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

Adryan Castro Feres, Thales Gabriel Balbino dos Santos
adryancastro@alunos.utfpr.edu.br, thales.2020@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, que busca capacitar o aluno no desenvolvimento de software, exige a elaboração de um jogo em C++, visando a aplicação dos conceitos vistos no decorrer da disciplina. Sendo assim, foi previsto o desenvolvimento do jogo Sonho++. Um jogo de plataforma, desenvolvido em C++, em que o jogador é um mago que está dentro do mundo dos sonhos e tem o objetivo de neutralizar todos os monstros que possam vir a assombrar as pessoas no mundo real.

O planejamento foi realizado a partir do Diagrama de Classes em Linguagem de Modelagem Unificada (UML), possuindo como base um diagrama genérico fornecido. No desenvolvimento do jogo, foi utilizado os requisitos propostos textualmente, que incluem os conceitos básicos da POO, como classes, objetos, relações entre as classes, encapsulamento, herança, e conceitos mais avançados como polimorfismo, sobrecarga de métodos, sobrecarga de operador e gabaritos.

Após o desenvolvimento, testes, reuniões semanais com o professor e auxílio da monitoria, é possível considerar o jogo funcional e finalizado, com o comprimento parcial dos requisitos, salientando a evolução pessoal no decorrer da disciplina e o aprendizado nas relações que permeiam todo esse contexto de desenvolvimento.

Palavras-chave ou Expressões-chave: Desenvolvimento de um Jogo em C++, Trabalho Acadêmico Voltado a Aplicação de Conceitos da Programação Orientada à Objetos, Jogo de Plataforma.

Abstract - *This document presents the general aspects of the development of a platform game for the class of Programming techniques (Técnicas de programação). The development of the game was motivated by the need to better comprehend the aspects of the C++ programming language, software engineering and graphic libraries.*

Key-words or Key-expressions: *Academic Work Related to C++ Implementation, Platform game, POO*

INTRODUÇÃO

A disciplina de Técnicas de Programação tem o objetivo de capacitar o aluno no desenvolvimento e técnicas de modelagem de software a partir do paradigma da Programação Orientada a Objetos (POO). Um conceito que se difere bastante do modelo procedimental a qual se iniciam os alunos no contexto da computação. Deste modo, como uma forma de contextualizar e aplicar os conceitos vistos no decorrer do curso, foi previsto o desenvolvimento do jogo Sonho++, um jogo de plataforma que permite ao aluno a aplicação dos pilares da Programação Orientada à Objetos.

O desenvolvimento do Sonho++ partiu da compreensão dos requisitos propostos textualmente e abordados em sala. A partir disso, o planejamento e modelagem via diagrama de classe em

UML foi realizado, o qual permite uma melhor organização e entendimento geral da proposta prevista previamente, proporcionando condições de se iniciar corretamente a implementação em C++ do jogo de plataforma, somada ao acompanhamento recorrente do professor da disciplina no andamento do projeto. Com o decorrer do desenvolvimento, testes pertinentes fizeram parte do processo de evolução do projeto, tendo em vista a conferência do funcionamento e cumprimento dos requisitos propostos.

Através desse ciclo de desenvolvimento, a proposta prevista se tornou uma realidade, e as etapas que permearam esse processo, que serão detalhadas a seguir, permitiram a evolução pessoal e profissional dos envolvidos.

EXPLICAÇÃO DO JOGO EM SI

O jogo Sonho++ trata da breve jornada de um mago no submundo dos sonhos. Neste local sombrio e decrépito o herói luta contra criaturas sombrias que, se libertadas, causarão pesadelos em todos os seres vivos! O mago do Sonho++, não possui um nome, ele é também um ser desse lugar estranho, e ao contrário de outras representações fantasiosas, não lança bolas de fogo ou conjura seres rûnicos para combater em seu lugar. No Sonho++ o poder do mago é único: sua aura mágica! Ao tocar nas criaturas sombrias que espreitam no submundo dos sonhos o herói as destrói, isso porque sua magia é altamente danosa aos inimigos.

A jornada deste mago herói já dura milhares de anos, e, como era de se esperar, seus inimigos desenvolveram artimanhas para evitar sua aura poderosa: agora apenas locais específicos de seus corpos estão sujeitos ao poder heróico.

