

Gabriel Eduardo Lima

Relatório sobre o desenvolvimento de um Simulador Físico 2D simples

Blumenau-SC

2021

Gabriel Eduardo Lima

Relatório sobre o desenvolvimento de um Simulador Físico 2D simples

Relatório desenvolvido para apresentação e
avaliação de projeto da disciplina de Progra-
mação Orientada a Objetos I

Instituto Federal Catarinense - Campus Blumenau

Blumenau-SC

2021

Resumo

O presente trabalho objetiva a apresentação do projeto desenvolvido para a disciplina de Programação Orientada a Objetos I, onde o tema definido para abordagem é simulação física 2D. Através de pesquisas a cerca de aspectos de programação como Orientação a Objetos e linguagem C++, juntamente do estudo de diversos aspectos matemáticos e físicos, é possível construir a simulação de um sistema elástico onde aplicam-se conhecimentos como integração para movimentação de um corpo físico a partir de uma função de aceleração conhecida, conservação de movimento e reflexão especular para cálculo de colisões. O desenvolvimento do projeto consiste na divisão desse em três pacotes, onde cada um é responsável por uma funcionalidade do sistema como um todo. Os pacotes podem ser brevemente separados em Base, Interface e Arquivo onde têm-se respectivamente componentes para representação e estruturação da base matemática, abstração e criação de uma interface gráfica e por fim, persistência de dados. Além da implementação desses pacotes obtêm-se como resultado final do projeto uma série de testes dessas componentes, bem como uma simulação final correlacionando a teoria matemática e física à programação desenvolvida.

Palavras-chaves: Simulação Física. Aplicação Matemática e Física. Programação Orientada a Objetos.

Lista de ilustrações

Figura 1 – Ciclo de Desenvolvimento aplicado no Projeto	9
Figura 2 – Exibição do conteúdo do arquivo README.md presente no diretório Básico SDL	19
Figura 3 – Exibição da declaração de variáveis SDL e inicialização da biblioteca .	21
Figura 4 – Exibição da estrutura para representar uma circunferência, bem como o cálculo para gerar ela graficamente	22
Figura 5 – Exibição do tratamento de eventos simples	23
Figura 6 – Exibição do cálculo de FPS e as chamadas de finalização da biblioteca ao final do programa	24
Figura 7 – Aproximação numérica da integração a partir da soma de áreas retangulares	30
Figura 8 – Colisão elástica em uma direção de movimento	34
Figura 9 – Fórmula da velocidade dos corpos após uma Colisão elástica para simulações bidimensionais	35
Figura 10 – Apresentação do efeito de reflexão especular	36
Figura 11 – Exemplo da aplicação do equacionamento de reposicionamento da circunferência verde resultando na circunferência azul usando-se o software Geogebra	38
Figura 12 – Apresentação de algumas matrizes de transformação, bem como o resultado causado na multiplicação por uma matriz representando os vértices de um quadrado	40
Figura 13 – Diagrama de classes da componente Objeto do Pacote Base	45
Figura 14 – Código do método movimentar presente no Objeto que realiza os processos de Integração numérica	47
Figura 15 – Diagrama de classes da componente DetectorColisao do Pacote Base .	49
Figura 16 – Código para a detecção de colisão entre circunferências	51
Figura 17 – Código para a aplicação da conservação da quantidade de movimento .	51
Figura 18 – Código para o cálculo de reposicionamento do Objeto na colisão com uma Linha	52
Figura 19 – Teste da classe Objeto com foco para a movimentação	53
Figura 20 – Resultado obtido com a execução do código anterior	54
Figura 21 – Diagrama de classes da componente Renderizador do Pacote Interface .	58
Figura 22 – Código da classe Renderizador - Método par desenhar circunferências .	60
Figura 23 – Código da classe Renderizador - Método par desenhar vetores	60
Figura 24 – Código da classe Renderizador - Atributos	61
Figura 25 – Código teste do Renderizador - Chamada dos métodos de desenho . . .	61

Figura 26 – Resultado da execução do código apresentado na figura anterior	62
Figura 27 – Código do teste realizado para a classe Gerenciador Arquivo	65
Figura 28 – Resultado da execução do código apresentado na figura anterior - exibi- ção no console	66
Figura 29 – Resultado da execução do código apresentado na figura anterior - exibi- ção do arquivo onde os Objetos foram salvos	66
Figura 30 – Teste realizado com funções lambdas para criar uma função de ordem superior	71
Figura 31 – Resultado da execução do código apresentado na figura anterior - cha- mada às funções criadas a partir da função de ordem superior	71
Figura 32 – Comentário efetuado na classe Objeto que será compilado pelo Doxygen	74
Figura 33 – Resultado da documentação compilada apresentada na figura anterior .	75
Figura 34 – Elemento introdutório da documentação da classe Objeto gerado pelo Doxygen	76
Figura 35 – Menu de documentação listando todas as classes documentadas, bem como apresentando uma breve descrição conforme comentado no arquivo daquela classe	76
Figura 36 – Diagrama da classe Matriz - Classe muito importante e já citada no trabalho	78
Figura 37 – Recorte das componentes responsável pela colisão e movimentação de corpos na simulação	78
Figura 38 – Recorte da integração entre componentes de diferentes pacotes	79
Figura 39 – Diagrama da classe GerenciadorArquivo responsável pela persistência de dados no sistema	79
Figura 40 – Recorte das componentes (exceto CorRGBA) responsáveis pelo trata- mento de eventos da interface	80
Figura 41 – Imagem do repositório Gitlab finalizado com todos os elementos desen- volvidos durante o projeto para exploração	81
Figura 42 – Momento 1 da simulação final com diversos corpos movimentando-se aplicando tudo o que foi discutido anteriormente	82
Figura 43 – Momento 2 da simulação final com diversos corpos movimentando-se aplicando tudo o que foi discutido anteriormente	83

Lista de tabelas

Tabela 1	–	Etapas do trabalho divididas por atividade e tempo aplicado - Parte 1	10
Tabela 2	–	Etapas do trabalho divididas por atividade e tempo aplicado - Parte 2	11
Tabela 3	–	Componentes do Pacote Base - Parte 1	42
Tabela 4	–	Componentes do Pacote Base - Parte 2	43
Tabela 5	–	Componentes do Pacote Interface - Parte 1	56
Tabela 6	–	Componentes do Pacote Interface - Parte 2	57
Tabela 7	–	Componentes do Pacote Arquivo	64
Tabela 8	–	Testes realizados na etapa de integração de Pacotes	68

