funcionalidade conforme os requisitos e a modelagem elaborada. Por último, se notabiliza o fato de que o desenvolvimento do projeto permitiu cumprir o objetivo de aprendizado visado.

Palavras-chave ou Expressões-chave – Artigo-Relatório para o trabalho de Técnicas de Programação, Projeto Acadêmico voltado para a implementação em C++, Utilização da Programação Orientada a Objetos para Desenvolvimento, Implementação baseada em Uma Biblioteca Gráfica.

Abstract - *This document presents a model for the manuscript to the academic work of Programming Techniques (Técnicas de Programação) as well as it presents general instructions/specifications about this academic work and details about its evaluation process. 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

Este trabalho é realizado ao longo da disciplina de Técnicas de Programação, representando cinquenta por cento (50%) da nota da disciplina. Visando, através do desenvolvimento de um jogo de plataformas, o uso da linguagem C++ para o aprendizado sobre a Programação Orientada a Objetos (POO) e o uso de bibliotecas Gráficas como o SFML (*Simple and Fast Multimedia Library*). O desenvolvimento foi realizado em dupla e baseado em uma tabela de requisitos e um diagrama de classes base em UML, ambos previamente disponibilizados pelo Professor.

O objeto de estudo para este trabalho, é o desenvolvimento de um jogo de Plataformas visando o cumprimento de uma série de requisitos tanto funcionais quanto conceituais definidos pelo Professor. Desse modo, através do ciclo da Engenharia de Software e de várias reuniões tanto com o professor quanto com os monitores da disciplina foi feito o desenvolvimento do projeto. Utilizando-se para tal da linguagem C++, a POO e a biblioteca gráfica SFML.

O método utilizado para o desenvolvimento foi o ciclo da Engenharia de Software. Iniciando-se pela compreensão dos requisitos dados pelo Professor. Seguido pela modelagem do projeto através de um diagrama de classes em UML baseado no diagrama base disponibilizado de antemão. Por último é feita a implementação utilizando a linguagem C++ e a POO.

O resto deste deste relatório será composto pela explicação do jogo criado e uma descrição do desenvolvimento, utilizando para isso a tabela de requisitos funcionais e o modelo do código em UML. Seguido pelas tabelas de requisitos conceituais, sendo a primeira para especificar os conceitos que foram ou não utilizados e a segunda para justificar os usos dos conceitos utilizados, e pelas reflexões, discussões, conclusões e considerações. Ao fim, estão a divisão de trabalho, os agradecimentos e as referências.

EXPLICAÇÃO DO JOGO EM SI

O zelda ++ é composto de duas fases de sobrevivência, ou seja a fase tem um tempo de duração no qual inimigos nascem continuamente, sendo elas:

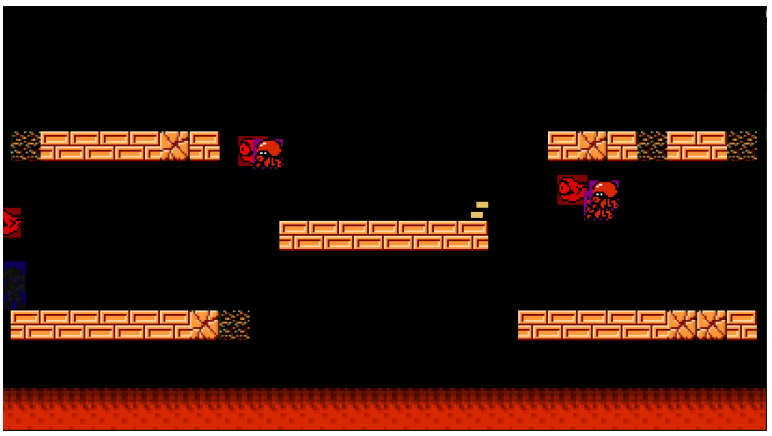

figura 1. primeira fase.

