TRABALHO PARA A DISCIPLINA DE TÉCNICAS DE PROGRAMAÇÃO DO CURSO DE SISTEMAS DE INFORMAÇÃO DA UTFPR: *DINO++ - RELATÓRIO DO PROJETO*

Guilherme Aguilar de Oliveira, Tiago Gonçalves da Silva
guilhermeoliveira.2019@alunos.utfpr.edu.br, tiagosilva.2019@alunos.utfpr.edu.br

Disciplina: **Técnicas de Programação – CSE20** / S73 – Prof. Dr. Jean M. Simão
Departamento Acadêmico de Informática – DAINF - Campus de Curitiba
Curso Bacharelado em: 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 tem como trabalho final o desenvolvimento de um software, na forma de um jogo de plataforma, para que haja a aplicação dos diversos conceitos relacionados à engenharia de *software*, sobretudo orientação a objeto em C++, vistos ao longo do semestre. Com isso em mente foi escolhido o jogo Dino++, no qual o jogador enfrenta inimigos e obstáculos em um cenário pré-histórico ao longo de duas fases. Como guia para o desenvolvimento do trabalho foram considerados os requisitos levantados textualmente, além de um Diagrama de Classes em Linguagem de Modelagem Unificada (*Unified Modeling Language - UML*) fornecido como base para a modelagem (análise e projeto) do sistema. Em seguida realizou-se o desenvolvimento do projeto, em linguagem de programação C++ com suporte da biblioteca gráfica *Simple and Fast Multimedia Library* (SFML), que conectou os conceitos básicos de Orientação a Objetos como Classes, Relacionamentos e Heranças, com os mais avançados, Classes Abstratas, Polimorfismo, Sobrecarga de Operadores e Métodos. Durante e após a implementação, os testes de uso do jogo foram feitos pelos desenvolvedores para demonstrar a funcionalidade e aplicação dos conceitos previamente requisitados, garantindo assim o aprendizado dos conceitos da ementa da disciplina inicialmente visado.

Palavras-chave ou Expressões-chave: Trabalho Acadêmico com Implementação em C++, Jogo de Plataforma Orientado a Objeto, Programação Orientada a Objeto com Biblioteca Gráfica, Trabalho Final da Disciplina de Técnicas de Programação.

Abstract - *The subject of Técnicas de Programação has as its final work the development of a software, taking the form of a platform game, in order to apply the several topics related to software engineering, specially those related to object-oriented programming in C++ language, seen through the semester. With this in mind the game chosen was Dino++, in which the player faces various enemies and obstacles in a pre-historic scenario through two levels. Acting as guides for the development of the game were considered the requirements raised textually, as well as a Class Diagram in the Unified Modeling Language (UML) provided as the base for the modeling (analysis and project) of the system. After that the actual development of the project took place, in the C++ programming language with support of the Simple and Fast Multimedia Library (SFML), that connected the base concepts of object orientation, like Classes, Relationships and Inheritance, with the more advanced ones, Abstract Classes, Polimorfism, Operators and Methods Overloading. During and after the implementation, tests of the usage of the game were made by the developers to demonstrate functionality and correct applications of the topics previously required, ensuring this way the learning of the concepts of the discipline menu.*

Key-words or Key-expressions Academic Work with Implementation in C++, Object-oriented Platform Game, Object-oriented Programming with Graphic Library Support, Final Work of the Técnicas de Programação Subject.

INTRODUÇÃO

A produção de um projeto orientado a objeto em C++ é apresentada como trabalho final da disciplina de Técnicas de Programação, com o intuito de aplicar e reforçar os conceitos de orientação a objeto contemplados durante o semestre.

Para realizar tal tarefa foi inicialmente acordado que o programa deve ter a forma de um jogo, aos moldes dos de plataforma da era 16 bits e usando uma biblioteca gráfica, de forma a estimular a capacidade do aluno de obter conhecimento além da tutela do professor.

O método aplicado é um clássico ciclo de engenharia de software, em uma forma simplificada, em que todas as etapas, desde a modelagem (análise e projeto) na forma de diagrama UML, até a implementação e testes na linguagem C++ orientada a objeto.