Sumário

0.1	Metodologia	8
0.2	Ferramentas	12
0.3	Objetivos	13
0.3.1	Objetivos Gerais	13
0.3.2	Objetivos Específicos	14
0.4	Implicações do Trabalho	14
1	DESENVOLVIMENTO	15
1.1	Estudo da Linguagem	15
1.2	Levantamento de Requisitos	17
1.3	Primeiro Contato com SDL	18
1.4	Explicação da Matemática e da Física	25
1.4.1	Aplicação da Matemática – Elementos estruturais	25
1.4.2	Aplicação da Física – Grandezas Vetoriais	26
1.4.3	Aplicação da Física – Movimento	27
1.4.4	Aplicação Matemática – Movimentação usando Vetores e Integração	28
1.4.5	Aplicação da Matemática – Métodos numéricos de Integração: O Integrador de Euler	29
1.4.6	Aplicação da Matemática – Circunferências e Verificação da Colisão entre Objetos e outros elementos	31
1.4.7	Aplicação da Física – Efeitos da Colisão entre Objetos, Colisões Elásticas, Conservação de Quantidade de Movimento e Reflexão Especular	33
1.4.8	Aplicação Matemática – Reposicionamento do Objeto após colisão com uma Linha	37
1.4.9	Aplicação Matemática – Matrizes e Transformações Geométricas	39
1.5	Desenvolvimento do Pacote Base	41
1.5.1	Discussão sobre a classe Objeto	44
1.5.2	Discussão sobre a classe DetectorColisao	48
1.6	Desenvolvimento do Pacote Interface	55
1.6.1	Discussão sobre a classe Renderizador	58
1.7	Desenvolvimento do Pacote Arquivo	63
1.8	Testes e Integração da Pacotes	67
1.9	Elementos extras que não estão no Trabalho Final	69
1.10	Documentação	73
1.11	Repositório e Simulação Final	81

2	CONCLUSÃO	84
2.1	Dificuldades	84
2.2	Trabalhos Futuros	85
2.3	Avaliação do processo de Aprendizagem	86
	REFERÊNCIAS	88

Introdução

O presente relatório tem como motivo de sua produção a apresentação do processo de desenvolvimento de um projeto prático para a disciplina de Programação Orientada a Objetos I do curso de Bacharelado em Ciências da Computação da turma de ingresso em 2020. Esse projeto executado ao longo do decorrer da disciplina citada tem como tema o desenvolvimento de um simulador físico.

0.1 Metodologia

A metodologia adotada para o desenvolvimento do projeto está fundamentada na realização de pesquisas para produção de uma base teórica sobre o tema que é usada como fundamentação para a produção prática do simulador físico. Dessa forma ao longo do projeto foi necessário a realização de diversas pesquisas em diversos materiais relacionados a programação, orientação a objetos, a linguagem C++ e fundamentos matemáticos e físicos. Como destaque para os materiais utilizados, pode-se citar livros sobre a linguagem C++, além de tutoriais online. Devido a grande correlação com a área física, foi necessária a consulta de livros da área. Uma vez tendo sido realizadas as pesquisas teóricas, era possível aplicar os conhecimentos adquiridos para a implementação de códigos visando a construção do simulador em partes.

Para facilitar o desenvolvimento do trabalho, e deixá-lo mais organizado e menos suscetível a complicações, foi necessário adotar uma abordagem dividida em etapas pertencentes em sua grande maioria de um ciclo, uma vez observado o fato dos elementos do trabalho serem interligados e a alteração de um aspecto facilmente pode causar a necessidade de aplicar mudanças a outra componente. Logo abaixo na figura 1 é possível observar uma representação gráfica da estruturação do desenvolvimento do trabalho.

Essas etapas fazem parte de um ciclo que perdurou ao longo da produção do trabalho todo. Vale detalhar que esse ciclo nunca possui um período exato pré-definido para ser completo, uma vez que optou-se por deixar o desenvolvimento um processo mais flexível e capaz de aceitar as diferentes situações vivenciadas ao longo do trabalho.

A fim de explicar essa metodologia cíclica, logo em seguida é possível conferir uma explicação sobre cada uma das etapas supra apresentadas.

- Definir Tema: Etapa inicial do ciclo. É nessa etapa que era definido o objeto de estudo e aplicação para o ciclo. Esse tema poderia ser um tema novo, ou algum aspecto já antes trabalhado, como da mesma forma não limitava-se a um único

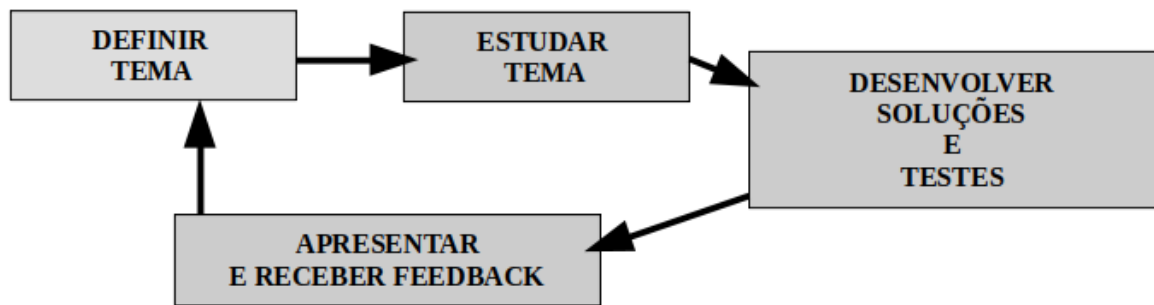


Figura 1 – Ciclo de Desenvolvimento aplicado no Projeto

Fonte: Produção Própria

elemento isolado do trabalho. Como será possível observar futuramente, o trabalho pode ser dividido em módulos, e geralmente esses eram usados como definição de um tema para o ciclo;

- **Estudar Tema:** Após ter sido escolhido o tema de aplicação do ciclo, era necessário estudar sobre ele. Como exemplo pode-se citar a necessidade de desenvolver um conhecimento sobre elementos físicos para poder aplicá-los ao trabalho. Nessa etapa fazia-se a produção teórica do trabalho que conforme citado fundamentava o desenvolvimento prático;
- **Desenvolver Soluções e Testes:** Uma vez compreendido e estudado o tema, era possível dar início ao desenvolvimento prático do trabalho, o qual está em sua grande parte relacionado ao desenvolvimento da programação do sistema, visto que esse refere-se ao processo mais recorrente no trabalho;
- **Apresentar e Receber Feedback:** Com uma série de materiais desenvolvidos, chegava a parte mais crítica do ciclo, a apresentação dos resultados e a obtenção de feedback do professor responsável pela disciplina. Semanalmente era disponibilizado um horário de atendimento para os alunos da disciplina entrarem em contato síncrono com o professor para discutir sobre o projeto. Era nesse momento que os erros e acertos podiam ser discutidos mais facilmente e um direcionamento para o próximo ciclo era dado.

A título de detalhamento do processo metodológico do trabalho, abaixo é possível conferir uma tabela em que as principais etapas de desenvolvimento estão apontadas, assim como é possível observar as principais atividades nelas desenvolvidas e resultados obtidos, bem como uma simples estimativa do tempo empregado para completar essa etapa.