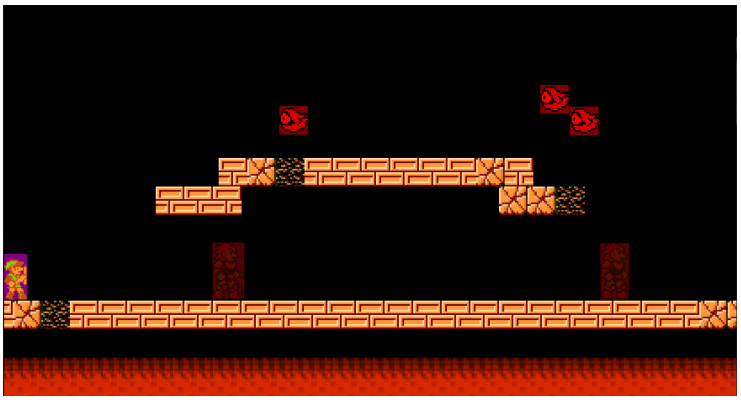

figura 2. segunda fase.

Dentro das fases estão dispostos 3 tipos de inimigos e 4 tipos de obstáculos, sendo os inimigos o moa que é um olho voador que fica invisível, o octorok que é um polvo voador atirador de projéteis e o ganondorf que é o boss e pode aumentar seu dano de tempos em tempos, sendo que ao longo da fase os jogadores acumulam pontos ao matá-los. E os obstáculos estão divididos em espinhos que são obstáculos causadores de dano, lava um obstáculo que mata o personagem instantaneamente, plataforma falsa um obstáculo que some após o jogador ficar um tempo em cima dele e a plataforma que é um obstáculo base que serve como chão. Sendo os sprites utilizados para eles os da imagem a seguir:

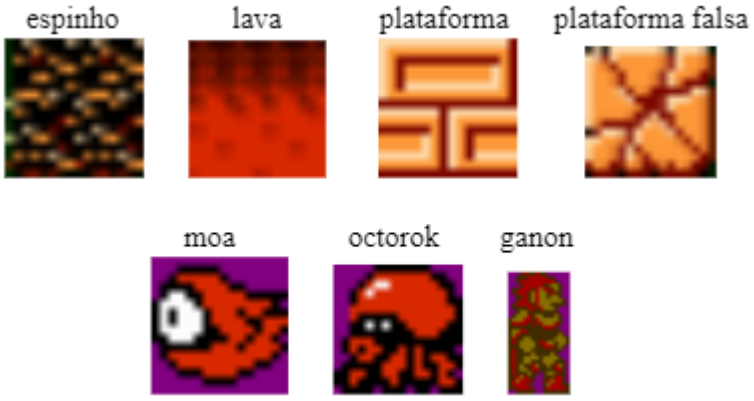

figura 3. conjunto de sprites usados.

Cada jogador terá uma aparência, sendo elas:

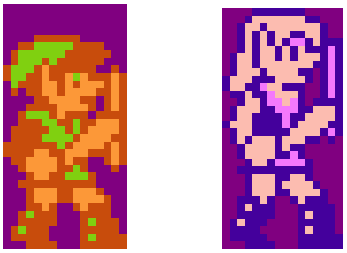

figura 4. sprites dos jogadores 1 e 2.

DESENVOLVIMENTO DO JOGO

O desenvolvimento do jogo se iniciou pela leitura do artigo modelo disponibilizado pelo Professor no site da disciplina, encontrando-se nele tanto detalhes sobre a avaliação quanto quanto às instruções/especificações para a sua realização. Sendo os requisitos funcionais encontrados na tabela, agora preenchida com base no trabalho desenvolvido, a seguir:

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

N.	Requisitos Funcionais	Situação	Implementação
1	Apresentar graficamente menu de opções aos usuários do Jogo, no qual pode se escolher fases, ver colocação (*ranking*) de jogadores e demais opções pertinentes.	Requisito previsto inicialmente e realizado.	Requisito cumprido através das classes Menu e ranking e seus respectivos objetos, com suporte da SFML.

