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

LOST IN THE CAVE

Gabriel de França Picinato, Gustavo Chemin Ribeiro
gabrielpicinato@alunos.utfpr.edu.br, gustavochemin@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 propõe o desenvolvimento de um jogo de plataforma como forma de aplicar e consolidar conceitos de programação orientada a objetos em C++. Neste trabalho, foi desenvolvido o jogo *Lost In The Cave*, onde o jogador deve explorar uma caverna e encontrar a saída, enfrentando desafios e inimigos ao longo do caminho. O jogo pode ser jogado por um ou dois jogadores e conta com duas fases, cada uma oferecendo diferentes personagens, obstáculos e níveis de dificuldade. O processo de desenvolvimento envolveu a definição de requisitos e a criação de uma modelagem do jogo utilizando Diagramas de Classe em UML. Em seguida, o jogo foi implementado em C++, aplicando conceitos de POO como classes, herança, polimorfismo, e outros aspectos mais avançados, como o uso de classes abstratas e persistência de dados. Além disso, o jogo permite salvar a pontuação em um menu de classificações e retomar uma jogada salva a qualquer momento. Após a conclusão da implementação, foram realizados testes que confirmaram que o jogo funciona de acordo com as especificações propostas. O desenvolvimento de *Lost In The Cave* permitiu um aprendizado prático e profundo dos conceitos estudados ao longo da disciplina.

Palavras-chave ou Expressões-chave: Desenvolvimento de Jogo em C++ utilizando POO, Modelo de Projeto de Técnicas de Programação, Persistência de Dados e uso de UML no Desenvolvimento de Jogos.

INTRODUÇÃO

Este trabalho foi desenvolvido no contexto da disciplina de Técnicas de Programação, cujo objetivo é aplicar conceitos de programação orientada a objetos em C++ através da implementação de um projeto prático. A proposta consiste no desenvolvimento de um jogo de plataforma, permitindo aos alunos consolidar o conhecimento adquirido em sala de aula ao longo do curso. O trabalho visa não apenas a construção de um software funcional, mas também o entendimento aprofundado de metodologias de desenvolvimento de software e de padrões de design aplicados.

O objeto de estudo e implementação deste trabalho é o jogo *Lost In The Cave*. Neste jogo, os jogadores devem explorar uma caverna, enfrentando desafios e obstáculos com o objetivo de encontrar a saída. O jogo foi projetado para ser jogado por um ou dois jogadores, apresentando diferentes fases, personagens e níveis de dificuldade que aumentam a complexidade do desafio proposto.

O método utilizado para o desenvolvimento deste projeto segue o ciclo de Engenharia de Software de forma simplificada. Primeiramente, houve a compreensão dos requisitos específicos do jogo, seguida pela modelagem utilizando diagramas de classe em UML, que foram derivados de um modelo genérico fornecido. A implementação foi realizada em C++, aplicando os princípios de programação orientada a objetos. Por fim, foram conduzidos testes para verificar o funcionamento correto do software, garantindo que ele atendesse aos requisitos estabelecidos.

As seções subsequentes deste relatório detalham o processo de desenvolvimento em maior profundidade. Serão abordados temas como dificuldades encontradas no desenvolver do jogo e aprendizados adquiridos.

EXPLICAÇÃO DO JOGO EM SI

O jogo *Lost In The Cave* (ou "Perdido Na Caverna") situa-se em um ambiente de caverna e busca trazer ao jogador uma experiência agradável e ao mesmo tempo desafiante. O jogo disponibiliza duas fases, que podem ou não serem jogadas sequencialmente, que possuem inimigos fictícios, de diferentes dificuldades, e obstáculos diversificados, que interferem na movimentação do jogador.

Lost In The Cave segue o formato de jogo 2D de plataforma, ao abrir o jogo, o usuário é apresentado a um menu principal, onde pode escolher começar um novo jogo (seja na primeira ou segunda fase), carregar um jogo pausado, conferir o placar de líderes ou até fechar a janela, como mostrado na Figura 1.