Os inimigos do feiticeiro são milhares, mas neste local em que ele se encontra há apenas três: o primeiro deles é o fantasma, um espírito vagante e zombeteiro. Esse pequeno ser aparentemente inofensivo é perverso: se movimenta aleatoriamente em busca de uma oportunidade de encontrar alguém e assustá-lo! Felizmente o fantasma tem pouca resistência e a mera aproximação do herói empoderado é suficiente para desintegrá-lo de uma vez por todas. Entretanto, caso o feiticeiro pule sobre sua cabeça, nada ocorrerá, pois ali ele é imune a aura heróica.

O segundo inimigo é a Hydra, uma estranha espécie de cogumelo. Essa criatura rûnica, além de perseguir o herói quando atingido, possui espinhos por todo o corpo, esses espinhos pontiagudos e envenenados causam dano alto caso o herói entre em contato, além disso o shamã enfeitiça os espinhos para que eles sejam imunes a aura do herói, a única maneira de derrotá-lo é pulando sobre sua cabeça, o singular local sem espinhos mortais.

O terceiro e último inimigo é o Ceifador. Este ser da escuridão é um dos piores inimigos do nosso herói. Ele lança bolas de fogo e nunca está sozinho, de tal maneira que um passo em falso e está tudo perdido. Felizmente seu corpo de sombras não é resistente, basta se aproximar para lhe causar dano, mas com muito cuidado, pois ele nunca para de lançar sua magia sobre o herói.

Já o cenário desses combates é simples: um local silencioso e escuro. Apenas plataformas cheias de monstros e com alguns obstáculos podem ser vistas. Algo inexplicável acontece sempre que o mago destrói todos os seus inimigos: num passe de mágica ele é teleportado a outro local, mas sempre para um espaço sombrio e com obstáculos, não necessariamente iguais aos obstáculos do lugar anterior. O mago espera um dia ser capaz de escapar de tamanha escuridão.

É sabido que o mago já encontrou todo tipo de obstáculo, mas neste momento o que ele mais encontra é uma espécie de caixa com aparência aterrorizante e espinhos com formatos estranhos, que o deixa mais lento e o ataca após um tempo, de maneira que todos os seus passos devem ser cuidadosos para evitar um grande acidente.

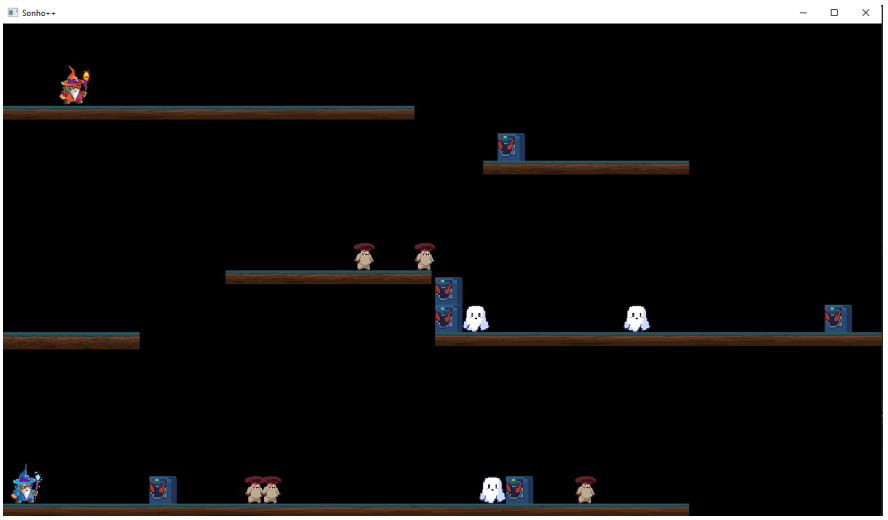

Imagem 5. Fase 1, exemplo de execução.

DESENVOLVIMENTO DO JOGO NA VERSÃO ORIENTADA A OBJETOS

A proposta do desenvolvimento de um jogo em C++, além de reforçar e aplicar os conceitos estudados, vem com o objetivo de familiarizar o aluno com o mercado de trabalho, por meio de algumas particularidades atreladas a este projeto, como por exemplo, reuniões obrigatórias semanais para acompanhar o desenvolvimento e uma tabela de requisitos requeridos pelo proponente, a luz do presenciado em ambientes profissionais de desenvolvimento.