Tabela 1 – Etapas do trabalho divididas por atividade e tempo aplicado - Parte 1

Etapa	Atividades Desenvolvidas	Tempo Percentual Aplicado
Definição do Tema do Trabalho e dos Aspectos Iniciais	Conversar com o professor para escolha do tema; Realizar levantamento de requisitos para o trabalho; Separar o projeto prático em componentes; Criar o Repositório Gitlab para o projeto;	5%
Desenvolvimento do Primeiro Contato com SDL	Estudar sobre a biblioteca SDL (Vídeos, Documentação, Tutoriais); Desenvolver uma série de testes com os principais conceitos e elementos da biblioteca para criação de interfaces gráficas básicas;	5%
Desenvolvimento da Base	Estudar e revisar sobre Orientação a Objetos e recursos da Linguagem C++ como sobrecarga de Operadores, Movimentação etc; Estudar e revisar sobre conceitos matemáticos e físicos como Vetores, Matrizes, Derivação, Integração e Integrador de Euler, Tipos de Movimento, Conservação de Movimento, Cálculo de Colisão, Reflexão especular etc; Desenvolver as classes básicas que permitem a estruturação dos elementos matemáticos e físicos que fundamentam a simulação; Desenvolver testes de funcionalidade para as classes criadas;	40%

Fonte: Produção Própria

Tabela 2 – Etapas do trabalho divididas por atividade e tempo aplicado - Parte 2

Etapa	Atividades Desenvolvidas	Tempo Percentual Aplicado
Desenvolvimento da Interface	Estudar e revisar aspectos da biblioteca SDL; Estudar e revisar aspectos sobre programação de interfaces gráficas e programação orientada a eventos; Estudar e revisar sobre aspectos de renderização de figuras como uso de matrizes de transformação, sistema de coordenadas computacionais e gráficas; Desenvolver as classes básicas que permitem a construção de uma camada de abstração a biblioteca SDL e tornam a construção da interface gráfica da simulação simplificada;	30%
Desenvolvimento da Persistência	Estudar sobre manipulação de arquivos em C++; Criar uma classe capaz de lidar com a persistência de dados para a Simulação; Criar um teste para a persistência de dados;	5%
Integração das Componentes	Criar uma série de pequenas simulações testes que interligam os elementos desenvolvidos na Base e na Interface; Corrigir erros proporcionados pelo processo de Integração;	8%
Documentação e Modelagem do Sistema	Documentar os códigos desenvolvidos; Modelar as componentes do Sistema; Criar o diagrama UML de Classes do Sistema;	7%

Fonte: Produção própria

Um ponto de ressaltar é que essas etapas não estão representadas em conformidade a uma sequência cronológica de desenvolvimento. Isso significa que a etapa 2 não foi iniciada somente após o término da etapa 1. Como fora dito anteriormente o trabalho foi operado em ciclos para construção do trabalho em partes que estão conectadas. Dessa forma o desenvolvimento de ciclos de diferentes etapas estão interligados, todavia para uma apresentação organizada de dados da metodologia do trabalho, faz-se necessário a divisão em etapas aparentemente modulares e fechadas como apresentado na tabela.

Da mesma forma deixa-se aqui esclarecido que essa será a maneira pela qual o detalhamento do desenvolvimento do trabalho será dado nesse relatório. Sendo assim tenha em mente que as etapas podiam muitas vezes estarem acontecendo simultaneamente.

0.2 Ferramentas

Para o desenvolvimento do trabalho não foi necessário o uso de ferramentas consideradas complexas, sendo que as necessidades do trabalho estão intimamente relacionadas ao ambiente acadêmico de um curso de Ciências da Computação. Logo em sequência é possível observar uma breve lista das ferramentas e tecnologias empregadas ao longo do processo de desenvolvimento do trabalho.

- Linguagem C++: Linguagem adotada pelo docente responsável pela disciplina para o desenvolvimento das atividades e do projeto. C++ É uma linguagem muito interessante de trabalhar uma vez que pode ser compreendida como um superconjunto da linguagem C onde têm-se recursos como alocação de memória dinâmica herdada do C junto de recursos da Orientação a Objetos adicionados pelas implementações C++;
- Compilador g++: Compilador padrão da distribuição Linux Ubuntu 20.04 utilizada ao longo do desenvolvimento do trabalho. A título de especificação, o projeto foi compilado usando-se da versão C++14;
- Biblioteca SDL: É uma biblioteca escrita em C que funciona nativamente em C++ e foi usada para proporcionar acesso de baixo nível a hardware de vídeo, som, teclado, mouse etc. A título de especificação, o projeto faz uso da versão 2.0.10;
- Doxygen: É uma ferramenta que permite a escrita de comentários no código C++ que futuramente podem ser compilados e gerar uma documentação para o projeto. A título de especificação, o projeto faz uso da versão 1.8.17;
- Gitlab: É um gerenciador de repositório de software que foi implementado baseado no git e permite que o armazenamento, o versionamento e o compartilhamento de códigos seja mais fácil. O perfil e o repositório usado para armazenar os códigos e

outros documentos do projeto pode ser acessado através do link: <<https://gitlab.com/Lima001/bcc-projeto-poo>>;

- Dia: É um software de diagramação simples que foi utilizado para a modelagem de diagramas de classe para o projeto.
- Geogebra: É um software de interface gráfica que implementa recursos matemáticos e permite a simulação de certos cálculos. Foi usado ao longo do projeto para modelagem de situações necessárias de implementação no projeto com grande rigor matemático, como as colisões;
- Visual Studio Code: É um editor de código fonte simples e foi utilizado para a escrita dos códigos do projeto;
- LibreOffice Write: Editor de texto nativo do sistema Linux Ubuntu que foi utilizado para a criação de material em formato texto, seja para apresentação de resultados ou documentação do projeto;
- Chromium: Navegador web utilizado para realizar o acesso aos endereços em que os materiais de estudo estão disponíveis;
- YouTube: Plataforma de vídeo utilizada para aperfeiçoamento dos conhecimentos nos assuntos relacionados ao projeto;

0.3 Objetivos

Os objetivos do trabalho podem ser divididos em dois tipos, sendo eles objetivos gerais e objetivos específicos. Logo abaixo é possível conferir os objetivos elencados para o trabalho de desenvolvimento do simulador físico, bem como um breve detalhamento de cada um desses objetivos.

0.3.1 Objetivos Gerais

- Aplicar conceitos de orientação a objetos no desenvolvimento de uma aplicação gráfica utilizando C++ e SDL2;
- Apresentar um projeto prático para fins avaliativos na disciplina de Programação Orientada a Objetos I do curso de Bacharelado em Ciências da Computação ofertado pelo Instituto Federal Catarinense Campus Blumenau;
- Aprimorar os conhecimentos em programação, orientação a objetos, linguagem C++, SDL2 e conteúdos da matemática e da física.

0.3.2 Objetivos Específicos

- Construir um projeto de simulação computacional simples aplicando conhecimentos matemáticos e físicos como integração numérica, conservação de movimento e transformações geométricas;
- Dividir o projeto em componentes/pacotes separando a programação dos aspectos matemáticos/físicos da simulação, da implementação dos recursos de interface para o usuário para que esses possam ser reutilizados;
- Implementar uma camada simples de abstração a biblioteca SDL2;
- Apresentar documentação e modelagem simples do sistema para auxiliar o seu uso compartilhado, ou possível reaproveitamento de código;

0.4 Implicações do Trabalho

O projeto do simulador físico pode ser avaliado como importante, pois permite em um primeiro momento a aplicação prática dos conhecimentos estudados ao longo da disciplina de Programação Orientada a Objetos I, bem como fundamenta o processo de busca do conhecimento. Por se tratar de um projeto de simulação física, além dos próprios conhecimentos pertinentes a programação, esse trabalho tem um considerável potencial de desenvolvimento do conhecimento matemático e físico.

Sendo dessa forma, estudar e desenvolver um trabalho como esse desenvolve o caráter científico de um aluno. Projetos de programação que aplicam explicitamente a matemática e a física são grandes ferramentas passíveis de serem utilizadas para demonstrar a importância desse tipo de conhecimento e como esse pode ser aplicado. Além disso o próprio tema de simulação instiga o aluno a desenvolver e estudar ações do mundo real. Mesmo que o domínio do projeto apresentado seja algo extremamente reduzido, esse pode ser usado para estimular alunos a buscarem estudar os problemas do mundo real e abordar possíveis soluções.