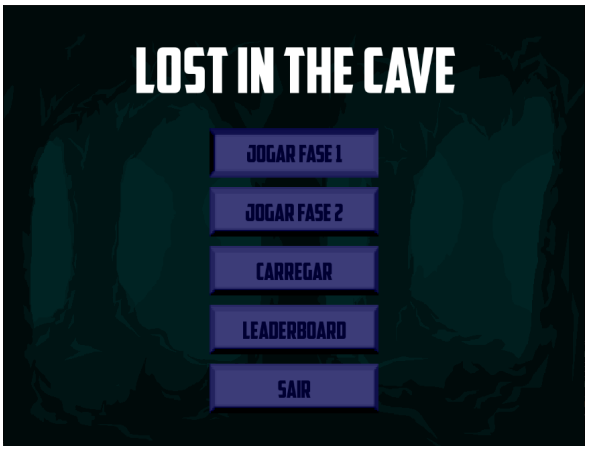

Figura 1: Menu principal do jogo.

Ao selecionar as opções "JOGAR FASE 1" ou "JOGAR FASE 2", o usuário é enviado para um novo menu, onde escolhe se deseja jogar em um ou dois jogadores, como mostrado na Figura 2. Caso a opção selecionada seja "CARREGAR", o jogo se inicia de onde estava na última jogada salva, por sua vez, o botão "LEADERBOARD" disponibiliza uma tela com as maiores pontuações registradas no jogo, se limitando as dez primeiras, representado na Figura 3. Finalmente, o botão "SAIR", como indicado, fecha o jogo.

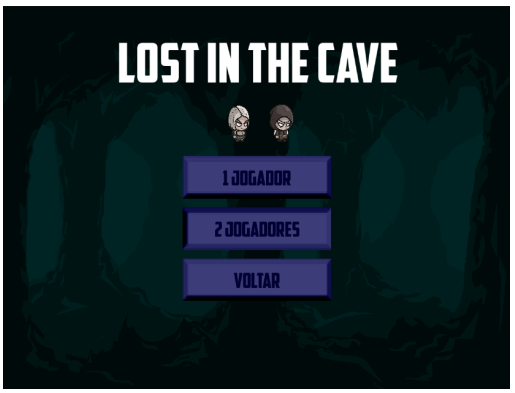

Figura 2: Menu de escolha de jogadores.

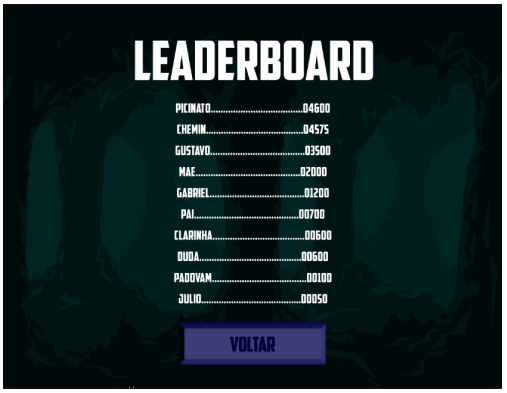

Figura 3: Placar de líderes.

Se o usuário selecionar a opção de um jogador, este será controlado pelas teclas “W”, “A” e “D”, além da tecla de espaço para atacar. Já se a opção escolhida for de dois jogadores, o segundo o jogador será comandado por meio das setas do teclado, usando a tecla “K” para atacar.

As Figuras 4 e 5 mostram, respectivamente, as Fases 1 e 2, que se diferenciam pelo mapa, que é consideravelmente mais desafiador na segunda, e pelos inimigos e obstáculos de cada uma: enquanto a Fase 1 possui voadores e espinhos - que tiram vida do jogador, além de empurrar ele para longe - a Fase 2 possui “Chefões” e pedras, que podem ser arrastadas pelo mapa. Ainda, as duas fases possuem teias, que desaceleram o jogador, e atiradores, que disparam projéteis na direção dos jogadores.

Figura 4: Fase 1.

Figura 5: Fase 2.

Em ambas as fases, não é necessário matar os inimigos para terminá-las, basta chegar à saída da caverna, no entanto, a pontuação é incrementada apenas ao neutralizar ou danificar um inimigo, logo, terminar uma fase fugindo dos combates acarretará em uma pontuação pequena. Durante uma fase, se o usuário apertar a tecla “P”, abrirá uma tela de pausa, onde o jogador pode escolher continuar o jogo, salvar aquela jogada, ou então voltar ao menu principal, como mostrado na Figura 6.