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.	Classe menu com todas as características pertinentes, utilizando estados, via identificadores, para que os estados do jogo sejam realizados.
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 cujos objetos são agregados em jogo, podendo ser apenas um jogador, entretanto.
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 cumprido via classe Fase1 e Fase2, cujos objetos são agregados na classe jogo. O jogador neutraliza os inimigos através do contato e os inimigos causam dano no jogador pelo contato e projétil.
4	Ter pelo menos três tipos distintos de inimigos, cada qual com sua representação gráfica, sendo que ao menos um dos	Requisito previsto inicialmente e realizado.	Requisito cumprido via classe Hydra, Fantasma e Anjo, que por sua vez é o chefão e agrega um objeto da

	inimigos deve ser capaz de lançar projetil contra o(s) jogador(es) e um dos inimigos dever ser um 'Chefão'.		classe Projétil que o permite lançá-lo na direção do jogador.
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 via Classe Fase1 e Fase2, as quais agregam ao menos dois inimigos e criam um número aleatório de instâncias.
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 cumprido através da classe Plataforma, Caixa e Espinhos, classe essa que causa dano ao jogador em uma colisão.
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 cumprido através da classe Fase1 e Fase2 que agregam ao menos dois tipos de obstáculos e instancia um número aleatório desses objetos.
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 através da classe Plataforma e Caixa que não causam dano ao Jogador e permitem o mesmo subir sobre elas.
9	Gerenciar colisões entre jogador para com inimigos e seus projeteis, 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 via classe GerenciadorColisoes. Todas as entidades sofrem o efeito da gravidade através da função movGravidade presente na classe Entidades.
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 não realizado.	
Total de requisitos funcionais apropriadamente realizados. *(Cada tópico vale 10%, sendo que para ser contabilizado deve estar realizado efetivamente e não parcialmente)*			90%

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

A biblioteca gráfica escolhida foi a SFML, orientada a Objeto por natureza e programada inteiramente em C++, a qual fornece todos os elementos necessários para o desenvolvimento da aplicação, além do seu fácil uso, característica essencial para a finalização deste projeto na data prevista.

A luz do diagrama de classes, foi iniciado o desenvolvimento pela classe base, Ente, da qual descende a maioria das classes do jogo. Ente possui um método virtual puro redefinido por

cada classe herdada, o Executar, que contém as funcionalidades principais para o funcionamento previsto dos objetos em questão.

A partir disso, se fez necessário a criação de um gerenciador gráfico, responsável por gerenciar tudo o que se relaciona com a renderização e exibição dos elementos na janela. Só deve existir uma janela do jogo em tempo de execução, sendo assim, o gerenciador foi desenvolvido segundo o padrão de projeto Singleton, o qual garante a existência de apenas uma instância dessa classe. Deste modo, o gerenciador gráfico é agregado ao Ente, do qual deriva a classe Entidade, uma classe abstrata que contém todos os elementos em comum as entidades, como corpo, método de ataque, entre outras.

Todas as entidades do jogo precisam sofrer o efeito da gravidade. Nesse sentido foi desenvolvido o método movGravidade, que implementa essa força fundamental, através da integração de Euler. Para entender o funcionamento dessa função, é preciso considerar que o movimento das entidades no jogo se dá por um acréscimo de um determinado valor (velocidade) na posição atual em x ou y desse corpo em uma sequência de pequenos intervalos de tempo, que se dão pelo loop principal do jogo. Sendo assim, é possível a aplicação do método de Euler, o qual implica que a partir de uma sequência de pequenos intervalos de tempo de duração Δt e uma aceleração constante é possível definir a velocidade do corpo nesse instante, como visto na equação (1). A partir disso, a velocidade em y do corpo é atualizada a cada chamada da função, criando o efeito da gravidade em todas as entidades.

(1) $V_y = V_{atual} + a * \Delta t$

A classe Gerenciador de Colisões, verifica constantemente se há colisão entre as entidades, e chama a função reagirColisões de cada entidade envolvida para resolvê-la como desejar. Ainda, como forma de impedir que uma entidade “atravesse” a outra, existe a função corrigeColisões, a qual recebe o valor de colisão calculado na verificação e incrementa esse valor na posição de uma das entidades. Essa classe foi implementada com o auxílio dos vídeos do monitor Giovane Limas e Matheus Burda.