No tempo em que esse relatório está sendo escrito, passaram-se aproximadamente um ano do início da pandemia de 2020. Nesse cenário onde o ceticismo é tamanho, sendo capaz de colocar em xeque a vida das pessoas, o desenvolvimento científico deve ser continuado e sua validade deve ser usada de fundamento para o avanço social. Uma maneira muito resumida e simples de atingir algo do tipo é através da demonstração de que esse conhecimento pode sim descrever a realidade, como é o caso do que acontece em simulações.

1 Desenvolvimento

Conforme fora citado na seção de metodologia aplicada ao desenvolvimento do trabalho, o projeto pode ser dividido em diversas etapas. Essas etapas podem ou não ter acontecido de maneira simultânea umas as outras. Visando facilitar a organização desse relatório, bem como a compreensão do leitor sobre as passagens efetuadas ao longo do trabalho, os próximos assuntos estão divididos em tópicos que relatam as atividades pertinentes ao trabalho.

Alguns desses tópicos referem-se a um processo contínuo que desenvolveu-se ao longo de todo o período de produção do projeto, como é caso do estudo da linguagem. Já outros referem-se a atividades mais isoladas a um período específico, como acontece com o levantamento de requisitos. Observado isso, ao longo da abordagem de cada tópico será informado explicitamente como esse foi desenvolvido e os resultados e impactos obtidos através de sua conclusão.

1.1 Estudo da Linguagem

Uma dos processos mais fundamentais e pertinentes a todo projeto, seja esse prático ou teórico, é o constante processo de pesquisa e aprendizagem que faz-se necessário. Ao longo de um projeto sempre será necessário aprofundar-se e buscar diversos conhecimentos para que o projeto tenha uma continuidade fluida.

Ao longo do desenvolvimento do Simulador Físico é possível observar um processo contínuo e que ocorre de maneira simultânea a todas as outras etapas. Esse processo refere-se ao estudo da linguagem de programação C++. Inicialmente o autor não possuía um conhecimento teórico-prático adequado da linguagem para a produção de um sistema relativamente complexo como um simulador computacional.

Devido a esse fato, foi necessário aplicar uma certa carga horária para aprender os recursos e compreender o funcionamento dos principais tópicos da programação em C++ e da Orientação a Objetos da Linguagem. Para alcançar esse objetivo destaca-se a necessidade de consultar e estudar materiais teóricos sobre o tema, dando destaque principalmente para alguns recursos sendo eles:

- Pesquisa em sites desenvolvidos especificamente para a abordagem da linguagem C++, como é o caso dos sites <cplusplus.com> e <learncpp.com>. Nessas páginas é possível encontrar uma grande quantidade de conteúdo relacionado a linguagem C++ apresentada de maneira simples, operando como fortes introduções ao tema,

garantindo uma visão inicial dos principais aspectos da linguagem;

- Leitura de referências bibliográficas citadas pelo docente da disciplina, principalmente o livro *The C++ Programming Language* escrito por Bjarne Stroustrup, idealizador e conhecido como o pai da linguagem. Por ser um livro escrito com um detalhamento mais profundo do que o encontrado nos tutoriais online citados anteriormente, esse material fez-se importante para a sedimentação e lapidamento dos conhecimentos inicialmente adquiridos, bem como serviu de consulta para as mais diversas necessidades relacionadas a programação que surgiram ao longo do trabalho.

Além do processo de pesquisa e estudo para formação de conhecimento, foi necessário aplicar esse conhecimento para sua consolidação e exploração dos possíveis pontos críticos, isso é, aspectos que não foram compreendidos de maneira adequada às necessidades do projeto. Como destaque para esse processo pode-se citar a realização de listas de exercícios de programação na plataforma URI conforme solicitação do docente responsável pela disciplina, bem como exercícios encontrados nos tutoriais anteriormente citados.

Uma vez tendo sido aprendido os principais conceitos introdutórios da linguagem, além dos aspectos relacionados a Orientação a Objetos, foi possível dar início ao processo de desenvolvimento prático do trabalho e mantê-lo ao longo do período de desenvolvimento. Como resultado foi possível obter não somente a produção dos códigos (processo detalhado em seções futuras), mas também um conhecimento considerável da linguagem que abre um leque de oportunidades para futuras ocasiões como a produção de um novo trabalho usando-se a linguagem ou uma linguagem similar como C.

Visando especificar um pouco mais esse processo do trabalho, deixa-se logo em seguida uma lista sumarizada dos principais tópicos da linguagem C++ que foram estudados durante esse processo. Esse detalhamento serve também como uma apresentação dos resultados adquiridos com essa atividade continuamente realizada. São alguns dos conhecimentos adquiridos em certo nível e que são julgados importantes.

- Teoria da Linguagem: História, conceitos bases, processo de compilação;
- Básico de Programação em C++: Operadores, tipos de dados, estruturas de repetição e condicionais;
- Alocação e desalocação de memória dinâmica;
- Ponteiros e referências, ponteiros para ponteiros e ponteiros para funções;
- Estruturas de dados previstas em C++: Struct, vector, enum;
- Básico de Orientação a Objetos em C++: Classes e objetos, herança, encapsulamento;

- Sobrecarga de operadores, sobrescrita de métodos, construtores e operadores de Atribuição usando-se do recurso de Cópia e Movimentação;
- Manipulação de Arquivos, camadas de entrada e saída de dados;

1.2 Levantamento de Requisitos

O processo de levantamento de requisitos é uma das etapas mais importantes no processo de pré desenvolvimento de um software, observado o fato de que é atribuído aos requisitos de um sistema modelar e ditar como esse deverá se comportar ao longo do processo de seu desenvolvimento. Todavia diferentemente do processo que ocorre no levantamento de requisitos em projetos empresariais, ou até mesmo para fins acadêmicos em disciplinas mais internas a um curso de computação, ao abordar-se o contexto do trabalho desenvolvido não será possível observar todo o rigor imposto a esse processo.

Simplificando, o processo de levantamento de requisitos para o projeto ocorreu de uma forma muito mais simples do que por exemplo a abordagem estudada em disciplinas de engenharia de software. Isso acontece devido ao fato da disciplina na qual o projeto foi desenvolvido possuir um caráter mais introdutório do curso de computação, onde o foco está na aprendizagem do paradigma de Orientação a Objetos e o desenvolvimento de um projeto usando-se desses conceitos adjuntos da linguagem C++. Os requisitos nesse contexto servem para guiar o projeto e permitir ao professor avaliar a integridade do desenvolvimento desse através da conclusão daquilo que fora definido inicialmente como requisitos do trabalho.

Uma vez compreendido esse fato, é possível discutir como se deu o processo de levantamento desses requisitos, bem como os resultados obtidos nessa etapa iniciada e finalizada no período inicial do trabalho.

No que tange a escolha dos requisitos para o projeto, a atividade que foi desenvolvida consistia na discussão do tema com o professor responsável pela disciplina. Visto que o tema já havia sido declarado como um Simulador Físico e que o autor do trabalho, bem como o próprio professor haviam participado de um projeto de extensão realizado online sobre introdução a programação e simulação, o levantamento de requisitos do sistema pode ser visto como uma tarefa relativamente simples.