O jogo Dino++ se baseia no conceito de jogos de plataforma, e em suas funcionalidades básicas. O jogador assume o papel de até dois dinossauros que, devem enfrentar diversos obstáculos, inimigos e desafios do ambiente, para chegar ao fim de duas fases, montanha e floresta, que contextualizam o ambiente em um cenário pré-histórico.

EXPLICAÇÃO DO JOGO EM SI

Ao iniciar o programa, é exibido o menu principal de opções, no qual o usuário pode escolher jogar com um ou dois jogadores, carregar uma jogada anterior, exibir o ranking ou sair. A navegação das opções é feita com as setas do teclado e a seleção é feita com a tecla ENTER.

Figura 1. Menu Principal.

Feita a seleção no menu principal, a fase montanha é iniciada. A jogabilidade segue os clássicos de plataforma da era 16 bits, em que o jogador, enquanto em cima de uma plataforma, tem livre mobilidade na horizontal, pode pular e é afetado pela aceleração da gravidade. Como forma de ataque, ele dispara um projétil de fogo que descreve movimento retilíneo e tem duração de meio segundo.

Compondo o cenário de jogo, tem-se diversos dinossauros inimigos. Sendo eles: Andino, inimigo comum que possui movimentação horizontal padrão e pula aleatoriamente; Atiradino, que não se movimenta e atira projéteis de fogo para a esquerda; Chefedino, que representa o desafio final da fase, tem tamanho e velocidade maiores que o Andino e mais

vidas que os outros inimigos comuns.

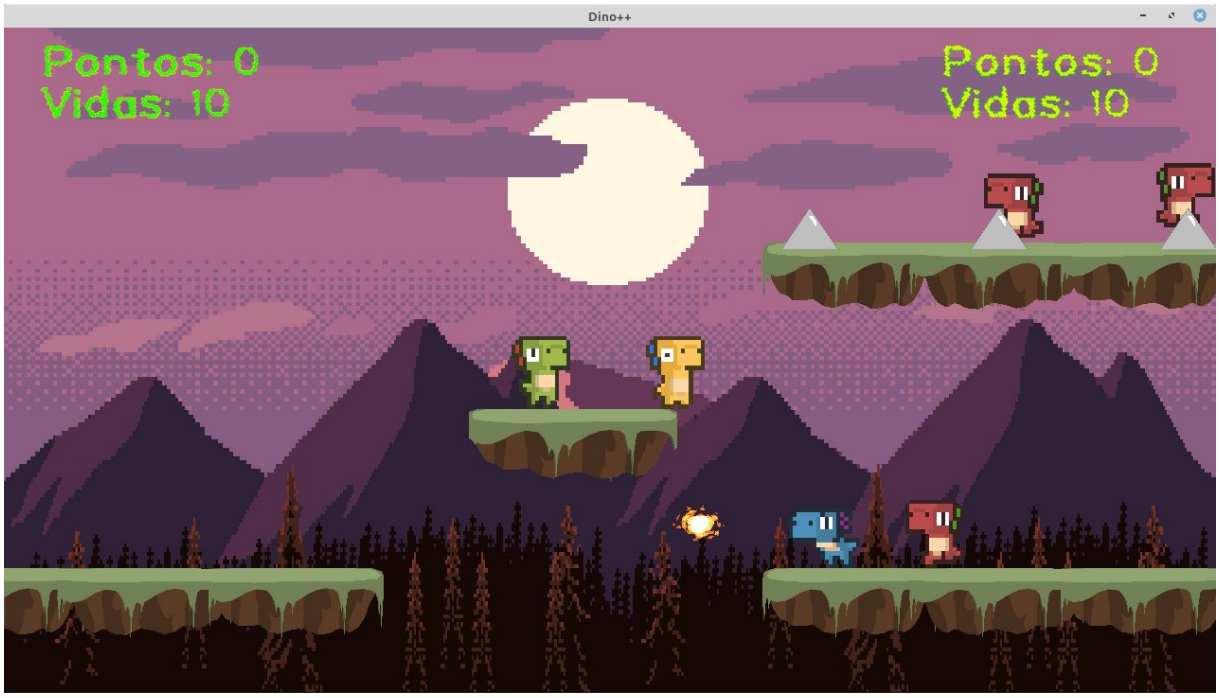

Figura 2. Fase Montanha com os dois jogadores.

