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

Daniel Insaurriaga Zagroba, Samuel Stroschein Yokoyama
danielzagroba@alunos.utfpr.edu.br, samuelyokoyama@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: Engenharia da Computação / Sistemas de Informação
Universidade Tecnológica Federal do Paraná - UTFPR
Avenida Sete de Setembro, 3165 - Curitiba/PR, Brasil - CEP 80230-901

Resumo – A disciplina de Técnicas de Programação exige o desenvolvimento de um *software* de plataforma, no formato de um jogo digital, para fins de aprendizado de técnicas de engenharia de *software*, particularmente de programação orientada a objetos em C++. Para tal, neste trabalho, escolheu-se o jogo ROBOT++, no qual o jogador enfrenta inimigos em um dado cenário. O jogo tem duas fases que se diferenciam por dificuldades para um ou dois jogadores. Para o desenvolvimento do jogo foram considerados os requisitos previamente propostos e elaborado um modelo (análise e projeto) via Diagrama de Classes em *UML* (*Unified Modeling Language*) usando como base um diagrama previamente proposto. Subsequentemente, em linguagem de programação C++, realizou-se o desenvolvimento que contemplou os conceitos usuais de Orientação a Objetos como Classe, Objeto e Relacionamento, bem como alguns conceitos avançados como Classe Abstrata, Polimorfismo, Gabaritos, Persistências de Objetos por Arquivos, Sobrecarga de Operadores e Biblioteca Padrão de Gabaritos (*Standard Template Library* - *STL*). Depois da implementação, os testes e uso do jogo feitos pelos próprios desenvolvedores demonstraram sua funcionalidade conforme os requisitos e a modelagem elaborada. Por fim, salienta-se que o desenvolvimento em questão permitiu cumprir o objetivo de aprendizado visado.

Palavras-chave ou Expressões-chave:

Desenvolvimento de um jogo digital ROBOT++ utilizando programação orientada a objetos em C++, Modelagem de jogo eletrônico através de Diagrama de Classes UML e com suporte de SFML, Uso de Técnicas de Programação na criação de um jogo digital para aprendizado de engenharia de software.

INTRODUÇÃO

Este trabalho se insere no contexto de aprendizagem avaliativa da disciplina de Técnicas de Programação (ICSE20) e engenharia de software, com o propósito de proporcionar aos desenvolvedores uma oportunidade de aplicar e consolidar conhecimentos sobre conceitos básicos e avançados de programação orientada a objetos, utilizando a linguagem C++. O projeto culmina na criação de um jogo de plataforma chamado “ROBOT++”.

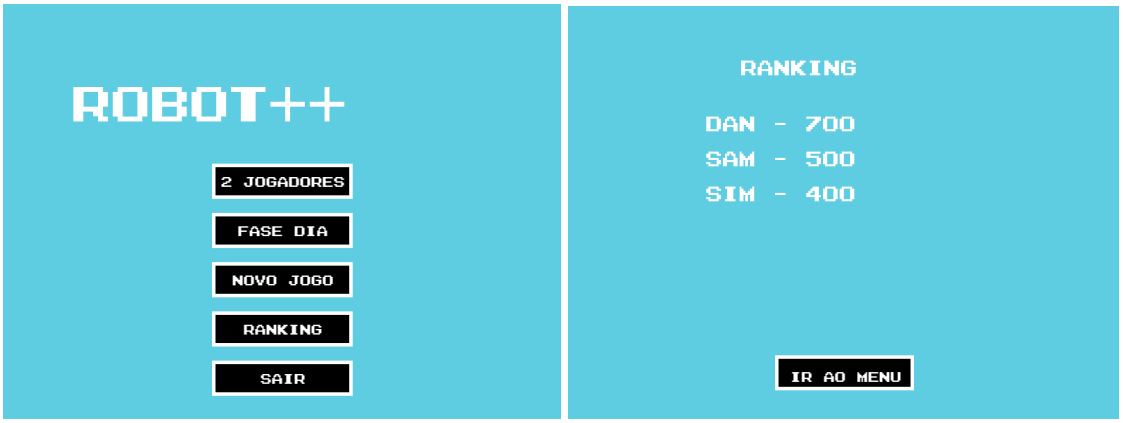
O objeto de estudo e implementação deste trabalho é o jogo de plataforma “ROBOT++”, no qual um ou dois jogadores controlam robôs que devem superar inimigos e obstáculos para alcançar o final de cada nível e concluir o jogo. Após completar os níveis, os jogadores têm a opção de registrar um apelido para verificar se suas pontuações estão entre as mais altas no ranking do jogo.