Após uma reunião online síncrona em horário disponibilizado pelo professor, foi possível chegar a um consenso onde os seguintes requisitos foram definidos para o trabalho:

- Desenho de plano cartesiano;
- Desenho de vetores de grandezas vetoriais (velocidade, aceleração, forças);
- Fazer simulações envolvendo Forças e Conservação de Movimento;

- Detectar colisões entre objetos de formas circulares;
- Interface para simulação;
- Salvar dados da simulação em arquivo;

De maneira geral pode-se resumir que os requisitos levantados referenciam a necessidade do projeto possuir um caráter físico e matemático bem definido, uma vez que o tema simulador físico está intrinsecamente fundamentado sobre esses elementos. Além disso é possível observar certos requisitos relacionados com a proposta da disciplina como a necessidade de uma interface e da persistência de dados no sistema e que fazem-se indispensáveis para a construção de um projeto com um nível aceitável de qualidade.

1.3 Primeiro Contato com SDL

Em conformidade com o que já fora comentado anteriormente, o desenvolvimento do trabalho necessitava da produção de uma interface gráfica para as simulações. Segundo previsto pelo plano de ensino do professor responsável pela disciplina, era necessário fazer o uso da biblioteca SDL em conjunto de C++ para a implementação de recursos de interface com o usuário.

Devido ao fato do autor não possuir conhecimento prévio acerca da biblioteca, este estava incapacitado de produzir uma camada de abstração a biblioteca para uso em conjunto dos elementos de simulação. Objetivando solucionar essa falta de conhecimento, uma das etapas desenvolvidas inicialmente no projeto foi criar um pequeno contato com as funcionalidades básicas SDL para formação de um conhecimento introdutório capaz de desenvolver as discussões sobre o tema ao longo do desenvolvimento do simulador físico.

Esse primeiro contato estruturou-se na forma de pesquisas a documentação da biblioteca, bem como o acesso a tutoriais em vídeos que apresentavam a implementação de interfaces simples mas capazes de utilizar os principais recursos para iniciar um projeto com SDL. Uma vez tendo sido adquirido um conhecimento básico acerca da biblioteca, foi dado início a um processo de desenvolvimento de códigos que testavam os recursos da biblioteca.

Os testes desenvolvidos não foram criados com base no paradigma Orientado a Objetos, visto os objetivos desses, mas foram implementados em uma programação estruturada mais simples capaz de evidenciar os elementos da biblioteca. Ao todo foram desenvolvidos quatro arquivos de teste, cada um sendo uma evolução do arquivo que sucede. Dessa forma ao longo dos testes é possível observar um avanço no que tange a exploração de recursos.

Todos esses códigos estão disponibilizados no repositório Gitlab do projeto, na pasta extras e na sua subpasta Básico SDL. Logo abaixo é possível conferir a 2 que demonstra a exibição do arquivo README.md criado para essa etapa de desenvolvimento que apresenta o trabalho desenvolvido.

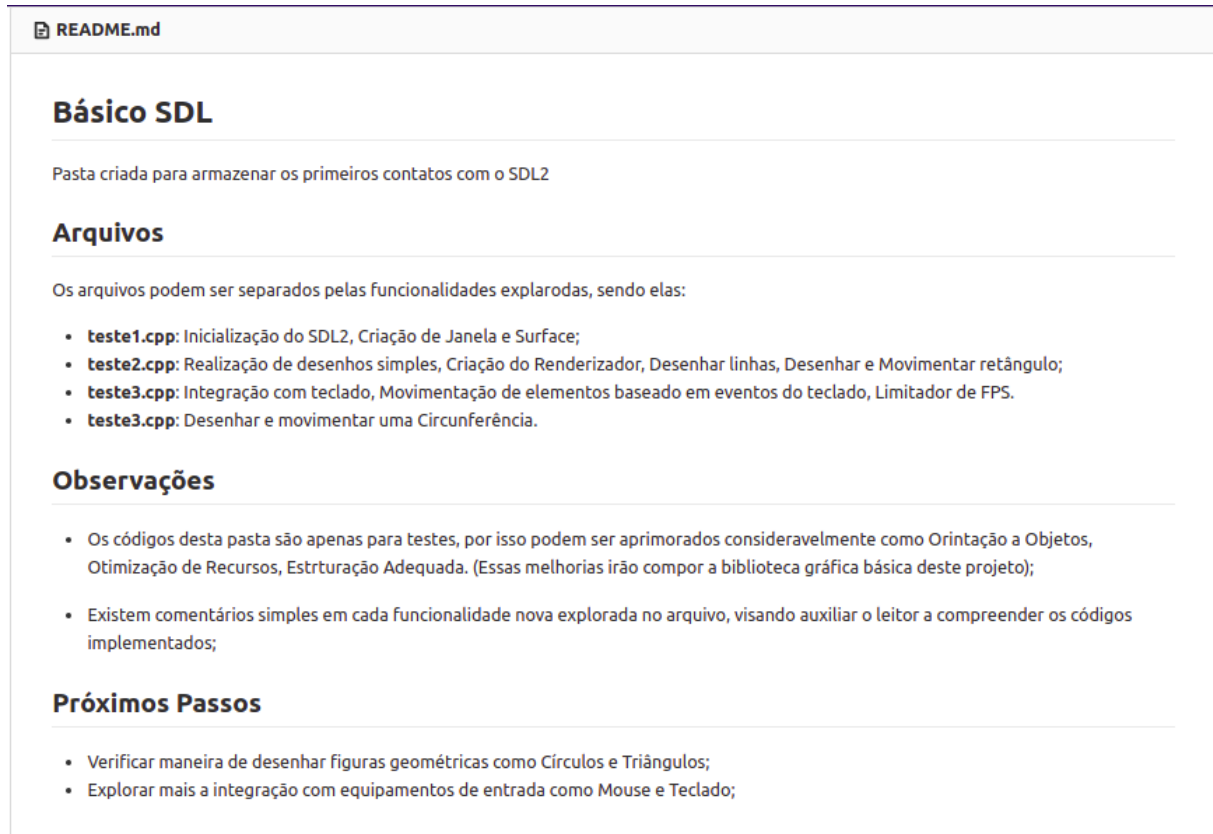


Figura 2 – Exibição do conteúdo do arquivo README.md presente no diretório Básico SDL

Fonte: Produção Própria

Conforme é possível observar na figura, os arquivos criados abordam desde os aspectos mais básicos da biblioteca e da sua aplicação como a inicialização dos recursos e criação de uma Janela e uma Área de Renderização, até controle de FPS e movimentação de figuras usando-se de tratamento de eventos do teclado.

Além disso pode-se destacar os apontamentos efetuados no tópico Observações, onde explicita-se o caráter básico e de teste dos códigos criados, bem como a capacidade de aprimoramento através da orientação a objeto – aspecto que está presente no trabalho e caracteriza-se como o Desenvolvimento do Pacote Interface.