2	Permitir um ou dois jogadores com representação gráfica 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 classe Jogador, sendo possível escolher se serão 1 ou 2 através do menu.
3	Disponibilizar ao menos duas fases que podem ser jogadas sequencialmente ou selecionadas, via menu, nas quais jogadores tentam neutralizar inimigos por meio de algum artifício e vice-versa.	Requisito previsto inicialmente e realizado.	Requisito realizado pelas classes Fase, PrimeiraFase e SegundaFase.
4	Ter pelo menos três tipos distintos de inimigos, cada qual com sua representação gráfica, sendo que ao menos um dos inimigos deve ser capaz de lançar projetil contra o(s) jogador(es) e um dos inimigos deve ser um 'Chefão'.	Requisito previsto inicialmente e realizado. .	Requisito cumprido através das classes Moa, Octorok, Ganondorf e seus respectivos objetos.
5	Ter a cada fase ao menos dois tipos de inimigos com número aleatório de instâncias, podendo ser várias instâncias e sendo pelo menos 3 instâncias por tipo.	Requisito previsto inicialmente e realizado.	Requisito cumprido inclusive via funções nas classes PrimeiraFase e SegundaFase, sendo eles gerados toda vez que todos os inimigos da tela morrem.
6	Ter três tipos de obstáculos, cada qual com sua representação gráfica, sendo que ao menos um causa dano em jogador se colidirem.	Requisito previsto inicialmente e realizado.	Requisito realizado através das 4 classes derivadas da classe obstáculo.
7	Ter em cada fase ao menos dois tipos de obstáculos com número aleatório de instâncias (*i.e.*, objetos), sendo pelo menos 3 instâncias por tipo.	Requisito previsto inicialmente e realizado.	Requisito realizado pelas funções convertePlatF e converteEsp na classe fase.
8	Ter em cada fase um cenário de jogo constituído por obstáculos, sendo que parte deles seriam plataformas ou similares, sobre as quais pode haver inimigos e podem subir jogadores.	Requisito previsto inicialmente e realizado.	Requisito cumprido inclusive via função virtual pura criaMapa da classe Fase e implementada nas classes derivadas
9	Gerenciar colisões entre jogador para com inimigos e seus projéteis, bem como entre jogador para com obstáculos. Ainda, todos eles devem sofrer o efeito da gravidade no âmbito deste jogo de plataforma vertical e 2D.	Requisito previsto inicialmente e realizado.	Requisito cumprido atrav´´es da classe GerenciadorColisões e da função gravidade na classe entidade

10	Permitir: (1) salvar nome do usuário, manter/salvar pontuação do jogador (incrementada via neutralização de inimigos) controlado pelo usuário e gerar lista de pontuação (*ranking*). **E** (2) Pausar e **Salvar** Jogada.	Requisito previsto inicialmente e realizado.	Requisito cumprido através das classes Menu , MenuGameOver e Ranking.
Total de requisitos funcionais apropriadamente realizados. *(Cada tópico vale 10%, sendo que para ser contabilizado deve estar realizado efetivamente e não parcialmente)*			**100%** (cem por cento).

O jogo foi desenvolvido utilizando de uma série de pacotes, sendo eles o das listas, das fases, dos gerenciadores e o das entidades que possui dois outros pacotes em seu interior, o dos personagens e o dos obstáculos. Pertencendo a estes pacotes a maioria das classes do projeto.

O pacote das listas contém as classes Lista e ListaEntidades, a primeira é um template de uma lista simplesmente encadeada. A segunda é uma especificação da Lista para entidades. Ambas as classes são utilizadas pelo código para, através do polimorfismo, armazenar todas as entidades do jogo.

Dentro do pacote das fases estão tanto a classe Fase quanto as classes PrimeiraFase e SegundaFase que a derivam. A Fase é uma classe abstrata e contém todos os atributos e funções comuns entre as duas fases, uma vez que ambas são utilizadas através de polimorfismo e se diferenciam pelo mapa e pelos inimigos presentes.

O pacote dos gerenciadores contém o GerenciadorColisoes e o GerenciadorGrafico, o GerenciadorColisoes tem como função gerenciar as colisões de todas as entidades e os efeitos delas. O GerenciadorGrafico por outro lado gerencia a janela e a exibição de todos os Entes.