O método adotado para o desenvolvimento do jogo segue o ciclo clássico de engenharia de software. Inicialmente, foi realizada a compreensão dos requisitos do projeto, seguida pela modelagem utilizando diagramas de classes em UML, com base no diagrama inicial fornecido. À medida que o desenvolvimento progrediu, os diagramas UML foram ajustados para refletir as mudanças e necessidades do projeto. A implementação foi executada em C++, com uma abordagem orientada a objetos, e a fase final incluiu testes rigorosos para assegurar a robustez e a funcionalidade do jogo.

As seções seguintes discorrem sobre o software desenvolvido, incluindo informações sobre o processo de desenvolvimento e os conceitos aprendidos na disciplina que foram aplicados em sua construção.

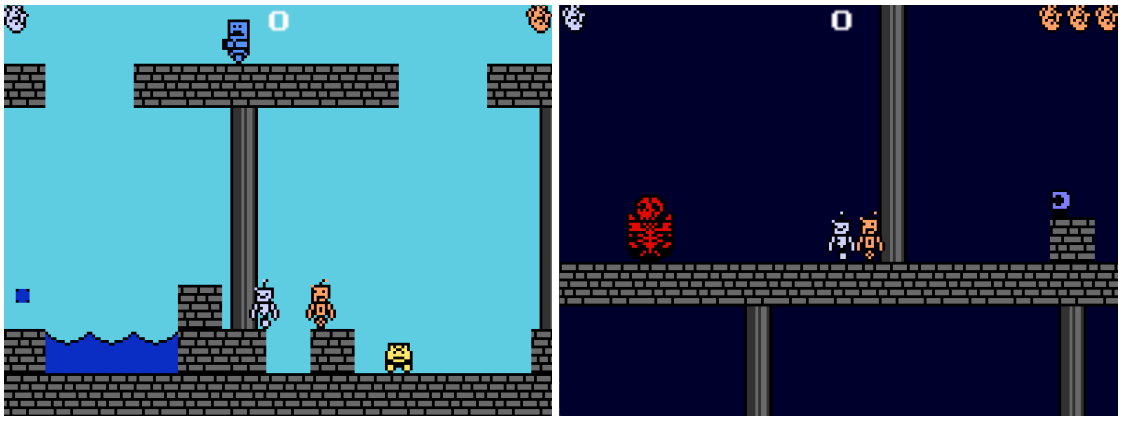
EXPLICAÇÃO DO JOGO EM SI

Ao iniciar a aplicação do jogo, o menu principal (Figura 1) aparece na tela, permitindo as seguintes opções respectivamente: selecionar um ou dois jogadores, escolher entre a Fase Dia ou Noite, iniciar uma nova partida, ver o *ranking*(Figura 2) e sair do jogo (fechar a aplicação).



Figuras 1 e 2. Tela do Menu principal e Ranking respectivamente

O primeiro jogador, controlará o Personagem cinza(Figuras 3 e 4), com as teclas WASD e espaço para o ataque, e caso o jogo o comece com dois jogadores, o segundo jogador controlará o personagem cor bronze(Figuras 3 e 4), com as setas e o botão enter para o ataque. Eles podem através do teclado se locomover, pular e atirar projéteis. Ademais há um sistema de vidas que aparece no topo da tela esquerdo e direito(Figuras 3 e 4), enquanto a fase está sendo jogada.