Como parte adicional do cenário, as fases agregam tipos de obstáculos, que diferem na maneira que interagem com o jogador: espinhos, matam o jogador ao contato e aparecem em ambas as fases; Pedras, que podem ser movidas, sofrem ação da gravidade e são exclusivas da montanha; Galhos, que são incinerados quando entram em contato com o projétil disparado pelo jogador.

Diferente de outras entidades, inimigos possuem vidas, sendo necessários múltiplos acertos de projétil do jogador para abatê-los. De maneira análoga, quando jogadores sofrem dano, seja por contato com inimigo ou por ameaças do ambiente, estes perdem vida e são reposicionados na fase, de forma que se algum dos dois ficar sem vida, o jogo acaba e volta para o menu principal.

Em qualquer momento, o jogador pode apertar a tecla ESC para adentrar o menu de pause, acessando assim as opções de: voltar para a fase, salvar a jogada atual, verificar o ranking, carregar uma jogada anterior, ou voltar ao menu principal. Após chegar à última plataforma da montanha, o jogador é levado à fase floresta, que por sua vez ao ser completada, o jogo é finalizado, então o usuário vai para a tela de registro, na qual digita seu nome e sua pontuação é salva no ranking geral dos campeões.

DESENVOLVIMENTO DO JOGO NA VERSÃO ORIENTADA A OBJETOS

Durante o desenvolvimento do projeto, foram considerados como base os requisitos funcionais listados na Tabela 1 e o diagrama de classes em UML fornecido pelo professor[1]. O diagrama foi desenvolvido, com a supervisão do professor e do monitor da disciplina, até chegar em sua versão final na Figura 3. Tal modelo possibilita desta forma, ter uma visão clara dos objetivos e organização geral do trabalho, bem como os caminhos a se seguir na etapa de implementação do código.

Como fundamento do projeto tem-se o agrupamento de classes chamado Controladoras, no qual tem em seu cerne a classe Jogo, que é a gerenciadora principal do sistema. Para a

execução das diversas funcionalidades, a classe principal utiliza o padrão de projeto *State*, agregando desta forma uma classe PilhaEstados, que organiza as derivadas da classe abstrata Estado e garante o polimorfismo, uma vez que a classe Principal chama a função executar do estado mais acima da pilha, os menus e as fases, e tem como núcleo uma pilha padrão da Biblioteca Padrão de Gabaritos (Standard Template Library - STL).

As principais funções das fases são realizar a construção do cenário e gerenciar interações entre as entidades que o compõem. Para realizar a primeira tarefa foi utilizado o padrão de projeto *Factory Method*, de forma que o estado mais atual da pilha chama a construção da fase e de seus componentes, tal construção pode ser feita de duas formas, criando uma novo jogo do início ou começando a partir de um anterior.

Ao começar uma nova jogada, um arquivo de plataformas é lido e, em cima de cada uma delas, é instanciado um inimigo ou obstáculo de tipo aleatório específico para a determinada fase que se está construindo. Caso o usuário opte por iniciar a partir de um jogo anterior, são lidos arquivos de todas as entidades previamente instanciadas, bem como os diferentes atributos do jogador, como quantidade de pontos e vidas.

A fim de realizar as relações entre os elementos, toda fase agrega uma Colisora[2] e uma lista de entidades, classe que tem como cerne uma lista encadeada de entidades, de forma que a primeira utiliza a segunda para realizar colisões entre os objetos, técnica que só é possível devido à especialização de atributos de controle (podeMorrer, podeMatar e empurrao), feita de acordo com as necessidades de cada classe derivada de entidade. O modelo de detecção de colisão adotado foi o AABB[2].

Com o intuito de garantir maior variedade e exploração de conceitos, foram criados diversos tipos de inimigos, obstáculos e plataformas para compor o cenário, que a fim de garantir maior coesão, realizam polimorfismo ao especializarem as funções de imprimir e executar da classe abstrata Entidade, bem como os atributos específicos para o gerenciamento de colisões, possibilitando o tratamento de forma genérica pelas classes controladoras da fase.

Pelo fato de ter sido usada uma biblioteca gráfica e para garantir maior desacoplamento, todas as entidades agregam as classes de apoio CorpoGráfico e Animadora[3], que tratam de tudo relacionado à representação gráfica de texturas e animações. De maneira análoga e como parte da infraestrutura geral do projeto tem-se a classe GerenciadorGráfico, conhecida por derivadas de entidade e estados, que gerencia estruturas de base como janela, fundo e câmera.