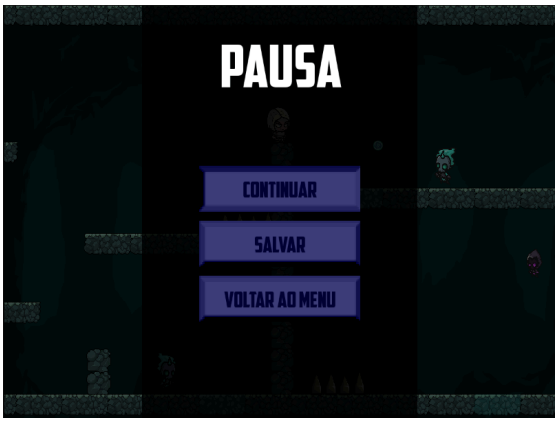

Figura 6: Menu Pausa.

DESENVOLVIMENTO DO JOGO

Nessa seção, serão abordadas algumas especificações do desenvolvimento do *software* do jogo. Na Tabela 1, estão detalhados conceitos funcionais do jogo, especificando em que parte do projeto cada conceito foi aplicado.

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 (previstas nos demais requisitos).	Requisito previsto inicialmente e realizado.	Requisito cumprido via classes do namespace Menu e seus respectivos objetos e métodos, 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 cujos objetos são agregados nas classes do namespace Fases. A classe MenuJogar permite a escolha do número de jogadores.
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 por meio das classes do namespace Fases e da classe GerenciadorEstados, permitindo a troca ou inicialização das Fases, que, por sua vez, são Estados também. Os jogadores desferem espadadas para neutralizar os inimigos.
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 dever ser um 'Chefão'.	Requisito previsto inicialmente e realizado.	Requisito cumprido com as 3 classes derivadas de Inimigo, que se encontram no namespace Personagem, sendo a classe Atirador capaz de lançar projéteis, e a classe Chefão se caracterizando como o inimigo mais forte e resistente.
5	Ter a cada fase ao menos dois tipos de inimigos com número aleatório de instâncias, podendo ser várias instâncias (definindo um máximo) e sendo pelo menos 3 instâncias por tipo.	Requisito previsto inicialmente e realizado.	Requisito cumprido via método criarAleatorios(), presente nas classes FaseUm e FaseDois, que definem por meio da função rand() do C++ o número de cada inimigo em específico, com um máximo predefinido.
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 com as 4 classes derivadas de Obstáculo, presentes no pacote Obstáculos, sendo que a classe Espinho causa dano e repele o jogador ao colidirem.
7	Ter em cada fase ao menos dois tipos de obstáculos com número aleatório (definindo um máximo) de instâncias (*i.e.*, objetos),	Requisito previsto inicialmente e realizado.	Requisito cumprido via método criarAleatorios(), presente nas classes FaseUm e FaseDois, que definem por

	sendo pelo menos 3 instâncias por tipo.		meio da função rand() do C++ o número de cada obstáculo em específico, com um máximo predefinido.
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 via método criarMapa() das classes FaseUm e FaseDois, que faz a leitura de um arquivo .txt que cria todo o cenário com obstáculos e plataformas.
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 de alguma ´gravidade´ no âmbito deste jogo de plataforma vertical e 2D.	Requisito previsto inicialmente e realizado.	Requisito cumprido via classe GerenciadorColisoes, verificando todas as colisões possíveis, e via função executar(dt: const float), que aplica a gravidade para cada Entidade do jogo.
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/Recuperar** Jogada.	Requisito previsto inicialmente e realizado.	(1) Requisito cumprido via classe MenuSalvarColocacao e MenuPrincipal, para salvamento de pontuações e consulta às maiores. (2) Requisito cumprido via MenuPausa e MenuPrincipal.
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).

A classe principal do projeto é a classe Jogo, que contém alguns gerenciadores responsáveis pela execução do código.

No namespace *Gerenciadores*, existem classes diferentes para facilitar o desacoplamento do código. A classe *GerenciadorColisoes*, por exemplo, funciona como o padrão de projeto *Mediator*, avisando a cada entidade que ela colidiu com outra, intermediando as ações entre as *Entidades*. Já o *GerenciadorEstados,* segue o padrão de projeto *State,* possuindo uma pilha de Estados (como Menus ou Fases) e executando sempre o topo da pilha. O *GerenciadorEventos* verifica qualquer evento que acontece enquanto o jogo está aberto, se o evento for um botão pressionado, por exemplo, o *GerenciadorEventos* comunica-se com o *GerenciadorInputs*, que, por sua vez, se comunica com todos os *Observadores* do código, estes, por fim se comunicam com os *Jogadores* ou *Menus*. Essa sequência de passos segue o padrão de projeto *Chain of Responsibility,* já entre o *GerenciadorInputs* e os *Observadores* foi usado o padrão de projeto *Observer*, onde o Gerenciador faz o papel do *subject*. Por fim, as classes *GerenciadorGrafico* e *GerenciadorArquivos* são responsáveis por administrar os acessos à janela do jogo e aos arquivos externos utilizados no código, respectivamente. Além disso, o *GerenciadorEstados*, *GerenciadorEventos, GerenciadorInputs* e o *GerenciadorGrafico* são criados com base no padrão de projeto *Singleton*. A Figura 7 apresenta o diagrama de classes em UML do projeto.