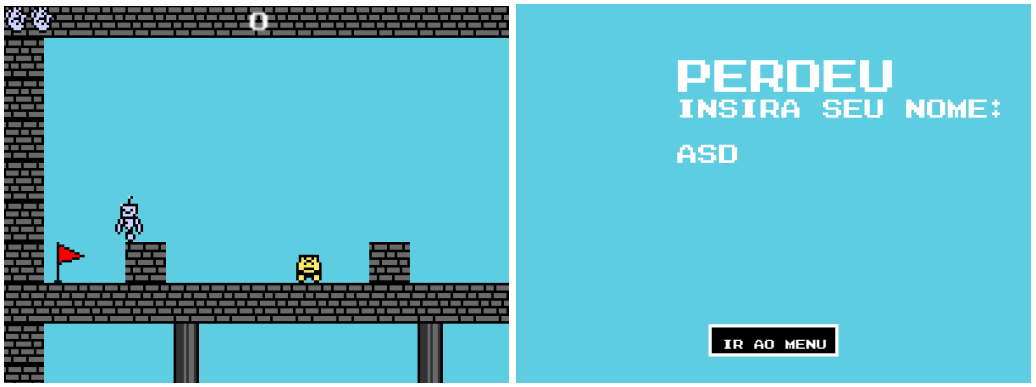


Figuras 3 e 4. Fase dia e Noite respectivamente

Ao iniciar a partida na Fase Dia, o jogo começa em um cenário de cor azul clara, onde o(s) jogador(es) enfrenta(m) inimigos como o Gumbot (amarelo), que causa dano ao ser tocado e se move constantemente, e o ProjectileBot (azul), que causa dano ao lançar projéteis. Ambos são representados na Figura 3. Os obstáculos incluem três elementos, semelhantes aos da Fase Noite: água, que causa dano ao contato; a máquina de projéteis, que os dispara; e a plataforma, que restringe o cenário, os jogadores e outras entidades. Cada um desses obstáculos interfere no objetivo de chegar ao fim da fase.

Ao terminar a Fase Dia (alcançando a bandeira que é visível na Figura 5) ou ao escolher a Fase Noite no menu e iniciar um novo jogo, o(s) jogador(es) é/são levado(s) a um ambiente azul escuro, onde a dificuldade é maior em relação à fase anterior. Na Fase Noite, o jogador encontra novamente o Gumbot, a água, e a máquina de projéteis. Além desses, surge o ByteCrusher, um chefão vermelho visível na Figura 4, que apresenta um nível de dificuldade superior em comparação aos inimigos da fase anterior.

Ao alcançar a bandeira (Figura 5) da Fase Noite ou perder todas as vidas, uma tela aparece onde o(s) jogador(es) pode(m) escolher um apelido de até 3 caracteres e, em seguida, voltar ao Menu Principal para verificar se a pontuação, visível nas Figuras 3, 4 e 5, foi alta o suficiente para entrar no *ranking*. A pontuação é incrementada ao derrotar os inimigos.



Figuras 5 e 6. Fim da Fase Dia e Tela de Derrota, respectivamente

DESENVOLVIMENTO DO JOGO NA VERSÃO ORIENTADA A OBJETOS

Nesta seção, são detalhadas as especificações do desenvolvimento do software do jogo. São apresentados os detalhes das classes e pacotes criados, com ênfase na análise e modelagem, que foram fundamentadas nos requisitos propostos. Além disso, são descritas as soluções implementadas no código para atender a esses requisitos, como demonstrado na Tabela 1 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 (previstas nos demais requisitos).	Requisito previsto inicialmente, mas realizado apenas **PARCIALMENTE** – faltou a opção salvar.	Requisito semi-cumprido via classes dos namespaces Estados e Gerenciadores em conjunto 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 na classe Fase, 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 o Estado MenuPrincipal e juntamente o namespace Fases.
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 via namespace Fases e também namespace Personagens, sendo o inimigo que atira o ProjectileBot e o chefão o ByteCrusher.
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 namespace Fases, que além de checar instâncias mínimas e máximas coloca inimigos com quantidade de instâncias aleatórias.
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 via namespace Fases e também namespace Obstáculos, sendo o obstáculo que dá dano ao colidir é a Água.
7	Ter em cada fase ao menos dois tipos de obstáculos com número aleatório (definindo um máximo) de instâncias (*i.e.*, objetos), sendo pelo menos 3 instâncias por tipo.	Requisito previsto inicialmente e realizado.	Requisito cumprido via ambas as fases terem os obstáculos Água e Máquina Projéteis, que possuem um número mínimo e máximo de instâncias, sendo o número de instâncias aleatório.
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 namespace Fases, que possuem as fases Fase Dia e Fase Noite.
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 Gerenciador de Colisões e também todas entidades sofrem gravidade através da função executar que elas possuem.

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, mas realizado apenas **PARCIALMENTE** – faltou ainda a segunda parte do requisito.	Requisito semi-cumprido através da classe Ranking e do método salvarPontuacao da classe FimJogo, utilizando a biblioteca json[2] para salvar e ler de arquivos json.
Total de requisitos funcionais apropriadamente realizados. *(Cada tópico vale 10%, sendo que para ser contabilizado deve estar realizado efetivamente e não parcialmente)*			**80%** (noventa por cento).

A classe Jogo é a classe principal do programa, responsável por englobar e gerenciar todas as outras classes utilizadas no jogo, garantindo o funcionamento adequado do código.