O pacote de classe entidades é composto por todas as figuras do jogo, sendo essas o pacote de obstáculos, personagens e o projétil. E, como forma de facilitar a renderização na tela e a verificação das colisões entre as entidades, são utilizadas duas listas encadeadas de entidades, uma lista de personagem e uma de obstáculos, criadas via Gabarito nas classes Lista e Elemento e parametrizadas na classe ListaEntidades.

Herdada da classe Entidade, e pertencente ao seu pacote, temos a classe Personagens. Todos os personagens possuem a capacidade de andar, porém, cada um tem a sua peculiaridade nessa questão, sendo assim, há um método virtual puro Mover, redefinido por cada personagem. Além do movimento, todo o personagem possui uma quantidade de vida, que por sua vez, pode ser diminuída através da redefinição do método atacar de cada entidade. A verificação constante dessa vida se dá pelo método verificaVida, o qual se identifica que o personagem morreu, seta seu atributo visível (presente na classe Ente) como false, impedindo com que o objeto execute e renderize.

Ainda, os personagens possuem a capacidade de pular, implementada através da equação de torricelli (6), que calcula a velocidade em y do personagem, contrária a gravidade.

(2) $V_y = -\sqrt{2 * a * \Delta s}$

Herdada da classe Personagens, temos a classe Inimigo e Jogador. A classe Jogador redefine as funções necessárias e contém um atributo que indica se o jogador é o primeiro ou segundo e dois atributos estáticos referentes à pontuação do jogador (pontuação_j1 e pontuação_j2), que são incrementadas a partir da redefinição da função atacar.

A classe Inimigo, é abstrata e contém um conjunto de funções importantes para as características de cada inimigo, todas relacionadas ao movimento, além de conhecer o jogador. Os inimigos possuem dois tipos de movimento, o aleatório e o que persegue o jogador. A função que implementa esse movimento aleatório é a movAleatorio que sorteia um número entre 0 e 1 a cada 1 segundo, o qual indica a direção do movimento do inimigo. Já a perseguição do jogador é um pouco mais complexa de ser implementada, tendo em vista a possibilidade de se jogar com dois jogadores.

Deste modo, a classe Inimigo possui uma função que verifica quantos jogadores estão jogando a fase, e a partir disso é decidido como vai ser a perseguição. No caso de só um jogador, o inimigo verifica se o mesmo está no seu raio de detecção e o persegue até chegar na mesma posição. No caso de dois jogadores, há uma função que verifica o jogador mais próximo e o persegue, seguindo a mesma lógica do caso anterior.

No jogo existem 3 inimigos, Hydra, fantasma e o ceifador. Esse último é o chefão, que persegue e atira um projétil no jogador quando entra no seu raio de detecção. O projétil por sua vez, é uma classe derivada de Entidade que se move, em uma trajetória parabólica, através da função atirar em sua definição, a qual sai do seu estado inicial (mesma posição do portador) até colidir com o jogador ou algum obstáculo, fazendo com que o mesmo volte ao seu estado inicial. Ainda no pacote das entidades, temos os obstáculos, plataforma, caixa e espinhos, todos derivados de uma classe abstrata Obstáculos, derivada de Entidade. A plataforma flutua no mapa, então utiliza um método que anula a gravidade através da multiplicação da velocidade em y por -1, gerando um “vetor contrário” a força da gravidade. O espinho tem uma funcionalidade interessante, deixa o jogador lento quando está sobre ele e o causa dano a cada período de tempo, definido pelo seu atributo countdown.

A classe principal do projeto é a classe Jogo, que é composta pelas duas fases, dois jogadores e os menus. Nesta classe está o loop principal, que é baseado em estados e realiza determinada tarefa a partir do estado em que se está.