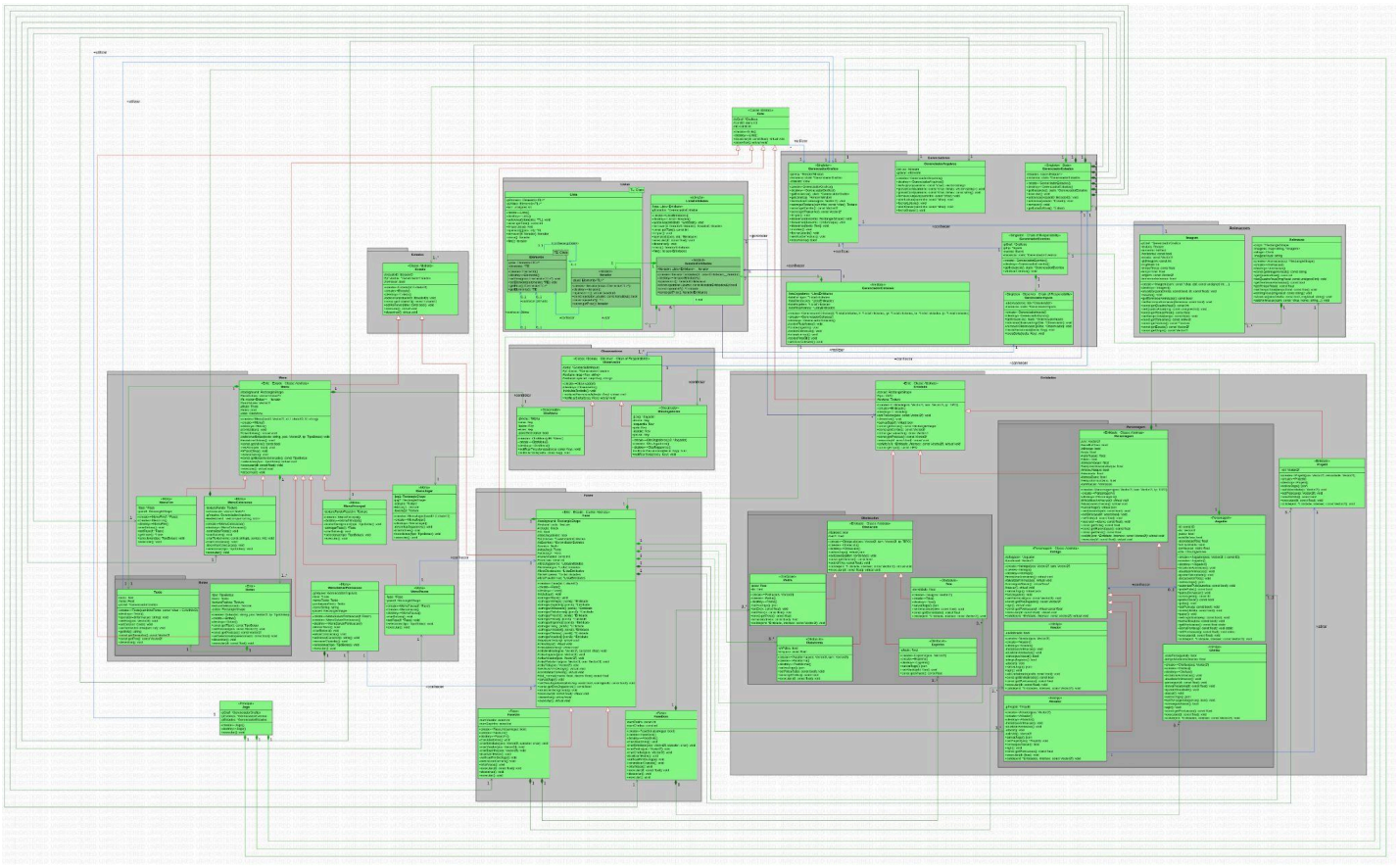

Figura 7. Diagrama de Classes de base em UML[1].