Devido aos aspecto compartilhado do trabalho, uma vez que os alunos da disciplina possuíam liberdade prevista pelo professor de acessar e compartilhar componentes de códigos entre si, juntamente do caráter avaliativo do projeto, os códigos desenvolvidos contam com pequenos comentários para situar o leitor do que está acontecendo em cada etapa fundamental dos testes. Esses comentários são interessantes pelo fato de representarem as anotações de um aluno durante o seu processo de estudo e desenvolvimento do conhecimento de um tema, similar ao que acontece em aulas teóricas onde usa-se das folhas de papel para aprender.

Dando sequência a essa seção, é possível logo mais observar algumas figuras que apresentam trechos dos códigos desenvolvidos, possibilitando compreender como esses foram implementados, além de demonstrar os resultados do trabalho prático aplicado à essa etapa.

```
6 using namespace std;
7
8 int main(){
9
10     //Variáveis que determinam características da tela
11     int largura_tela = 800;
12     int altura_tela = 600;
13     char titulo[] = "Basico SDL";
14
15
16     SDL_Window* janela = NULL; // Variável que apontará para a janela criada pelo SDL
17     SDL_Surface* tela = NULL; // Variável que apontará para a tela ("onde podemos desenhar")
18
19     SDL_Event evento;          // Variável que guardará os eventos registrados pelo SDL
20
21     bool executar = true;      // Variável do loop de execução principal
22
23
24     /*
25      Inicializa toda biblioteca e verifica se o processo ocorreu
26      sem erros. Em caso de erros, Imprime uma mensagem indicando
27      o erro identificado e finaliza a biblioteca.
28     */
29     if (SDL_Init(SDL_INIT_EVERYTHING) < 0){
30         cout << "Erro ao Iniciar SDL - " << SDL_GetError() << endl;
31         SDL_Quit();
32     }
33 }
```

Figura 3 – Exibição da declaração de variáveis SDL e inicialização da biblioteca

Fonte: Produção Própria

```

14 struct circunferencia{
15     ponto centro;
16     int raio;
17     /*
18      * Array que guarda pontos da circunferência
19      * para que retas possam ser traçadas e formar
20      * uma circunferência.
21      *
22      * Observação: Quanto maior for a quantidade escolhida,
23      * mais preciso a circunferência tende a ficar graficamente.
24      */
25     ponto demarcacao[PRECISAO];
26     // int precisao; // Define a quantidade de pontos a serem usados na demarcacao
27 };
28
29 void calcularDemarcacao(circunferencia *c){
30     /*
31      * Acha pontos pertencentes a circunferência para
32      * que possamos desenhar o contorno dessa circunferência
33      * futuramente com a função desenharBordaCircunferencia();
34      */
35
36     ponto p;
37     // Calcular angulo a partir da variável de precisão
38     float angulo = 360/PRECISAO;
39     // Necessário Otimizar para não ser necessário tantos cálculos
40     for (int i=0; i<PRECISAO; i++){
41         p.x = c->centro.x + cos((angulo*i) * (M_PI/180.0f))*c->raio;
42         p.y = c->centro.y + sin((angulo*i) * (M_PI/180.0f))*c->raio;
43         c->demarcacao[i] = p;
44     }
45 }
46
47 void desenharPontosCircunferencia(circunferencia *c, SDL_Renderer *renderizador){
48     /*

```

Figura 4 – Exibição da estrutura para representar uma circunferência, bem como o cálculo para gerar ela graficamente

Fonte: Produção Própria

```
144      /*  
145          Chamada dos Métodos para desenhar a circunferência;  
146      */  
147      // desenharPontosCircunferencia(&circ, renderizador);  
148      desenharBordaCircunferencia(&circ, renderizador);  
149  
150      while (SDL_PollEvent(&evento)){  
151  
152          if (evento.type == SDL_QUIT){  
153              executar = false;  
154              break;  
155          }  
156          /*  
157              Detectar entrada via teclado para movimentar a circunferência  
158          */  
159          else if (evento.type == SDL_KEYDOWN){  
160              /*  
161                  Detectar qual tecla foi pressionada para  
162                  aí poder modificar a posição do centro da  
163                  circunferência, alterando assim a sua posição  
164              */  
165              switch (evento.key.keysym.sym){  
166  
167                  /*  
168                      Chamar função calcularDemarcacao();  
169                      Necessário recalcular as demarcações!  
170                      Como o centro não é mais o mesmo, as  
171                      'bordas' também já não são mais as mesmas.  
172                  */  
173                  case SDLK_LEFT:  
174                      circ.centro.x-=movimento;  
175                      calcularDemarcacao(&circ);  
176                      break;
```

Figura 5 – Exibição do tratamento de eventos simples

Fonte: Produção Própria


```
99
100     SDL_RenderPresent(renderizador);
101
102     SDL_UpdateWindowSurface(janela);
103
104     // Limitação do FPS do loop de execução
105     // Verifica se o tempo que a execução levou é maior
106     // que o permitido em um segundo. Caso seja, atrasa
107     // a execução no tempo que excedeu o máximo de FPS
108     if ((1000 / FPS) > SDL_GetTicks() - clock_inicial){
109         SDL_Delay(1000 / FPS - (SDL_GetTicks() - clock_inicial));
110     }
111 }
112
113 SDL_DestroyRenderer(renderizador);
114 SDL_DestroyWindow(janela);
115 SDL_Quit();
116
117 return 0;
118 }
```

Figura 6 – Exibição do cálculo de FPS e as chamadas de finalização da biblioteca ao final do programa

Fonte: Produção Própria

1.4 Explicação da Matemática e da Física

Através de uma análise do tema do trabalho percebe-se que esse está totalmente relacionado a aplicação de conceitos matemáticos e físicos na programação para a construção de um software. Esse é o aspecto central do projeto desenvolvido, uma vez que sem esse tipo de conhecimento não seria possível a implementação de um simulador físico. Dessa forma, uma vez observado a íntima correlação do projeto com essas áreas, faz-se necessário apresentar de maneira geral e concisa os principais conceitos aplicados no projeto.

Ao longo desse trecho do relatório não tem-se a intenção de discutir aspectos da programação no que se refere a implementação de elementos do simulador, mas sim abordar toda a teoria básica que estrutura o simulador desenvolvido. No que tange aos aspectos de programação, certos detalhes são abordados nesse relatório nas próximas seções e podem também serem analisados inteiramente através do acesso ao código fonte disponibilizado no repositório Gitlab informado em momentos iniciais desse relatório.

Para facilitar a compreensão e deixar a apresentação do conteúdo mais organizado, a atual seção é dividida em partes sendo elas principalmente estruturadas em: Aplicação da Matemática e Aplicação da Física. Ao longo do desenvolvimento dessa seção do relatório é possível observar uma intercalação entre essas áreas, uma vez que para o desenvolvimento do projeto é necessário uma visão ampla dos tópicos como um todo interligado.

1.4.1 Aplicação da Matemática – Elementos estruturais

A matemática é a base do simulador físico construído. Sem ela e suas ferramentas não seria possível construir de maneira adequado os elementos que compõem o projeto. De maneira geral os elementos que são apresentados nesse tópico são implementados como classes no sistema e compõem o pacote Base – discutido em seções posteriores – além de serem usados em processos relacionados a renderização de figuras na interface gráfica do simulador.

Para dar início a essa abordagem, apresenta-se abaixo os elementos mais básicos em complexidade de implementação. Esses elementos podem ser observados como a base para outros recursos do simulador.