Além disso vale ressaltar que cada um dos diferentes tipos de personagens e obstáculos contemplam conceitos interdisciplinares anteriormente estudados. Exemplos de aplicação de conceito são: Andinos capazes de pular e desta forma serem afetados pela aceleração da gravidade, além de realizar movimento parabólico ao pular; Atiradinos que não andam, mas atiram projéteis que realizam movimento uniformemente variado com velocidade decrescente; e pedras, que diminuem a velocidade do jogador ao serem empurradas, caracterizando conceitos de atrito.

A fim de realizar maior exploração de conceitos, a técnica *multithreading* foi implantada no inimigo Atiradino, para tal foram criadas as classes PThread, que provém a infraestrutura geral de threads usando Posix Thread, e AtiradinoThread que deriva do inimigo comum e da classe de infraestrutura. Desta forma os inimigos derivados de Thread atiram, controlados por mutex, um a cada três segundos. A implementação feita das threads é uma adaptação dos slides do professor.

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 base Menu e suas respectivas classes especializadas MenuPrincipal e MenuPause.
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 via classe Jogador e suas derivadas Guigo e Titi cujos objetos são agregados nas duas fases.
3	Disponibilizar ao menos duas fases que podem ser jogadas sequencialmente ou selecionadas.	Requisito previsto inicialmente e realizado.	Requisito cumprido via classe Fase e suas derivadas Montanha e Floresta de forma que podem ser jogadas sequencialmente.
4	Ter três tipos distintos de inimigos (o que pode incluir 'Chefão', vide abaixo), sendo que pelo menos um dos inimigos deve ser capaz de lançar projetil contra o(s) jogador(es).	Requisito previsto inicialmente e realizado.	Requisito cumprido via classe Inimigo e suas derivadas Andino, Atiradino, AtiradinoThread e ChefeDino sendo que os três últimos atiram projéteis no(s) jogador(es).
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 via classe FabricaFase e suas derivadas FabricaMontanha, FabricaFloresta, sendo que essas fábricas instanciam aleatoriamente os inimigos nas fases, sendo pelo menos 5 inimigos de cada tipo.
6	Ter inimigo "Chefão" na última fase	Requisito previsto inicialmente e realizado.	Requisito cumprido via classe ChefeDino que é instanciada na fase floresta.
7	Ter três tipos de obstáculos.	Requisito previsto inicialmente e realizado.	Requisito cumprido via classe Obstaculo e suas derivadas Espinho, Pedra e Galho.
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 via classe FabricaFase e suas derivadas FabricaMontanha, FabricaFloresta, sendo que essas fábricas instanciam aleatoriamente os obstaculos nas fases, sendo pelo menos 5 obstaculos de cada tipo.
9	Ter representação gráfica de cada instância.	Requisito previsto inicialmente e realizado.	Requisito cumprido via classe Corpo_Grafico que fornece texturas para classes derivadas de Entidade, e classe Fundo

			para as Classes derivadas de Estado.
10	Ter em cada fase um cenário de jogo com os obstáculos.	Requisito previsto inicialmente e realizado.	Requisito cumprido via classe Plataforma que molda o cenário da fase. Aleatoriamente em cima de cada plataforma, podem ser instanciados inimigos e/ou obstaculos.
11	Gerenciar colisões entre jogador e inimigos, bem como seus projeteis (em havendo).	Requisito previsto inicialmente e realizado.	Requisito cumprido via classe Colisora que gerencia as colisões entre as entidades da fase.
12	Gerenciar colisões entre jogador e obstáculos.	Requisito previsto inicialmente e realizado.	Requisito cumprido via classe Colisora que gerencia as colisões entre as entidades da fase.
13	Permitir cadastrar/salvar dados do usuário, manter pontuação durante jogo, salvar pontuação e gerar lista de pontuação (*ranking*).	Requisito previsto inicialmente e realizado.	Requisito cumprido via classes derivadas de Estados. Os Menus realizam o cadastro do usuário e geram o ranking. Já as Fases gerenciam a pontuação entre os níveis.
14	Permitir Pausar o Jogo	Requisito previsto inicialmente e realizado	Requisito cumprido via classe MenuPause.
15	Permitir Salvar Jogada.	Requisito previsto inicialmente e realizado	Requisito cumprido via classe Persistidora que é chamada por classes derivadas de Estados para salvar jogada.