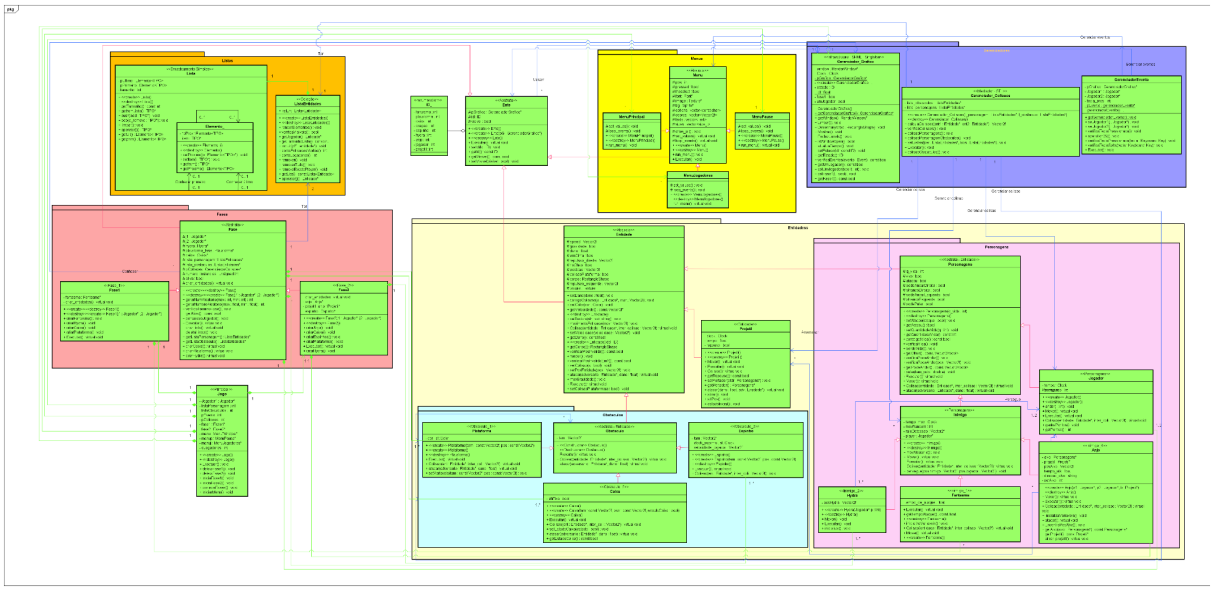

Figura 2. Diagrama de Classes de base em UML.

TABELA DE CONCEITOS UTILIZADOS E NÃO UTILIZADOS

Segue tabelado os conceitos utilizados no desenvolvimento do Sonho++.

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 por exemplo nas classes Jogo..
1.2	- Métodos (com retorno *const* e parâmetro *const*). & - Construtores (sem/com parâmetros) e destrutores	Sim	Construtores (sem/com parâmetros) estão em todos os .h e .cpp, como por exemplo na classe Jogo e Fase. Métodos com retorno e parâmetro const estão na maioria dos .h e .cpp, como na classe Entidade
1.3	- Classe Principal.	Sim	A classe principal é a classe Jogo
1.4	- Divisão em .h e .cpp.	Sim	Quase todas as classes são divididas em .h e .cpp, como por exemplo na classe Espinho
2	**Relações de:**		
2.1	- Associação direcional. & - Associação bidirecional.	Sim	Presente em todo o código, como por exemplo o inimigo associado ao jogador
2.2	- Agregação via associação. & - Agregação propriamente dita.	Sim	Presente em várias classes do jogo, como por exemplo a classe Anjo que agrega via associação a classe Projétil
2.3	- Herança elementar. & - Herança em diversos níveis.	Sim	Por todo o projeto, um exemplo são as classes MenuPrincipal, MenuJogadores e MenuPause herdadas da classe Menu
2.4	- Herança múltipla.	Não	
3	**Ponteiros, generalizações e exceções**		
3.1	- Operador *this* para fins de relacionamento bidirecional.	Sim	Em quase todas as classes. Exemplos na classe GerenciadorEventos
3.2	- Alocação de memória (*new* & *delete*).	Sim	Em várias classes. De maneira farta na classe Menu, Fase1 e Fase20
3.3	- Gabaritos/*Templates* criada/adaptados pelos autores (*e.g.*, Listas Encadeadas via *Templates*).	Sim	Gabarito de Listas Encadeadas. Na classe Lista.h e Elemento.h a qual compõe a ListaEncadeada
3.4	- Uso de Tratamento de Exceções (*try catch*).	**Sim**	Utilizado para tratar erros no carregamento de texturas, exemplo no método setTextura na classe Entidade
4	**Sobrecarga de:**		
4.1	- Construtoras e Métodos.	Sim	Sobrecarga de contrutores em quase todas as classes. Sobrecarga de métodos em várias classes, por exemplo na classe GerenciadorGrafico
4.2	- Operadores (2 tipos de operadores pelo menos – Quais?).	Não	Sobrecarga apenas do operador [].
---	**Persistência de Objetos (via arquivo de texto ou binário)**		
4.3	- Persistência de Objetos.	Não	. . .