- Ponto: Os pontos são elementos fundamentais da geometria, uma vez que ele pode ser usado para representar algo em um plano ou espaço. Todavia um aspecto extremamente interessante sobre os pontos é o fato deles serem utilizados para definirem outros elementos da matemática, como os Vetores e as Figuras Geométricas. Na simulação a sua aplicação segue esse pensamento matemático e dessa forma é possível observar pontos sendo usado para estruturar outros elementos do simulador;

- Vetor: Os vetores são classes extremamente fundamentais para representação e execução de diversos cálculos. Através do uso desses elementos é possível definir grandezas físicas não escalares aos objetos da simulação, bem como realizar diversos cálculos vetoriais para colisão por exemplo;
- Figuras Geométricas: As figuras geométricas são aspecto fundamentais da geometria. São elementos compostos de propriedades e podem ser utilizados para representar alguma forma. No caso do projeto, as figuras geométricas são utilizadas para representar o formato que os objetos da simulação possuem – formato que implica diretamente nos cálculos envolvendo o objeto, e que depois pode ser interpretada e representada graficamente;
- Plano Cartesiano: É um sistema de coordenadas formado pelos eixos da abcissa e da ordenada. É extremamente fundamental, pois é o local onde pode-se trabalhar com todos os elementos citados anteriormente e permite uma compreensão mais gráfica de certos aspectos matemáticos. No simulador é utilizado para representar o local da simulação, bem como implicitamente embasar todas as outras implementações de recursos matemáticos.

1.4.2 Aplicação da Física – Grandezas Vetoriais

Existem certas situações na física em que deseja-se descrever alguma situação ou algum sistema onde o simples uso de um escalar não é capaz de representar o sentido completo dessa descrição. Nesses casos muito que provavelmente essa situação refere-se a uma grandeza vetorial. Esse tipo de grandeza são expressas fazendo o uso de vetores, e por consequência possuem agregadas a si propriedades como módulo, direção e sentido.

Um exemplo de grande vetorial é a aplicação de uma força sobre um corpo. Nesse caso, nas situações comuns não basta simplesmente informar a intensidade dessa força. Se alguém informar que um corpo está sendo puxado por uma força com intensidade equivalente a 400N, não será possível concluir aspectos básicos desse sistema, como por exemplo para onde essa força está arrastando esse corpo? É no sentido Horizontal? É no sentido Vertical? É nos dois, e qual a intensidade em cada eixo? Essas informações estão associadas ao uso de vetores.

Para o desenvolvimento do simulador físico algumas propriedades dos objetos da simulação que representam os corpos que compõem a simulação necessitam ser especificadas utilizando-se de vetores, como é o caso da velocidade de um objeto. Usando-se de vetores é possível definir velocidades em todas as direções e sentidos de movimento no plano, tornando a simulação algo mais concreto. Além da velocidade é possível definir também a posição do corpo no plano como um vetor. Essa abordagem permite uma fácil manipulação

algébrica para realizar cálculos relacionados a movimentação de um corpo, uma vez que os vetores possuem agregados a si um conjunto bem definido de operações.

1.4.3 Aplicação da Física – Movimento

Na física existem diversos tipos de movimentos que regem como um corpo se comporta. A construção do simulador físico está totalmente baseado na capacidade de definir o movimento para os objetos da simulação para ver como esses se comportam ao longo de colisões. No que tange respeito a física existem alguns tipos de movimentos que podem ser definidos como é o caso do movimento uniforme e uniformemente variado.

Entretanto para o desenvolvimento do projeto foi necessário utilizar não somente um tipo de movimento, mas sim um conjunto de movimentos permitindo maior flexibilidade para a simulação. A título de especificação logo abaixo estão elencadas as possibilidades de movimentos na simulação.

- Movimento Uniforme: Esse movimento ocorre quando um corpo não sofre efeito de aceleração, movimentando-se a uma velocidade constante. O cálculo da posição de um corpo que movimenta-se dessa forma pode ser dado pela fórmula: $Sf = Si + Vt$, onde tem-se que a posição final Sf é igual a posição inicial Si mais a velocidade constante V multiplicada pelo tempo t do movimento;
- Movimento Uniformemente Variado: Esse movimento ocorre quando um corpo está movimentando-se sobre a ação de uma aceleração constante, variando dessa forma a sua velocidade em taxas uniformes. Ao tratar desse tipo de movimento pode-se descobrir algumas informações como a posição de um corpo em um determinado tempo e sua velocidade através de fórmulas como:

Para achar a velocidade

$$V = Vi + at$$

Onde têm-se que V é a velocidade a ser descoberta, Vi é a velocidade inicial do corpo, a é a sua aceleração constante e t representa o tempo;

Para achar a posição

$$Sf = Si + Vit + \frac{a}{2} * t^2$$

Onde têm-se que Sf é a posição final, Si é a posição inicial, Vi é a velocidade inicial, t representa o tempo e a a aceleração constante do corpo

- Movimento com Aceleração Variada: Esse movimento ocorre quando um corpo está movimentando-se sobre a ação de uma aceleração que varia ao longo do tempo. Nesses casos a aceleração pode estar descrita sobre a forma de uma função em relação ao tempo. O cálculo de informações como posição e velocidade nesses casos requer um certo recurso matemático mais avançado que é abordado no tópico seguinte.

Para a simulação desenvolvida, optou-se por permitir a definição do movimento com Aceleração Variada para os objetos através da definição de funções. Essa abordagem é interessante, pois ele não limita os outros dois tipos de movimento citados anteriormente. Dessa forma sendo capaz de calcular a velocidade e posição de um objeto que possui movimento com Aceleração variada, a implementação é capaz de calcular essas informações para movimentos mais simples.

1.4.4 Aplicação Matemática – Movimentação usando Vetores e Integração

Uma vez tendo sido compreendido os tipos de movimentos, pode-se explicar como o seu cálculo ocorre matematicamente. Para iniciar é preciso definir como um corpo é definido – ao menos em questões relacionadas ao movimento – na simulação. De maneira simples é possível expressar a movimentação de um corpo através da mudança de sua posição ocorrendo conforme uma velocidade definida que pode ou não ser variada a depender da aceleração sofrida pelo corpo.

A posição e a velocidade podem ser representadas através de um vetor localizado no plano cartesiano. Para variar a posição e por consequência representar a movimentação em um intervalo de tempo para um corpo na simulação basta somar a velocidade obtida em um período do tempo à posição. Nos casos mais simples de movimento uniforme é praticamente isso que as equações representam. Todavia para a simulação ser mais interessante foi definido que deve-se considerar movimentos em que a aceleração varia. Nesses casos não é possível utilizar as equações, visto que a aceleração está em constante mudança.

Para resolver esse problema e permitir que o cálculo da movimentação funcione para casos mais complexos e por consequência para os casos simplificados obtidos a partir desse pensamento, usa-se o recurso de integração da aceleração. Para compreender esse aspecto é necessário comentar brevemente a relação entre posição, velocidade e aceleração na visão do cálculo diferencial e integral.

Conhecendo-se a função que define a posição de um corpo em relação ao tempo é possível obter a sua velocidade em relação ao tempo através da derivação da posição. Repetindo-se o processo mas dessa vez derivando a velocidade em relação ao tempo, é possível conhecer a aceleração que aquele corpo está sofrendo sendo ela constante ou não.

Para exemplificar esse aspecto considere o exemplo abaixo e observe como esse processo é tecnicamente simples.