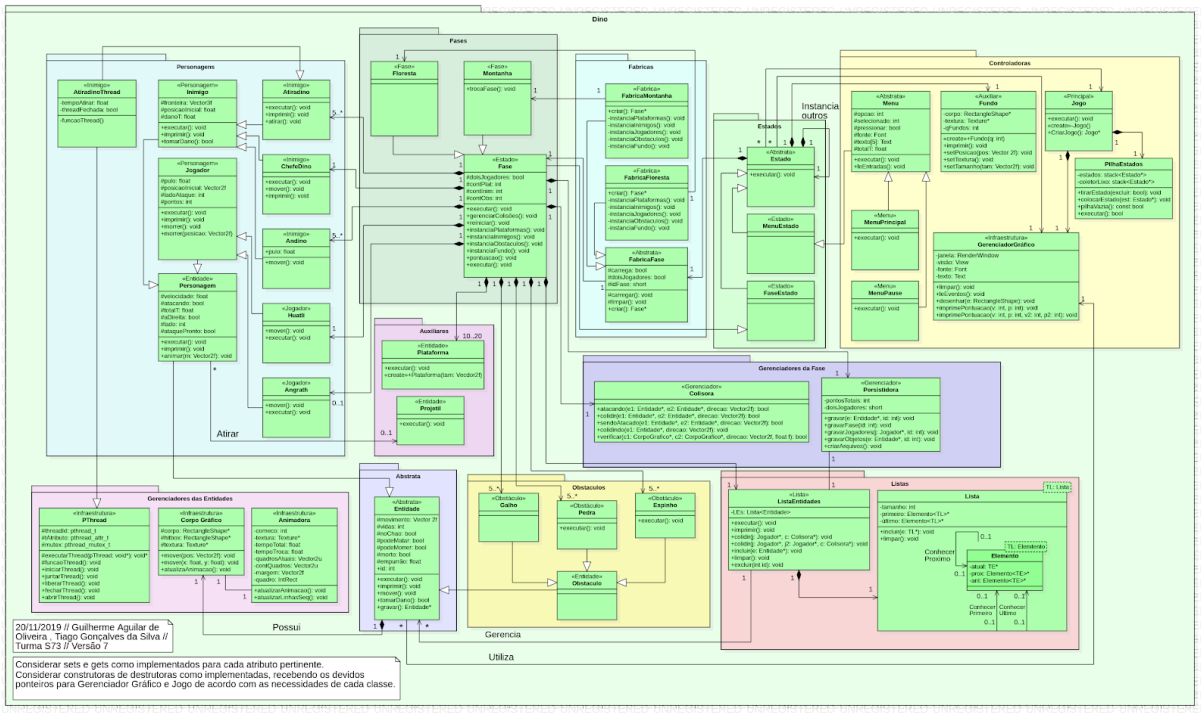

Figura 3. Diagrama de Classes do Projeto em UML.

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. & - Atributos (privados), variáveis e constantes. & - Métodos (com e sem retorno).	Sim	Todos .h e .cpp
	- Métodos (com retorno *const* e parâmetro *const*). & - Construtores (sem/com parâmetros) e destrutores	Sim	Todos .h e .cpp
	- Classe Principal.	Sim	Main.cpp & Jogo.h/.cpp
	- Divisão em .h e .cpp.	Sim	No desenvolvimento como um todo.
2	**Relações de:**		
	- Associação direcional. & - Associação bidirecional.	Sim	No desenvolvimento como um todo.
	- Agregação via associação. & - Agregação propriamente dita.	Sim	No desenvolvimento como um todo.
	- Herança elementar. & - Herança em diversos níveis.	Sim	No desenvolvimento como um todo.
	- Herança múltipla.	Sim	AtiradinoThread.h
3	**Ponteiros, generalizações e exceções**		
	- Operador *this*.	Sim	No desenvolvimento como um todo. Ex Corpo_Grafico
	- Alocação de memória (*new* & *delete*).	Sim	No desenvolvimento como um todo. Ex FabricaFase.cpp