O *namespace* Gerenciadores consta com os seguintes gerenciadores: (1) Gerenciador de Estados, (2) Gerenciador Gráfico, (3) Gerenciador de Eventos, (4) Gerenciador de Inputs, e (5) Gerenciador de Colisões.

(1) O Gerenciador de Estados possui uma pilha onde os estados do jogo são empilhados. Seu método executar, que é chamado na classe Jogo, sempre executa o estado no topo da pilha. Além disso, o gerenciador possui outros métodos, como adicionar estados ao topo e removê-los do topo.

(2) O Gerenciador Gráfico auxilia na exibição de imagens em conjunto com o SFML, tendo como métodos funções que definem opções de janela, de execução e também uma câmera para as fases.

(3) O Gerenciador de Eventos checa os eventos na janela através do sf:Event do SFML, e os notifica ao Gerenciador de Inputs ou os trata diretamente (Fechar janela).

(4) O Gerenciador de Inputs recebe as notificações do Gerenciador de Eventos e dependendo do estado atual trata os de certa forma. Existe também uma variável mapa que relaciona um evento com funções void e é bastante utilizado para chamar diretamente as funções de classes.

(5) O Gerenciador de Colisões age como mediador, checando e controlando as colisões das entidades como as plataformas, projéteis, inimigos e personagens.

O *namespace* Estados contém os seguintes estados: Menuprincipal, FimJogo, JogarFase e *Ranking*. Contém também elementos que aparecem nos estados como a classe Botão e a classe Texto. Para os elementos de texto foi usada a fonte Zig[3].

O *namespace* Fases contém a classe Fase que deriva nas classes FaseDia e FaseNoite.

O *namespace* Listas contém a classe Template Lista com uma classe aninhada Iterador e a classe ListaEntidades, que utiliza a Lista e o Iterador para chamar polimorficamente os métodos virtuais puros de Entidade.

O *namespace* Entidades contém a classe abstrata Entidade, base para Inimigos, Obstáculos, Jogador, Bandeira e Projétil.Os inimigos incluem Gumbot, que muda de direção ao encontrar obstáculos, ProjectileBot, que dispara projéteis, e ByteCrusher, o chefe final que atira projéteis em diferentes vetores. Os obstáculos incluem a Plataforma, a Água e a Máquina de Projéteis.

Como interdisciplinaridade na classe Entidade foram usados conceitos de gravidade do ensino básico, e conceitos de vetores (Geometria Analítica) para criar velocidade, aceleração e também a aceleração gravitacional.

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

TABELA DE CONCEITOS UTILIZADOS E NÃO UTILIZADOS

Além dos requisitos funcionais, o jogo foi desenvolvido com atenção aos conceitos na tabela a seguir. Nela também constam quais conceitos foram utilizados e onde, assim como os que não foram 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 .h e .cpp, como nas classes nos *namespaces* Entidades e Gerenciadores.
1.2	- Métodos (com retorno *const* e parâmetro *const*). & - Construtores (sem/com parâmetros) e destrutores	Sim	Principalmente na classe Entidade e é possível ver vários casos no UML.
1.3	- Classe Principal.	Sim	Jogo.cpp[1]

[1] Não foi possível apresentar o diagrama de maneira legível que coubesse inteiramente na página

1.4	- Divisão em .h e .cpp.	Sim	No desenvolvimento como um todo, como nas classes nos *namespaces* Entidades e Gerenciadores.
2	**Relações de:**		
2.1	- Associação direcional. & - Associação bidirecional.	Sim	Classes Bandeira e fases possuem associação bidirecional e e ListaEntidades e Entidades unidirecional
2.2	- Agregação via associação. & - Agregação propriamente dita.	Sim	Em vários dos .h e .cpp, como nas classes nos *namespaces* Gerenciadores e Entidades
2.3	- Herança elementar. & - Herança em diversos níveis.	Sim	Na maioria dos elementos do namespace Entidades ocorre ambos.
2.4	- Herança múltipla.	Sim	Precisamente no estado MenuPrincipal, herdeiro das classes Estado e Ente.
3	**Ponteiros, generalizações e exceções**		
3.1	- Operador *this* para fins de relacionamento bidirecional.	Sim	Precisamente na classe Projétil com os personagens que disparam Projéteis(ProjectileBot, Jogador, ByteCrusher).
3.2	- Alocação de memória (*new* & *delete*).	Sim	Principalmente nas classes Fase Dia e Fase Noite.
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*).	Sim	Usado no método salvarPontuacao() de FimJogo para capturar exceções ao abrir/manipular arquivos
4	**Sobrecarga de:**		
4.1	- Construtoras e Métodos.	Sim	Como a construtora da classe Elemento, e o método processainput no Gerenciador de Colisões.
4.2	- Operadores (2 tipos de operadores pelo menos – Quais?).	Sim	Foi usado o *operator*== e o *operator*++ na classe aninhada Iterator da Lista.
---	**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	Na classe Personagem.
5.2	- Polimorfismo.	Sim	O método executar, presente em várias classes.
5.3	- Métodos Virtuais Puros / Classes Abstratas.	Sim	Na classe abstrata obstáculo.
5.4	- Coesão/Desacoplamento efetiva e intensa com o apoio de padrões de projeto.	Não	Utilizamos padrões de projetos mas sem desacoplamento efetivo.
6	**Organizadores e Estáticos**		
6.1	- Espaço de Nomes (*Namespace*) criada pelos autores.	Sim	Namespace Estados.
6.2	- Classes aninhadas (*Nested*) criada pelos autores.	Sim	Classe iterator.
6.3	- Atributos estáticos e métodos estáticos.	Sim	Gerenciadores possuem tanto atributos como métodos estáticos.