O pacote das entidades possui a classe entidade, da qual herdam todos os personagens e obstáculos, e os pacotes personagens e obstáculos. O primeiro contém a classe Personagem e suas derivadas, sendo elas a classe Jogador e a classe Inimigo, da qual as classes específicas de Inimigos herdam. No segundo estão a classe Obstaculo, da qual se derivam as classes que constituem o mapa de cada uma das fases.

Fora dos pacotes estão as classes Jogo, Ente, e as que constituem o menu. A classe Jogo funciona como a classe principal e é onde fica o loop do jogo. Da Ente são derivadas todas as Entidades. O menu é constituído pelas classes Menu, Ranking, MenuGameOver e CaixaDeTexto.

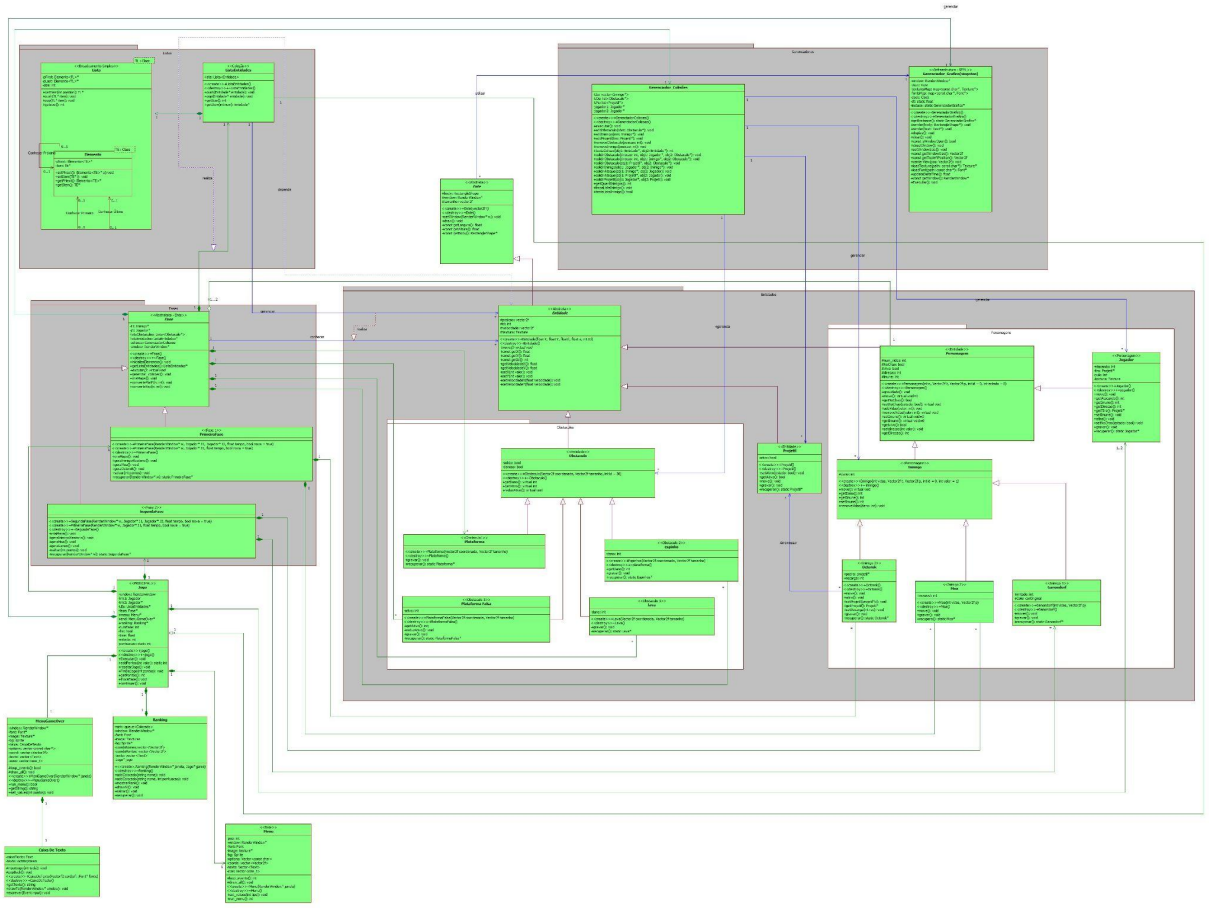

Figura 5. Diagrama de Classes de base em UML.