4.4	- Persistência de Relacionamento de Objetos.	Não	. . .
5	**Virtualidade:**		
5.1	- Métodos Virtuais Usuais.	Sim	Destrutoras virtuais, como na classe Menu e métodos redefinidos, como o atacar no Entidade
5.2	- Polimorfismo.	Sim	Fartamente em quase todas as classes.
5.3	- Métodos Virtuais Puros / Classes Abstratas.	Sim	Métodos virtuais puros em quase todo o projeto. Exemplo de classe abstrata há o Ente e nesta há funções virtuais puras
5.4	- Coesão/Desacoplamento efetiva e intensa com o apoio de padrões de projeto.	Sim	Código coeso e desacoplado.
6	**Organizadores e Estáticos**		
6.1	- Espaço de Nomes (*Namespace*) criada pelos autores.	Sim	Implementado na classe Menu, com o namespace Menus.
6.2	- Classes aninhadas (*Nested*) criada pelos autores.	Não	
6.3	- Atributos estáticos e métodos estáticos.	Sim	No gerenciador gráfico há método e atributo estático.
6.4	- Uso extensivo de constante (*const*) parâmetro, retorno, método...	Sim	Por todo código. A exemplo nos métodos da Ente.
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	Uso o vector em vários momentos, como por exemplo na classe Menu. Uso da string para facilitar o uso dos textos, como por exemplo para lidar com caminhos de arquivos.
7.2	- Pilha, Fila, Bifila, Fila de Prioridade, Conjunto, Multi-Conjunto, Mapa **OU** Multi-Mapa.	Sim	Uso pilha na implementação do ranking dos jogadores, em Jogo.cpp
---	**Programação concorrente**		
7.3	- *Threads* (Linhas de Execução) no âmbito da Orientação a Objetos, utilizando Posix, C-Run-Time **OU** Win32API ou afins.	*Não*	
7.4	- *Threads* (Linhas de Execução) no âmbito da Orientação a Objetos com uso de Mutex, Semáforos, **OU** Troca de mensagens.	*Não*	
8	**Biblioteca Gráfica / Visual**		
8.1	- Funcionalidades Elementares. **&** - Funcionalidades Avançadas como: ● tratamento de colisões ● duplo *buffer*	Sim	*Tratamento de colisões* *Gerenciador Gráfico*
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).	Sim	*Gerenciador de Eventos*
---	**Interdisciplinaridades via utilização de Conceitos de Matemática Contínua e/ou Física.**		
8.3	- Ensino Médio Efetivamente.	*Sim*	Equação de Torricelli (MUV), pular dos personagens na classe Personagens

8.4	- Ensino Superior Efetivamente.	*Sim*	Manipulação da integração de Euler na classe Entidade
9	**Engenharia de Software**		
9.1	- Compreensão, melhoria e rastreabilidade de cumprimento de requisitos. &	Sim	Intensivamente desde o início do projeto.
9.2	- Diagrama de Classes em *UML*.	Sim	Intensivamente desde o início do projeto.
9.3	- Uso efetivo e intensivo de padrões de projeto *GOF*, *i.e.*, mais de 5 padrões.	Não	Apenas o Singleton no gerenciadorGrafico e gerenciadorEventos
9.4	- Testes à luz da Tabela de Requisitos e do Diagrama de Classes.	Sim	Intensivamente desde o início 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*).	*Sim*	Uso do github desde o início do projeto para controle de versões e google drive para backup.
10.2	- Reuniões com o professor para acompanhamento do andamento do projeto.	*Sim*	1 reunião: 03/11/2022 as 11 horas. 2 reunião: 10/11/2022 as 11 horas. 3 reunião: 17/11/2022 as 11 horas e 4 reunião: 24/11/2022 as 11 horas.
10.3	- Reuniões com monitor da disciplina para acompanhamento do andamento do projeto.	*Sim*	27/10 - Thiago Aguiar (19:30) 09/11 - Alexei (15:30) 17/11 - Ian (11:45) [Chat] 22/11 - Thiago Henrique (11:50) 23/11 - Alexei (16:35) 24/11 - Geovane (11:45) [Chat] 25/11 - Thiago Aguiar (10:55) 25/11 - Alexei (14:40) 25/11 - Alexei (15:20) 25/11 - Thiago Aguiar (20:00)
10.4	- Revisão do trabalho escrito de outra equipe e vice-versa.	*Sim*	Frederico e Hyon
Total de conceitos apropriadamente utilizados.			**80%** (oitenta por cento).