6.4	- Uso extensivo de constante (*const*) parâmetro, retorno, método...	Sim	Demonstrado em várias classes, principalmente na Entidade.
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	A *String* é utilizada na classe Texto e o *Vector* e o *List* do STL para ponteiros de objetos é usada na Classe do Gerenciador de Colisões.
7.2	- Pilha, Fila, Bifila, Fila de Prioridade, Conjunto, Multi-Conjunto, Mapa **OU** Multi-Mapa.	Sim	Fila é usada no Gerenciador de Estados, mapa é utilizado no Gerenciador de Inputs, conjunto é usado no Gerenciador de Colisões.
---	**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	Foram utilizadas as funções *clear* e *display*(que fazem duplo buffer) da biblioteca gráfica SFML que são chamadas através de métodos com mesmo nome no Gerenciador Gráfico. O tratamento de colisões é feito no Gerenciador de Colisões utilizando a forma de colisão AABB(Axis-Aligned Bounding Box), que trata a área de colisão dos *sprites* como retângulos.
8.2	- Programação orientada a evento efetiva (com gerenciador apropriado de eventos inclusive) em algum ambiente gráfico. **OU** - *RAD – Rapid Application Development* (Objetos gráficos como formulários, botões etc).	Sim	Foi utilizado o sf::Event do SFML no Gerenciador de Eventos para controlar eventos de janela e repassar eventos de teclado e mouse para o Gerenciador de Inputs.
---	**Interdisciplinaridades via utilização de Conceitos de <u>Matemática Contínua e/ou Física</u>.**		
8.3	- Ensino Médio Efetivamente.	Sim	Aplicação da força gravitacional ao mover entidades.
8.4	- Ensino Superior Efetivamente.	Sim	Cálculos vetoriais (Geometria Analítica).
9	**Engenharia de Software**		

9.1	- Compreensão, melhoria e rastreabilidade de cumprimento de requisitos. &	Sim	Requisitos compreendidos durante o desenvolvimento do jogo e monitorados durante a implementação.
9.2	- Diagrama de Classes em *UML*.	Sim	Diagrama de classes confeccionado a partir do diagrama fornecido
9.3	- Uso efetivo e intensivo de padrões de projeto *GOF*, *i.e.*, mais de 5 padrões.	Não	Utilizamos Padrões de Projeto como o Command, State, Singleton, Iterator e Mediator.
9.4	- Testes à luz da Tabela de Requisitos e do Diagrama de Classes.	Sim	O desenvolvimento do jogo foi fundamentado no cumprimento integral da Tabela de Requisitos e do Diagrama de Classes. Os testes foram realizados com o objetivo de verificar e implementar os requisitos estabelecidos.
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	Utilizou-se o GitHub[1] para o controle de versão dos modelos, no qual cada membro da equipe desenvolvia em sua própria branch. Após alcançar um resultado estável, as alterações eram integradas à branch principal (main). Foi utilizado como forma de cópia de segurança, cópias do jogo estáveis e em máquina.
10.2	- Reuniões com o professor para acompanhamento do andamento do projeto.	Sim	Nos dias 21/08/ e 28/08, ambas reuniões às 16:00.
10.3	- Reuniões com monitor da disciplina para acompanhamento do andamento do projeto.	Sim	Samuel participou do curso do PETECO anteriormente a grave, Daniel e Samuel participaram do curso do PETECO após a greve. Reunião com Gabrielle e Giovane no dia 12/08. Reunião com Giovane dia 26/08. Reunião com Nicky dia 27/08.
10.4	- Revisão do trabalho escrito de outra equipe e vice-versa.	Sim	Trabalho revisado pela dupla Caio e Silva Barbieri e Ana Julia Molinos Leite da Silva
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%.)*			**85,0%** (oitenta e cinco por cento).