TABELA DE CONCEITOS UTILIZADOS E NÃO UTILIZADOS

Além dos requisitos funcionais, este trabalho também possui uma série de requisitos conceituais. Os quais estão listados na tabela a seguir:

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

N.	Conceitos	Uso	Onde / O quê
1	**Elementares:**		
1.1	- Classes, objetos. & - Atributos (privados), variáveis e constantes. & - Métodos (com e sem retorno).	Sim	Todos .h e .cpp, como nas classes Jogador e Menu.
1.2	- Métodos (com retorno *const* e parâmetro *const*). & - Construtores (sem/com parâmetros) e destrutores	Sim	Todas as classes possuem construtoras e destrutoras, e algumas possuem métodos com const como a classe Ente.
1.3	- Classe Principal.	Sim	Classe Jogo instanciada na main.cpp
1.4	- Divisão em .h e .cpp.	Sim	No desenvolvimento como um todo, como na classe Fase.
2	**Relações de:**		
2.1	- Associação direcional. & - Associação bidirecional.	Sim	bidirecional: classe MenuGameOver com a Classe Jogo direcional: classe GerenciadorColisoes com a classe Jogador, por exemplo

2.2	- Agregação via associação. & - Agregação propriamente dita.	Sim	via associação: classe ListaEntidades com a classe Jogo propriamente dita: a classe Ranking com a classe CaixaDeTexto
2.3	- Herança elementar. & - Herança em diversos níveis.	Sim	Em alguns dos .h e .cpp, como nas classes Entidade que herda Ente e na Personagem que herda Entidade
2.4	- Herança múltipla.	Não	Requisito não realizado.
3	**Ponteiros, generalizações e exceções**		
3.1	- Operador *this* para fins de relacionamento bidirecional.	sim	No relacionamento entre MenuGameOver e Jogo
3.2	- Alocação de memória (*new* & *delete*).	sim	Em vários dos .h e .cpp, como na classe Jogo.
3.3	- Gabaritos/*Templates* criada/adaptados pelos autores (*e.g.*, Listas Encadeadas via *Templates*).	sim	Na classe Lista.
3.4	- Uso de Tratamento de Exceções (*try catch*).	não	Requisito não realizado.
4	**Sobrecarga de:**		
4.1	- Construtoras e Métodos.	sim	em várias .h e .cpp, como na classe PrimeiraFase.
4.2	- Operadores (2 tipos de operadores pelo menos – Quais?).	não	Requisito não realizado..
---	**Persistência de Objetos (via arquivo de texto ou binário)**		
4.3	- Persistência de Objetos.	sim	Em várias .h e .cpp, como na classe Jogador
4.4	- Persistência de Relacionamento de Objetos.	sim	Na classe Octorok, o relacionamento com o Projétil.
5	**Virtualidade:**		. . .
5.1	- Métodos Virtuais Usuais.	sim	Em várias classes, como na classe Inimigo.
5.2	- Polimorfismo.	sim	Em vários .cpp, feito através da ListaEntidades, em específico na Fase.
5.3	- Métodos Virtuais Puros / Classes Abstratas.	sim	Em várias classes, como na classe Fase.
5.4	- Coesão/Desacoplamento efetiva e intensa com o apoio de padrões de projeto.	sim	isso pode ser encontrado em vários lugares, um exemplo é a classe CaixaDeTexto que não foi criada dentro da classe MenuGameOver
6	**Organizadores e Estáticos**		
6.1	- Espaço de Nomes (*Namespace*) criado pelos autores.	sim	Em grande parte das classes, como na classe Entidade
6.2	- Classes aninhadas (*Nested*) criada pelos autores.	sim	Na classe Lista
6.3	- Atributos estáticos e métodos estáticos.	sim	Em várias classes, como na função recuperar da classe jogador
6.4	- Uso extensivo de constante (*const*) parâmetro, retorno, método...	sim	em entidades têm a função getX que é uma função constante
7	**Standard Template Library *(STL)* e String OO**		
7.1	- 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	Na classe CaixaDeTexto