		- Gabaritos/*Templates* criada/adaptados pelos autores (e.g. Listas Encadeadas via *Templates*).	Sim	Lista.h.
		- Uso de Tratamento de Exceções (*try catch*).	Sim	Persistidora.cpp.
	4	**Sobrecarga de:**		
		- Construtoras e Métodos.	Sim	No desenvolvimento do projeto como um todo.
		- Operadores (2 tipos de operadores pelo menos).	Sim	Corpo_Grafico.h & PilhaEstados.h.
		Persistência de Objetos (via arquivo de texto ou binário)		
		- Persistência de Objetos.	Sim	Persistidora.cpp, FabricaFase.cpp.
		- Persistência de Relacionamento de Objetos.	Sim	Persistidora.cpp.
	5	**Virtualidade:**		
		- Métodos Virtuais.	Sim	No desenvolvimento do projeto como um todo.
		- Polimorfismo	Sim	No desenvolvimento do projeto como um todo.
		- Métodos Virtuais Puros / Classes Abstratas	Sim	No desenvolvimento do projeto como um todo.
		- Coesão e Desacoplamento	Sim	No desenvolvimento como um todo.
	6	**Organizadores e Estáticos**		
		- Espaço de Nomes (*Namespace*) criada pelos autores.	Sim	Todos os .h e .cpp
		- Classes aninhadas (*Nested*) criada pelos autores.	Sim	Lista.h.
		- Atributos estáticos e métodos estáticos.	Sim	PThread.h, Jogo.h, AtiradinoThread.h e Plataforma.h.
		- Uso extensivo de constante (*const*) parâmetro, retorno, método...	Sim	No desenvolvimento do projeto como um todo.
	7	**Standard Template Library *(STL)* e String OO**		
		- A classe Pré-definida *String* ou equivalente. & - *Vector* e/ou *List* da *STL* (p/ objetos ou ponteiros de objetos de classes definidos pelos autores)	Sim	No desenvolvimento do projeto como um todo. Menu.h, Menu.cpp.
		- Pilha, Fila, Bifila, Fila de Prioridade, Conjunto, Multi-Conjunto, Mapa **OU** Multi-Mapa.	Sim	PilhaEstados.h e PilhaEstados.cpp.
		Programação concorrente		
		- *Threads* (Linhas de Execução) no âmbito da Orientação a Objetos, utilizando Posix, C-Run-Time **OU** Win32API ou afins.	Sim	PThread.h, PThread.cpp, AtiradinoThread.h e AtiradinoThread.cpp.
		- *Threads* (Linhas de Execução) no âmbito da Orientação a Objetos com uso de Mutex, Semáforos, **OU** Troca de mensagens.	Sim	AtiradinoThread.h, AtiradinoThread.cpp, & PThread.h.
	8	**Biblioteca Gráfica / Visual**		
		- Funcionalidades Elementares. & - Funcionalidades Avançadas como: ● tratamento de colisões ● duplo *buffer*	Sim	Biblioteca SFML: Gerenciador_Grafico.h & Gerenciador_Grafico.cpp

		- Programação orientada e evento em algum ambiente gráfico. **OU** - *RAD – Rapid Application Development* (Objetos gráficos como formulários, botões etc).	Sim	Gerenciador_Grafico.cpp, Menu.cpp, Guigo.cpp, Titi.cpp & Jogo.cpp.
		Interdisciplinaridades por meio da utilização de Conceitos de Matemática e/ou Física.		
		- Ensino Médio.	Sim	Álgebra, Funções, Gravidade, Probabilidade, Geometria plana, Geometria Analítica, Movimento Acelerado.
		- Ensino Superior.	Sim	Lógica, Estrutura de dados, Geometria Analítica, Cálculo 1, Velocidade instantânea.
	9	**Engenharia de Software**		
		- Compreensão, melhoria e rastreabilidade de cumprimento de requisitos. &	Sim	Diagrama de Classes como apoio, e impressão do modelo para o trabalho com o fim de melhor atender os requisitos. Outrossim, foram marcadas reuniões com o monitor e o professor para sanar eventuais dúvidas.
		- Diagrama de Classes em *UML*.	Sim	Largamente utilizado na implementação do projeto, sendo constantemente atualizado tendo em sua totalidade sete versões.
		- Uso efetivo (quiçá) intensivo de padrões de projeto (particularmente GOF).	Sim	Singleton, Composite, Factory Method, State.
		- Testes a luz da Tabela de Requisitos e do Diagrama de Classes.	Sim	Foram utilizados ferramentas para debugar e testar o código como gdb, sempre visando atender os requisitos satisfatoriamente.
	10	**Execução de Projeto**		
		- Controle de versão de modelos e códigos automatizado (via SVN e/ou afins) **OU** manual (via cópias manuais). & - Uso de alguma forma de cópia de segurança (backup).	Sim	Utilizado a ferramenta Git e uma de suas respectivas interfaces acompanhado de um repositório online chamado GitHub.
		- Reuniões com o professor para acompanhamento do andamento do projeto.	Sim	Reuniões no LIC/CITEC dias 29/10, 5/11 e em sala de aula dias 12/11 e 13/11.
		- Reuniões com monitor da disciplina para acompanhamento do andamento do projeto.	Sim	Reuniões nos dias: 4/10, 8/10, 14/10, 16/10, 18/10, 21/10, 25/10, 13/11, 18/11 22/11.