DISCUSSÃO E CONCLUSÕES

O desenvolvimento do jogo ROBOT++ permitiu que os desenvolvedores aplicassem na prática os conceitos fundamentais de programação orientada a objetos e modelagem de software aprendidos no decorrer do semestre. A equipe conseguiu cumprir a maior parte dos requisitos estabelecidos, e a experiência prática evidenciou a importância do uso de padrões de projeto e desacoplamento, mesmo que tais elementos não tenham sido totalmente implementados no código. A conclusão do projeto destacou-se como uma etapa crucial de aprendizado, promovendo tanto o aprimoramento das habilidades técnicas quanto a colaboração efetiva em equipe. O resultado final reflete claramente o progresso alcançado e a competência desenvolvida durante o desenvolvimento do projeto.

DIVISÃO DO TRABALHO

A tabela 3 mostra como foram divididas as atividades pertinentes ao desenvolvimento do jogo ROBOT++ entre os desenvolvedores.

Tabela 3. Lista de Atividades e Responsáveis.

Atividades.	**Responsáveis**
Compreensão de Requisitos	Daniel e Samuel
Diagramas de Classes	Mais Samuel que Daniel
Programação em C++	Daniel e Samuel
Design Gráfico	Mais Daniel que Samuel
Implementação de *Template*	Samuel
Implementação da Persistência dos Objetos...	Mais Samuel que Daniel
Implementação do namespace Gerenciadores	Mais Daniel que Samuel
Implementação do namespace Entidades	Daniel e Samuel
Implementação do namespace Estados	Mais Daniel que Samuel
Implementação do namespace Fases	Daniel e Samuel
Preparação e apresentação do trabalho	Daniel e Samuel
Escrita do Trabalho	Daniel e Samuel
Revisão do Trabalho	Daniel e Samuel

- Daniel trabalhou em 100% das atividades ou as realizando ou colaborando nelas efetivamente.
- Samuel trabalhou em 100% das atividades ou as realizando ou colaborando nelas efetivamente.

AGRADECIMENTOS PROFISSIONAIS

Gostaríamos de expressar nossa gratidão ao professor Jean M. Simão, da disciplina, por seus ensinamentos e orientações durante o desenvolvimento deste trabalho. Agradecemos também aos monitores da disciplina, Giovane, Gabrielle e Nicky, pelo suporte e assistência oferecidos.

Agradecemos aos estudantes que realizaram o minicurso do PETECO pela colaboração e contribuição para o projeto. Por fim, nossa sincera gratidão a Caio e Silva Barbieri e Ana Julia Molinos Leite da Silva por revisarem o trabalho.

REFERÊNCIAS CITADAS NO TEXTO

[1] Endereço do repositório do jogo no GitHub, Acessado em 02/09/2024, às 20:17 - https://github.com/danzagroba/ROBOT_plus_plus.
[2]LOHMANN, N. Repositório JSON para C++, Acessado em 02/09/2024, às 20:39 - https://github.com/nlohmann/json.
[3]Brandon, S. Fonte Zig. Acessado em 02/09/2024, às 21:15 - https://www.1001fonts.com/zig-font.html.

REFERÊNCIAS UTILIZADAS NO DESENVOLVIMENTO

[A] SALVI, G. L. "Criando um Jogo em C++ do ZERO", Playlist do YouTube, Acessado em 02/09/2024, às 20:32 - https://www.youtube.com/playlist?list=PLR17O9xbTbIBBoL3lli44N8LdZVvg-_uZ.
[B] SIMÃO, J. M. Site das Disciplina de Fundamentos de Programação 2, Curitiba – PR, Brasil, Acessado em 02/09/2024, às 20:22 - https://pessoal.dainf.ct.utfpr.edu.br/jeansimao/Fundamentos2/Fundamentos2.htm.
[C] SFML DEVELOPMENT TEAM. "Learn SFML", Tutoriais do SFML Acessado em 02/09/2024, às 21:08 - https://www.sfml-dev.org/learn.php.