7.2	- Pilha, Fila, Bifila, Fila de Prioridade, Conjunto, Multi-Conjunto, Mapa **OU** Multi-Mapa.	sim	Na classe Ranking foi usada uma fila
---	**Programação concorrente**		
7.3	- *Threads* (Linhas de Execução) no âmbito da Orientação a Objetos, utilizando Posix, C-Run-Time **OU** Win32API ou afins.	não	Requisito não realizado.
7.4	- *Threads* (Linhas de Execução) no âmbito da Orientação a Objetos com uso de Mutex, Semáforos, **OU** Troca de mensagens.	não	Requisito não realizado.
8	**Biblioteca Gráfica / Visual**		
8.1	- Funcionalidades Elementares. **&** - Funcionalidades Avançadas como: ● tratamento de colisões ● duplo *buffer*	sim	Em várias .h e .cpp, como na classe GerenciadorGrafico para exibir as Entidades
8.2	- Programação orientada e evento efetiva (com gerenciador apropriado de eventos inclusive) em algum ambiente gráfico. **OU** - *RAD – Rapid Application Development* (Objetos gráficos como formulários, botões etc).	não	Requisito não realizado.
---	**Interdisciplinaridades via utilização de Conceitos de Matemática Contínua e/ou Física.**		
8.3	- Ensino Médio Efetivamente.	sim	uso do conceito de vetores para a manipulação das posições e velocidades de todas as Entidades.
8.4	- Ensino Superior Efetivamente.	não	Especificar quais conceitos aqui.
9	**Engenharia de Software**		
9.1	- Compreensão, melhoria e rastreabilidade de cumprimento de requisitos. &	sim	Feito através do diagrama em UML e das tabelas de requisitos
9.2	- Diagrama de Classes em *UML*.	sim	Feito no starUML
9.3	- Uso efetivo e intensivo de padrões de projeto *GOF*, *i.e.*, mais de 5 padrões.	não	Requisito não realizado.
9.4	- Testes à luz da Tabela de Requisitos e do Diagrama de Classes.	sim	Feito ao longo do desenvolvimento para garantir o cumprimento dos requisitos
10	**Execução de Projeto**		
10.1	- Controle de versão de modelos e códigos automatizado (via github e/ou afins). & - Uso de alguma forma de cópia de segurança (*i.e.*, *backup*).	sim	via github : https://github.com/ErreraV/jogo-TecProg
10.2	- Reuniões com o professor para acompanhamento do andamento do projeto.	sim	4 reuniões dias 3/11, 10/11, 17/11 e 24/11
10.3	- Reuniões com monitor da disciplina para acompanhamento do andamento do projeto.	não	Requisito não realizado.
10.4	- Revisão do trabalho escrito de outra equipe e vice-versa.	sim	Ian Ishikawa e Pedro Neves
Total de conceitos apropriadamente utilizados. *(Cada grande tópico vale 10% do total de conceitos. Assim, por exemplo, caso se tenha feito metade de um tópico, então valeria 5%.)*			**77,5%** (setenta e sete e meio por cento).

A seguir está a tabela com a justificativa para a utilização dos conceitos da tabela 2:

Tabela 3. Lista de Justificativas para Conceitos Utilizados.