		- Revisão do trabalho escrito de outra equipe e vice-versa.	Sim	Equipe formada pelos alunos Matheus dos Santos e Meika Farias.

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

No.	**Conceitos**	***Listar apenas os utilizados* Situação**
1	**Elementares**	Classe, objetos, atributos, variáveis, constantes e métodos foram utilizados, por serem fundamentais para a programação orientada a objetos. Métodos com retorno const ou parâmetro const foram utilizados principalmente para o bom funcionamento do código em geral.
2	**Relações**	Todas as relações foram utilizadas porque se fez excepcionalmente necessário.
3	**Ponteiros, generalizações e exceções**	Operador *this* foi utilizado porque foi extremamente necessário. Alocação de memória foi utilizado para aproveitamento de memória e melhor desempenho do programa. Gabaritos e *try catch* foram utilizados somente como exemplos para cumprir a tabela de conceitos.
4	**Sobrecarga**	Sobrecarga de métodos e construtoras foram utilizadas pois se fez muito necessário para o desenvolvimento do projeto. Sobrecarga de operadores foi utilizada somente como exemplo para cumprir a tabela de conceitos. Persistência foi utilizada para manter dados gravados fora da execução do programa.
5	**Virtualidade**	Virtualidade foi utilizada porque é fundamental para o paradigma OO.
6	**Organizadores e Estáticos**	Namespace e Nested foram utilizados porque se fez necessário para organizar o código. Estáticos foram utilizados porque se fez extremamente necessário para o funcionamento do programa. Constantes foram utilizadas porque são necessárias para o bom desenvolvimento do código.
7	**Standard Template Library *(STL)* e String OO**	Classes da STL foram utilizadas porque foram necessárias para facilitar o desenvolvimento de algumas soluções.
8	**Biblioteca Gráfica / Visual**	Biblioteca Gráfica e suas funcionalidades foram utilizadas porque foram extremamente necessárias para a viabilização do desenvolvimento do jogo.
9	**Engenharia de Software**	Padrões de Projeto foram utilizados porque são boas ferramentas para desenvolvimento de software. Diagrama de Classes, rastreabilidade de requisitos e testes foram utilizados porque são etapas fundamentais para engenharia de software.
10	**Execução de Projeto**	Git e Github foram utilizados para facilitar o gerenciamento de versões, backups e compartilhamento

		do código. Reuniões com o professor e monitor foram extremamente necessárias para sanar dúvidas e ajudar no encaminhamento do projeto.

DISCUSSÃO E CONCLUSÕES

O desenvolvimento do trabalho como um todo possibilitou aos alunos aplicarem, como demonstrado na Tabela 3, toda a gama de conceitos referente ao paradigma de Programação Orientada a Objetos.

Inicialmente, na etapa de modelagem (análise e projeto) foram levantados diversos requisitos, referentes tanto à execução do diagrama de classes em UML quanto à implementação do código em si. Em tempo foram corrigidas e aperfeiçoadas, em reuniões iniciais com o professor, certas particularidades postuladas no diagrama de classes base[1] e no modelo para elaboração artigo-relatório[3], a fim de melhor atender as necessidades do projeto.

Referente à elaboração do diagrama de classes do projeto e posteriormente implementação do código, foram realizadas múltiplas reuniões para acompanhar o desenvolvimento do trabalho, com o intuito de proporcionar aos alunos uma situação real de engenharia de software, no qual o professor e o monitor da disciplina representam profissionais de maior hierarquia.