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

Abaixo uma tabela detalhando a razão da utilização de conceitos.

NO.	**Conceitos**	**Utilizados**
1.1	- Classes, objetos. & - Atributos (privados), variáveis e constantes. & - Métodos (com e sem retorno).	Classes, objetos, atributos e variáveis são conceitos básicos da programação orientada a objetos. Além de que, atributos privados são importantes pois promovem o encapsulamento
1.2	Métodos (com retorno const e parâmetro const). & - Construtores (sem/com parâmetros) e destrutores	A constância dos métodos, retornos e parâmetros evitam mudanças não previstas no código. Construtores sem/com parâmetros e destrutores são necessárias para decidir especificamente como vai se iniciar e destruir a classe em questão
1.3	- Classe Principal.	“Purismo” em relação à Programação Orientada a Objetos

1.4	- Divisão em .h e .cpp.	Melhor compreensão e organização do sistema
2.1	- Associação direcional. & - Associação bidirecional.	Utilizada pois algumas classes possuem funcionalidades que se torna pertinente conhecer outras
2.2	- Agregação via associação. & - Agregação propriamente dita.	Pois algumas classes só irão ter o funcionamento previsto se conterem instâncias de outras classes.
2.3	- Herança elementar. & - Herança em diversos níveis.	Utilizada para evitar repetição de código e melhorar a coesão
3.1	- Operador this para fins de relacionamento bidirecional.	Utilizado por uma questão de clareza. Também quando um objeto precisava passar 'a si mesmo' por parâmetro.
3.2	- Alocação de memória (new & delete).	A memória teve de ser alocada para instâncias de classes. Depois de alocadas foram deletadas.
3.3	Gabaritos/Templates criada/adaptados pelos autores (e.g., Listas Encadeadas via Templates).	Necessário para evitar repetição de código em algo que pode ser replicado várias vezes.
3.4	- Uso de Tratamento de Exceções (try catch).	Utilizado para tratar erros no carregamento de texturas, exemplo no método setTextura na classe Entidade
4.1	- Construtoras e Métodos.	São conceitos básicos de POO. Impossível não utilizar.
5.1	- Métodos Virtuais Usuais.	Importante para especializar certas funcionalidades
5.2	- Polimorfismo.	Importante para um comportamento dinâmico de algumas classes especializadas.
5.3	- Métodos Virtuais Puros / Classes Abstratas.	Classes abstratas utilizadas de maneira semelhante a interfaces. Métodos virtuais puros seguem do uso.
5.4	- Coesão/Desacoplamento efetiva e intensa com o apoio de padrões de projeto.	Facilita a identificação de erros e a reutilização de seções específicas do programa.
6.1	- Espaço de Nomes (Namespace) criada pelos autores.	Namespaces criados por razões organizacionais.
6.3	- Atributos estáticos e métodos estáticos.	Utilizado para continuar com um único endereço de um atributo em tempo de execução.
6.4	- Uso extensivo de constante (const) parâmetro, retorno, método...	Se retorna algo constante então usa const. Se é algo que não deve ser alterado também usa const.
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)	A classe string facilita em parâmetros de métodos e atributos para controle de algumas funcionalidades e o vector é utilizado pela sua facilidade na implementação.