No.	**Conceitos**	***Listar apenas os utilizados* Situação**
1	**Elementares**	
1.1	- Classes, objetos. & - Atributos (privados), variáveis e constantes. & - Métodos (com e sem retorno).	São conceitos básicos da orientação a objetos
1.2	- Métodos (com retorno *const* e parâmetro *const*). & - Construtores (sem/com parâmetros) e destrutores	Métodos const foram utilizados para evitar mudanças equivocadas em funções. Construtoras são necessárias para a inicialização de objetos, enquanto destrutoras servem para guiar o computador na hora de desalocar algum objeto
1.3	- Classe Principal.	Permite melhor utilizar a orientação a objetos
1.4	- Divisão em .h e .cpp.	torna o código mais legível e organizado.
2	**Relações**	
2.1	- Associação direcional. & - Associação bidirecional.	Associação bidirecional foi utilizado porque facilitava o acesso da pontuação pelo ranking e o Jogo executa o ranking
2.2	- Agregação via associação. & - Agregação propriamente dita.	Agregação via associação foi usada em um dos casos para facilitar o jogo executar todos os movimento dos personagens
2.3	- Herança elementar. & - Herança em diversos níveis.	Aumenta o desacoplamento e permite o uso do polimorfismo.
3	**Ponteiros, generalizações e exceções**	
3.1	- Operador this para fins de relacionamento bidirecional.	Utilizado para relacionar o jogo com o MenuGameOver na construtora do MenuGameOver.
3,2	- Alocação de memória (*new* & *delete*).	necessário para alocar memória para objetos manipulados através de ponteiros e para deslocá-los.
3.3	- Gabaritos/*Templates* criada/adaptados pelos autores (*e.g.*, Listas Encadeadas via *Templates*).	Utilizado para criar uma lista simplesmente encadeada que foi utilizada para armazenar e manipular objetos em várias partes do código.
4	**Sobrecarga de:**	
4.1	- Construtoras e Métodos.	Utilizada para construir objetos de diferentes formas com base em parâmetros específicos.
	Persistência de Objetos (via arquivo de texto ou binário)	
4.3	- Persistência de Objetos.	Requisito funcional.
4.4	- Persistência de Relacionamento de Objetos.	Necessário para que os octoroks não percam seus projéteis
5	**Virtualidade:**	
5.1	- Métodos Virtuais Usuais.	Utilizados para, quando necessário, especificar métodos nas classes derivadas.
5.2	- Polimorfismo.	Permite manipular diferentes tipos de entidades através de uma única lista.
5.3	- Métodos Virtuais Puros / Classes Abstratas.	Utilizado para chamar métodos de classes derivadas através de polimorfismo.
5.4	- Coesão/Desacoplamento efetiva e intensa com o apoio de padrões de projeto.	Boa prática da orientação a objetos.
6	**Organizadores e Estáticos**	
6.1	- Espaço de Nomes (*Namespace*) criado pelos autores.	Utilizado para melhorar a organização do código.

6.2	- Classes aninhadas (*Nested*) criada pelos autores.	Permite o encapsulamento de objetos.
6.3	- Atributos estáticos e métodos estáticos.	Fundamental para criar variáveis de "globais" para uma classe. Torna possível chamar funções de uma classe sem o uso de um objeto.
6.4	- Uso extensivo de constante (*const*) parâmetro, retorno, método...	Evita a manipulação acidental de alguma variável ou parâmetro.
7	**Standard Template Library *(STL)* e String OO**	
7.1	- A classe Pré-definida *String* ou equivalente. **&** - *Vector* e/ou *List* da *STL* (p/ objetos ou ponteiros de objetos de classes definidos pelos autores)	String usado para trabalhar com o nome dos colocados no ranking List foi usado para ter uma lista de projéteis para poder gerenciar as suas colisões
7.2	- Pilha, Fila, Bifila, Fila de Prioridade, Conjunto, Multi-Conjunto, Mapa **OU** Multi-Mapa.	Foi usada para poder facilitar a manutenção da ordem do ranking
8	**Biblioteca Gráfica / Visual**	
8.1	- Funcionalidades Elementares. **&** - Funcionalidades Avançadas.	Necessário para o cumprimento de requisitos.
8.3	- Ensino Médio Efetivamente.	Uso de vetores para direcionar os movimentos e manipular as posições das entidades
9	**Engenharia de Software**	
9.1	-Compreensão, melhoria e rastreabilidade de cumprimento de requisitos	Permite monitorar progresso do código.
9.2	-Diagrama de Classes em UML	Ajuda a analisar como o projeto está estruturado.
9.34	- Testes à luz da Tabela de Requisitos e do Diagrama de Classes.	Auxilia no monitoramento do progresso do projeto.
10	**Execução de Projeto**	
10.1	- Controle de versão de modelos e códigos automatizado (via github e/ou afins). & - Uso de alguma forma de cópia de segurança (*i.e.*, *backup*).	Segurança caso algo dê errado.
10.2	- Reuniões com o professor para acompanhamento do andamento do projeto.	Requisito, ajuda a direcionar o projeto.
10.4	- Revisão do trabalho escrito de outra equipe e vice-versa.	Melhora a qualidade do relatório.