Salienta-se que todos os requisitos foram cumpridos, com seus respectivos conceitos utilizados, seja em ampla escala ou em situações específicas. Em relação a isso, destaca-se que a implementação de padrões de projeto foi especialmente benéfica, dado que tal prática é de suma importância para o paradigma OO e seu conhecimento representa amadurecimento das habilidades necessárias a um bom programador. Nesse sentido, conclui-se que a execução e resultado do trabalho foram, considerando o intuito de aprendizado, muito satisfatórios.

CONSIDERAÇÕES PESSOAIS

Tendo em vista o projeto como um todo, considera-se a experiência como muito gratificante do ponto de vista didático e acadêmico, principalmente quando se considera o nível de autonomia e trabalho em equipe que foi necessário para o desenvolvimento do trabalho, tanto na parte de usar uma ferramenta de modelagem, neste caso Star UML, quanto na de aprender a usar uma biblioteca gráfica, conhecimentos que certamente apresentam grande enriquecimento no aprendizado e formação profissional do aluno de sistemas de informação.

DIVISÃO DO TRABALHO

Nesta seção além da Tabela 4, é importante destacar que a divisão do trabalho foi feita de maneira igual para cada um dos autores, sendo que os dois trabalharam um número equivalente de horas neste projeto.

Tabela 4. Lista de Atividades e Responsáveis.

Atividades.	Responsáveis
Levantamento de Requisitos	Guilherme e Tiago
Diagramas de Classes	Tiago
Programação em C++	Mais Guilherme que Tiago

Implementação de *Threads*	Guilherme
Implementação da Persistência dos Objetos	Mais Guilherme que Tiago
Análise do Projeto	Guilherme e Tiago
Implementação das Colisões	Mais Tiago que Guilherme
Escrita do Trabalho	Guilherme e Tiago
Revisão do Trabalho	Guilherme e Tiago

AGRADECIMENTOS

Agradecemos principalmente o Prof. Jean M. Simão pelo aprendizado que nos proporcionou. Agradecemos também o monitor da disciplina Felipe Alves que nos ajudou muito para o desenvolvimento do trabalho. Por fim, somos gratos pelas outras equipes e colegas que nos ajudaram como Mateus dos Santos, Pedro Ostroschi e Thiago Frois.

REFERÊNCIAS CITADAS NO TEXTO

[1] SIMÃO, J. M. Diagrama de Classes de Base, Curitiba – PR, Brasil, Acessado em 22/11/2019, às 16:00: http://www.dainf.ct.utfpr.edu.br/~jeansimao/Fundamentos2/TopicosTrab/DiagramaJogoModelo.jpg

[2] VONCK, Hilze. SFML 2.4 For Beginners - 12: Collision Detection (AABB). 2016. (19m36s). Acesso em: 08/10/2019. Disponível em: <https://www.youtube.com/watch?v=l2iCYCLi6MU&list=PL21OsoBLPpMOO6zyVlxZ4S4hwkY_SLRW9&index=13>.

[3] VONCK, Hilze. SFML 2.4 For Beginners - 9: Animation. 2016. (19m36s). Acesso em: 09/10/2019. Disponível em: <https://www.youtube.com/watch?v=Aa8bXSq5LDE&list=PL21OsoBLPpMOO6zyVlxZ4S4hwkY_SLRW9&index=11&t=0s>.

REFERÊNCIAS UTILIZADAS NO DESENVOLVIMENTO

[A] BEZERRA, E. Princípios de Análise e Projeto de Sistemas com UML. Editora Campus. 2003. ISBN 85-352-1032-6.

[B] HORSTMANN, C. Conceitos de Computação com o Essencial de C++, 3ª edição, Bookman, 2003, ISBN 0-471-16437-2.

[C] DEITEL, H. M.; DEITEL, P. J. C++ Como Programar. 5ª Edição. Bookman. 2006.

[D] SIMÃO, J. M. Site das Disciplina de Fundamentos de Programação 2, Curitiba – PR, Brasil, Acessado em 09/09/2015, às 15:15: http://www.dainf.ct.utfpr.edu.br/~jeansimao/Fundamentos2/Fundamentos2.htm.