7.2	- Pilha, Fila, Bifila, Fila de Prioridade, Conjunto, Multi-Conjunto, Mapa **OU** Multi-Mapa.	Utilizado Pilha para realizar o ranking dos jogadores pela facilidade e por imprimir na tela primeiro a mais recente pontuação.
8.1	- Funcionalidades Elementares. & - Funcionalidades Avançadas como: tratamento de colisões duplo buffer	Gerenciador gráfico utilizado para gerir todas as tarefas relativas a janela e o tratamento de colisões utilizado para gerir o contato entre as entidades do jogo.
8.2	- Programação orientada e evento efetiva em algum ambiente gráfico. **OU** - *RAD – Rapid Application Development*	Gerenciador de eventos utilizado para gerir alguns dos inputs que ocorrem no momento da execução da fase, como o comando para o jogador andar.
8.3	- Ensino Médio Efetivamente	Utilizado para calcular a velocidade do pulo dos personagens.
8.4	- Ensino Superior Efetivamente.	Utilizado pois na execução do jogo temos vários instantes de tempo que permitem a aplicação dessa integração, ainda mais com a aceleração constante.
9.1	- Compreensão, melhoria e rastreabilidade de cumprimento de requisitos. &	Processos realizados intensivamente para conseguir cumprir os requisitos e utilizar os conceitos definidos.
9.2	- Diagrama de Classes em UML.	Utilizado intensivamente para modelar o jogo.
9.4	- Testes à luz da Tabela de Requisitos e do Diagrama de Classes.	Organização e cumprimento de requisitos.
10.1	- Controle de versão de modelos e códigos automatizado. & - Uso de alguma forma de cópia de segurança	Facilita a organização do trabalho como um todo, sincronização da dupla e resolução de problemas.
10.2	- Reuniões com o professor para acompanhamento do andamento do projeto.	Instruções gerais sobre a realização do trabalho. Dicas e boas práticas.
10.3	- Reuniões com monitor da disciplina para acompanhamento do andamento do projeto.	Tirar dúvidas, esclarecer dificuldades no código, aprender com quem já fez.
10.4	- Revisão do trabalho escrito de outra equipe e vice-versa.	Importante para a conferência de possíveis informações equivocadas ou fora do padrão proposto.

Tabela 3. Lista de Justificativas para Conceitos Utilizados.

REFLEXÃO COMPARATIVA ENTRE DESENVOLVIMENTOS

A programação procedimental é semelhante a uma receita de bolo, ou seja, um passo a passo. Já a orientada a objetos é uma maneira de modelar tudo o que há no mundo, inclusive utilizando procedimentos, se necessário.

DISCUSSÃO E CONCLUSÕES

O desenvolvimento deste trabalho se deu em tempo exíguo. Apesar disso, os resultados obtidos foram satisfatórios. Conquanto satisfatórios, Infelizmente não foi possível obter um jogo completo, no sentido de um jogo com salvamento efetivo, animações e grande variedade de inimigos. Entretanto, parte majoritária dos requisitos avaliativos foram devidamente realizados.

Tem-se em mãos um jogo com menus, personagens texturizados, gerenciamento de colisões, gerenciamento de projéteis e mais tarefas relativas a um jogo básico de plataforma. Ponto principal deste projeto foi a observação das características únicas da programação orientada a objetos e também o descobrimento de um novo espaço no vasto campo da programação: o desenvolvimento de jogos.

Conclui-se a partir de todos esses aspectos que a labuta, apesar de dura e extenuante, traz consigo aprendizados impossíveis de obter de maneira distinta.

DIVISÃO DO TRABALHO

A divisão do trabalho se deu de maneira orgânica, humana e igualitária, respeitando as diferenças inerentes a cada membro do grupo. Segue tabelado de que maneira e por quem as tarefas foram realizadas.

Atividades		**Responsáveis**
Compreensão de Requisitos	Implementação do Pacote Listas	Adryan e Thales
Diagrama de Classes	Implementação do Pacote Fases	Adryan e Thales
Programação em C++	Implementação do Pacote Personagens	Adryan e Thales
Implementação do Pacote Menu	Implementação do Pacote Gerenciadores	Adryan e Thales
Implementação do Pacote Obstáculos	Implementação Geral	Adryan e Thales
Revisão geral	Escrita do Trabalho	Adryan e Thales

Tabela 4. Lista de Atividades e Responsáveis.

REFERÊNCIAS UTILIZADAS NO DESENVOLVIMENTO

[1] LIMAS, GIOVANE. Playlist do Youtube Criando um Jogo em C++ do ZERO, Curitiba - PR, Brasil - https://youtube.com/playlist?list=PLR17O9xbTbIBBoL3lli44N8LdZVvg-_uZ

[2] BURDA, MATHEUS. Playlist do YouTube Tutorial Jogo SFML, Curitiba – PR, Brasil - https://www.youtube.com/playlist?list=PLSPev71NbUEBIQlT2QCd-gN6l_mNVw1cJ

[3] ROOT, TERMINAL. Como Criar um Game MENU com C++ e SFML, Curitiba – PR, Brasil - https://www.youtube.com/watch?v=h8-Q4eu3Qt4&t=1169s

[4] Roteiro de Integração Numérica (Método de Euler), Curitiba – PR, Brasil - http://www.fep.if.usp.br/~fisfoto/guias/integracao_numerica_2015.pdf