TABELA DE CONCEITOS UTILIZADOS E NÃO UTILIZADOS

Além da realização dos requisitos funcionais, também são apresentados os conceitos que foram lecionados e estudados durante a disciplina.

A Tabela 2 ilustra tais conceitos, com suas respectivas justificativas e locais de uso, caso tenham sido utilizados.

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

N.	Conceitos	Uso	Onde / O quê / Justificativa em uma linha
1	**Elementares:**		
1.1	- Classes, objetos. & - Atributos (privados), variáveis e constantes. & - Métodos (com e sem retorno).	Sim	Todos .hpp e .cpp, como nas classes nos *namespaces* Entidades e Menu.
1.2	- Métodos (com retorno *const* e parâmetro *const*). & - Construtores (sem/com parâmetros) e destrutores	Sim	Na maioria dos .hpp e .cpp, com destaque aos métodos *sets* e *gets*. Construtoras e destrutoras presentes em todas as classes.
1.3	- Classe Principal.	Sim	Classe Jogo
1.4	- Divisão em .h e .cpp.	Sim	No desenvolvimento como um todo.

[1] Para melhor visualização do UML, visite e faça o download em: LostInTheCave_UML.pdf

2	**Relações de:**		
2.1	- Associação direcional. & - Associação bidirecional.	Sim	Entre Atirador e Projetil e Jogador e ObsJogador, respectivamente, por exemplo.
2.2	- Agregação via associação. & - Agregação propriamente dita.	Sim	Em vários dos .hpp e .cpp, por exemplo, as Fases agregam as Entidades e Botões agregam Texto.
2.3	- Herança elementar. & - Herança em diversos níveis.	Sim	Entre os diferentes Menus e entre Inimigos e Personagens, por exemplo.
2.4	- Herança múltipla.	Sim	Classes Fase e Menu, herdam Ente e Estado.
3	**Ponteiros, generalizações e exceções**		
3.1	- Operador *this* para fins de relacionamento bidirecional.	Sim	Para o relacionamento entre Jogador e Menu com seus respectivos Observadores.
3.2	- Alocação de memória (*new* & *delete*).	Sim	Para criação das Entidades nas Fases, por exemplo.
3.3	- Gabaritos/*Templates* criada/adaptados pelos autores (*e.g.*, Listas Encadeadas via *Templates*).	Sim	Foi usada uma Lista Encadeada via *Templates* baseada na disponibilizada pelo professor no Moodle da disciplina.
3.4	- Uso de Tratamento de Exceções (*try catch*).	Sim	Foi utilizado na criação de mapas das Fases para ler o arquivo .txt.
4	**Sobrecarga de:**		
4.1	- Construtoras e Métodos.	Sim	FaseUm e FaseDois possuem construtoras com parâmetros diferentes para criar ou carregar a fase. Diversas classes possuem duas funções executar, uma sem parâmetros e outra com um “const float dt”.
4.2	- Operadores (2 tipos de operadores pelo menos – Quais?).	Sim	Foi usado **operator=** na classe Texto, para atribuir a string do texto, e foi usado **operator-=** na classe Personagem, para tirar vida do personagem.
---	**Persistência de Objetos (via arquivo de texto ou binário)**		
4.3	- Persistência de Objetos.	Sim	Salvamento e carregamento das pontuações salvas, via arquivo .txt.
4.4	- Persistência de Relacionamento de Objetos.	Sim	Salvamento e carregamento da Fase em .json, mantendo relacionamento entre objetos.
5	**Virtualidade:**		
5.1	- Métodos Virtuais Usuais.	Sim	A função salvarJogo() de Entidade é virtual usual.
5.2	- Polimorfismo.	Sim	Todos os Personagens e Obstáculos são tratados como Entidade na ListaEntidades.
5.3	- Métodos Virtuais Puros / Classes Abstratas.	Sim	Em diversas classes, como Ente, Entidade, Obstaculo, Personagem e Inimigo.
5.4	- Coesão/Desacoplamento efetiva e intensa com o apoio de padrões de projeto.	Sim	Nesse projeto foram utilizados os seguintes padrões que auxiliaram no desacoplamento: *Singleton*, *Mediator*, *Iterator*, *State*, *Observer* e *Chain of Responsibility*
6	**Organizadores e Estáticos**		
6.1	- Espaço de Nomes (*Namespace*) criada pelos autores.	Sim	Praticamente todas as pastas se tornam *namespaces*, como Entidades, Personagens e Estado.
6.2	- Classes aninhadas (*Nested*) criada pelos autores.	Sim	Nas classes Lista e ListaEntidades, são aninhadas, respectivamente, as classes Iterador e IteradorEntidades.
6.3	- Atributos estáticos e métodos estáticos.	Sim	Tudo que envolve a pontuação dos jogadores é estático, tanto atributo, como métodos.