REFLEXÃO COMPARATIVA ENTRE DESENVOLVIMENTOS

O desenvolvimento orientado a objetos permite, quando comparado ao procedimental, permite maior organização e maior adaptabilidade do software. Principalmente em grandes projetos, onde o grande desacoplamento possibilitado pela OO torna mais fácil tanto a manutenção quanto a expansão do programa.

DISCUSSÃO E CONCLUSÕES

Com a conclusão do projeto pode se perceber um grande desenvolvimento em nosso conhecimento sobre a orientação a objetos e o ciclo de da engenharia de software, além de

maior facilidade no uso de ferramentas como o Github e o debugger. Por fim, foram realizados 100% dos requisitos funcionais, ainda que com alguns erros, e 77,5% dos requisitos conceituais.

CONSIDERAÇÕES PESSOAIS

Thiago:

O projeto é bem coerente com a matéria, entretanto, o tamanho do projeto obstrui o estudo das outras matérias, dificultando o desenvolvimento das demais matérias.

Victor:

O projeto possibilita grande aprendizado, tanto sobre a orientação a objetos quanto sobre o desenvolvimento de software em geral, ainda que durante sua execução devido ao tamanho e complexidade gere bastante estresse.

DIVISÃO DO TRABALHO

A seguir se encontra a tabela com a divisão de trabalho:

Tabela 4. Lista de Atividades e Responsáveis.

Atividades.	**Responsáveis**
Compreensão de Requisitos	Victor e Thiago
Diagramas de Classes	Victor e Thiago
Programação em C++	Victor e Thiago em geral
Implementação de *Template/lista Entidade*	Mais Victor que Thiago
Implementação da Persistência dos Objetos...	Mais Victor que Thiago
Implementação do ranking	Victor e Thiago
Gerenciador De Colisões	Mais Thiago que Victor
Movimentos das entidades	Mais Thiago que Victor
Inimigos	Thiago e Victor
Menus	Mais Thiago que Victor
Jogador	Thiago e Victor
Fases	Thiago e Victor
Obstáculos	Thiago e Victor
Gerenciador gráfico	Mais Victor que Thiago
Escrita do Trabalho	Victor e Thiago
Revisão do Trabalho	Victor e Thiago

- Thiago trabalhou em 100% das atividades ou as realizando ou colaborando nelas efetivamente.
- Victor trabalhou em 100% das atividades ou as realizando ou colaborando nelas efetivamente.

AGRADECIMENTOS

Agradecemos aos monitores e aos colegas de turma pela ajuda durante o desenvolvimento do projeto.

REFERÊNCIAS CITADAS NO TEXTO

[1] SIMÃO, J. M. Site das Disciplina de Fundamentos de Programação 2, Curitiba – PR, Brasil, Acessado em 27/011/2022, às 22:15 -

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

REFERÊNCIAS UTILIZADAS NO DESENVOLVIMENTO

[A] MARCOS OLIVEIRA . terminalroot, Acessado em 17/11/2022, às 17:38 -
Como Criar um Game MENU com C++ e SFML

[B] TERMSPAR,C++/SFML 2D Game Development #1: Making Textboxes and Buttons
Acessado em 22/11/2022, às 23:27
https://www.youtube.com/watch?v=T31MoLJws4U&t=810

[C] Playlist do monitor Felipe Alvez Barboza
Tutorial Jogo SFML - YouTube
Acessado em 27/11/2022, às 22:17

[D] Canal do monitor Matheus Augusto Burda
Burda Canal - YouTube
Acessado em 22/11/2022, às 22:20