6.4	- Uso extensivo de constante (*const*) parâmetro, retorno, método...	Sim	Praticamente todos os métodos *set's* recebem parâmetro *const* e praticamente todos os *get's* são *const* e seu retorno também é *const.*
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	O GerenciadorArquivos utiliza um *vector* de *string*, já cumprindo os dois requisitos.
7.2	- Pilha, Fila, Bifila, Fila de Prioridade, Conjunto, Multi-Conjunto, Mapa **OU** Multi-Mapa.	Sim	O GerenciadorEstados utiliza uma pilha para gerenciar os estados, por exemplo.
---	**Programação concorrente**		
7.3	- *Threads* (Linhas de Execução) no âmbito da Orientação a Objetos, utilizando Posix, C-Run-Time **OU** Win32API ou afins.	Não	Não foi aplicado no projeto.
7.4	- *Threads* (Linhas de Execução) no âmbito da Orientação a Objetos com uso de Mutex, Semáforos, **OU** Troca de mensagens.	Não	Não foi aplicado no projeto.
8	**Biblioteca Gráfica / Visual**		
8.1	- Funcionalidades Elementares. **&** - Funcionalidades Avançadas como: ● tratamento de colisões ● duplo *buffer*	Sim	Tratamento de colisões realizado na classe GerenciadorColisões. Jogo é atualizado com base na diferença de tempo entre dois frames.
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	O GerenciadorEventos analisa qualquer evento que aconteça enquanto o jogo está aberto e toma as devidas decisões.
---	**Interdisciplinaridades via utilização de Conceitos de <u>Matemática Contínua e/ou Física</u>.**		
8.3	- Ensino Médio Efetivamente.	Sim	Conceito de queda livre, com uso da gravidade.
8.4	- Ensino Superior Efetivamente.	Sim	Distribuição normal da posição dos Inimigos e Obstáculos com média no centro do mapa
9	**Engenharia de Software**		
9.1	- Compreensão, melhoria e rastreabilidade de cumprimento de requisitos. &	Sim	O rastreamento dos requisitos cumpridos foi feito quase que diariamente pelos integrantes da dupla.

9.2	- Diagrama de Classes em *UML*.	Sim	Foi feito o diagrama de classes, representado na Figura 7.
9.3	- Uso efetivo e intensivo de padrões de projeto *GOF*, *i.e.*, mais de 5 padrões.	Sim	Foram utilizados: *Singleton, Iterator, Mediator, Observer, State* e *Chain of Responsibility.*
9.4	- Testes à luz da Tabela de Requisitos e do Diagrama de Classes.	Sim	O jogo foi desenvolvido pensando-se sempre no cumprimento dos requisitos e no modelo base do diagrama de classes.
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	As versões do código foram armazenadas no github, além disso, alguns arquivos .zip foram armazenados em pastas seguras ao longo do desenvolvimento do projeto. Link do github: https://github.com/gabpicinato/LostInTheCave
10.2	- Reuniões com o professor para acompanhamento do andamento do projeto. **[ITEM OBRIGATÓRIO PARA A ENTREGA DO TRABALHO]**	Sim	Reuniões realizadas: 20/08 - 09h30 - 10h00 26/08 - 10h30 - 11h00
10.3	- Reuniões com monitor da disciplina para acompanhamento do andamento do projeto. **[ITEM OBRIGATÓRIO PARA A ENTREGA DO TRABALHO]**	Sim	Reuniões realizadas: Monitor Giovane - 19/08 - 08h30 - 09h00 Monitor Nicky - 22/08 - 10h00 - 10h30 Monitor Giovane - 22/08 - 21h40 - 22h20 Monitora Gabrielle - 26/08 - 08h50 - 09h30
10.4	- Revisão do trabalho escrito de outra equipe e vice-versa.		Trabalho revisado pela dupla Gustavo Henrique e Gustavo Padovam e vice-versa.
Total de conceitos apropriadamente utilizados.			**95%** (noventa e cinco por cento).

DISCUSSÃO E CONCLUSÕES

O projeto foi um sucesso em sua proposta: aplicação prática dos conteúdos aprendidos na disciplina de Técnicas de Programação. O jogo tornou necessário o uso de conceitos importantes de programação orientada a objetos, como herança, polimorfismo, associação e agregação. Além disso, o uso de Diagramas de Classe UML ajudou no desenvolvimento do projeto, além de introduzir metodologias de Engenharia de *Software.* Além disso, o uso de padrões de projeto introduziu a importância do desacoplamento em um código.

Por fim, dados os objetivos do projeto, a dupla considera que o desenvolvimento do jogo foi bem sucedido, uma vez que a aplicação prática de conceitos passados em sala de aula foi de extrema importância para o aprendizado efetivo do conteúdo.

DIVISÃO DO TRABALHO

Em maior parte, o trabalho foi realizado em conjunto, de forma síncrona, apenas em dois momentos um dos integrantes fez mais que o outro por conta de compromissos externos à faculdade. A Tabela 3 detalha as etapas do projeto com seus respectivos responsáveis.

Tabela 3. Lista de Atividades e Responsáveis.

Atividades.	**Responsáveis**
Compreensão de Requisitos	Gabriel e Gustavo
Diagramas de Classes	Gabriel e Gustavo
Programação em C++	Gabriel e Gustavo
Engenharia de Software	Gabriel e Gustavo
Implementação de *Template*	mais Gabriel que Gustavo
Implementação de Padrões de Projeto	Gabriel e Gustavo
Implementação dos Gerenciadores	Gabriel e Gustavo
Implementação das Entidades	Gabriel e Gustavo
Implementação das classes do namespace Menu	Gabriel e Gustavo
Implementação dos Observadores	Gabriel e Gustavo
Implementação das classes do namespace Animações	Gabriel e Gustavo
Procura e edição dos sprites utilizados no jogo	Gabriel e Gustavo
Construção do mapa das fases	Gabriel e Gustavo
Implementação da Persistência dos Objetos	Gabriel e Gustavo
Preparação para a apresentação	Gabriel e Gustavo
Escrita do Trabalho	mais Gustavo que Gabriel
Revisão do Trabalho	Gabriel e Gustavo

Em termos gerais, é possível afirmar que:

- Gabriel Picinato trabalhou em 100% das atividades ou as realizando ou revisando elas posteriormente.

- Gustavo Chemin trabalhou em 100% das atividades ou as realizando ou revisando elas posteriormente.

AGRADECIMENTOS PROFISSIONAIS

Nessa seção, gostaríamos de agradecer ao Prof. Dr. Jean M. Simão, pelas correções e dicas valiosas durante as reuniões citadas na Tabela 2. Agradecemos também à toda equipe do PETECO que participou da realização do "*Project Simas*", que trouxe oficinas com conselhos importantes para realização do jogo. Além disso, agradecemos a todos os monitores da disciplina, pela atenção e ajuda durante as reuniões e consultas ao longo do desenvolvimento do jogo. Por fim, agradecemos à dupla Gustavo Santos e Gustavo Padovam, por revisarem este documento.

REFERÊNCIAS CITADAS NO TEXTO

[1] SIMÃO, J. M. Site das Disciplina de Fundamentos de Programação 2, Curitiba – PR, Brasil, Acessado em 29/08/2024, às 18:16 - http://www.dainf.ct.utfpr.edu.br/~jeansimao/Fundamentos2/Fundamentos2.htm.

[2] PICINATO, Gabriel; CHEMIN, Gustavo. Lost in the Cave: GitHub, 2023. Disponível em: https://github.com/gabpicinato/LostInTheCave. Acesso em: 30 ago. 2024.

REFERÊNCIAS UTILIZADAS NO DESENVOLVIMENTO

[A] GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. Design patterns: elements of reusable object-oriented software. Boston: Addison-Wesley, 1994.

[B] REFACTORING GURU. Design Patterns. Refactoring Guru, [2024]. Disponível em: https://refactoring.guru/design-patterns. Acesso em: 25 ago. 2024.