Considere $S(t)$ como a função da posição. A derivada de $S(t)$ corresponde a função da velocidade que aquele corpo está sob efeito e a derivada da sua velocidade informa a sua aceleração. Confira abaixo os cálculos de derivação para encontrar a aceleração a partir da posição de um corpo

$$S(t) = 5x^3 - 4x^2 + 10$$

$$S'(t) = V(t) = 15x^2 - 8x$$

$$V'(t) = A(t) = 30x - 8$$

Dessa forma é possível compreender e definir a movimentação de um corpo a partir de sua posição. Entretanto na simulação o que acontece é que existe apenas um conhecimento de como a aceleração está definida através de uma função. Nesse caso é necessário descobrir as informações da velocidade e posição do corpo, o contrário do exemplo anterior. Felizmente é possível realizar esse tipo de operação através da integração da aceleração em um período do tempo. Realizando essa operação de integração da aceleração considerando um período de tempo, é possível descobrir a aproximação da velocidade nesse período de tempo. Repetindo-se o processo mas considerando dessa vez a velocidade descoberta para a integração, é possível também achar os valores para a posição do corpo. Nesses casos não importa se o corpo movimenta-se com uma aceleração variada ou constante, podendo inclusive ser nula. O processo de integração ainda será capaz de resolver o problema da movimentação.

1.4.5 Aplicação da Matemática – Métodos numéricos de Integração: O Integrador de Euler

Uma vez tendo sido compreendido como é possível movimentar um corpo a partir do conhecimento de sua aceleração, faz-se necessário entender como realizar esse processo de forma que seja possível replicá-lo computacionalmente. Para isso foi necessário abordar e utilizar um método numérico que visa aproximar o resultado da integração.

O método numérico utilizado foi o método de Euler que calcula a integral em um determinado período através da decomposição da área da curva da função em diversas outras áreas menores de forma que essas possam ser aproximada a uma forma retangular, visto que a área de retângulo é muito mais simples de calcular do que a área de uma curva, sendo somadas ao final e resultando uma aproximação da área analisada. Uma representação dessa aplicação pode-se observada na figura 7, onde cada retângulo representa uma aproximação para a área da função naquele intervalo. No que tange as cores verde e amarela, essas estão sendo usadas para diferenciar as duas possíveis aproximações.

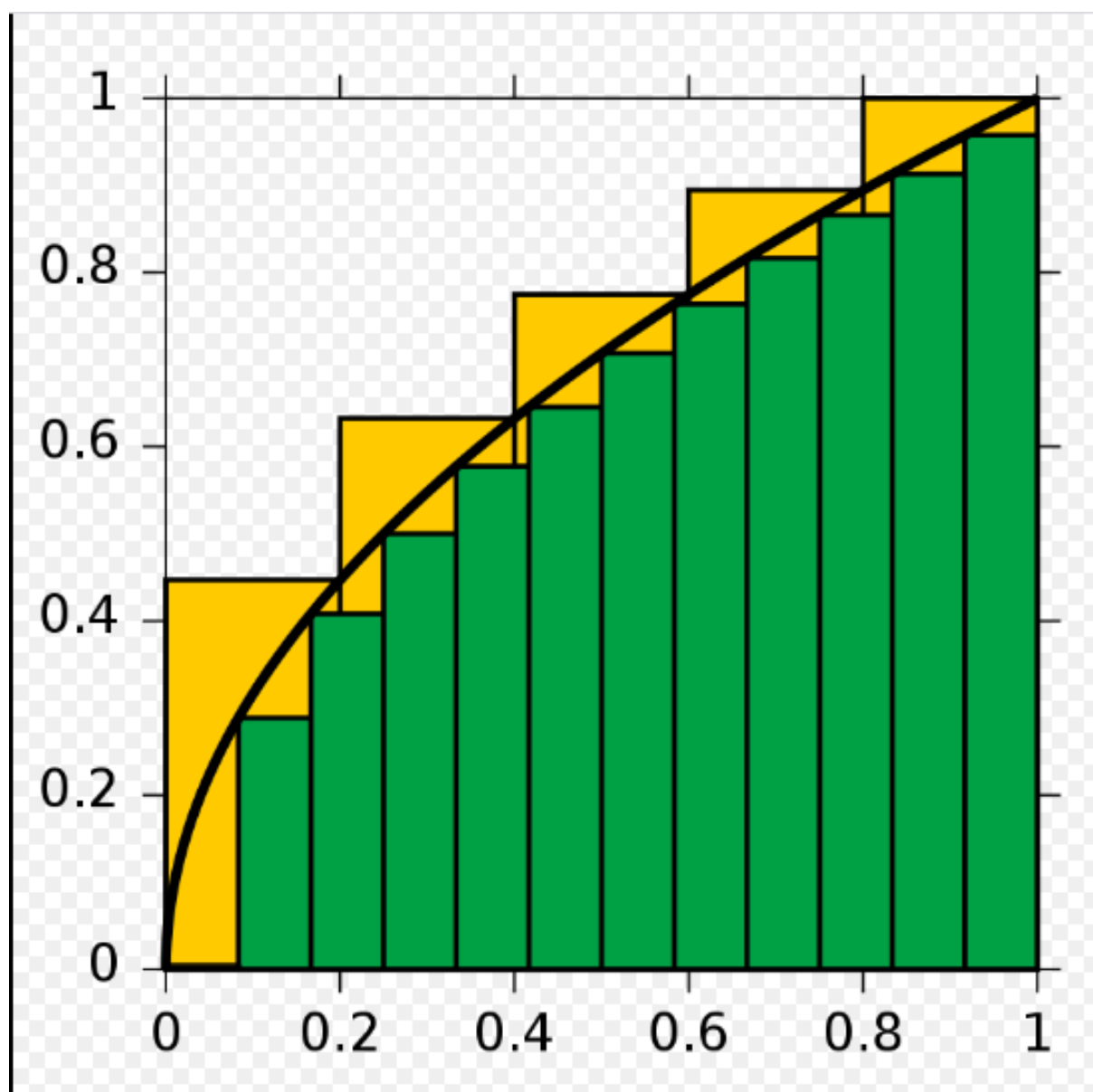


Figura 7 – Aproximação numérica da integração a partir da soma de áreas retangulares

Fonte: <<https://pt.wikipedia.org/wiki/Integral/>>

A formulação para esse cálculo é relativamente mais complexa e não é apresentada com o devido rigor nesse relatório, todavia utilizando-se da representação esquemática do cálculo apresentado anteriormente, bem como uma simplificação da formulação matemática que é apresentada logo abaixo é possível compreender o aspecto básico desse cálculo.

Para calcular numericamente a Integral de uma função em um determinado período pode-se realizar o seguinte cálculo considerando a seguinte perspectiva:

$$V = h * f(t)$$

Onde têm-se que V é a aproximação da velocidade em um intervalo de tempo - é um retângulo da representação esquemática vista anteriormente. h refere-se ao passo do integrador. Essa medida refere-se na medida da base do retângulo e corresponde com a divisão do intervalo em que deseja-se calcular a integral pelo nível de precisão considerado. $f(t)$ é o valor da função aceleração no tempo t que essa está sendo calculada. O tempo é controlado através do controle dos passos dados. Se você sabe que dividiu um intervalo que vai de 5 a 10 em 5 partes inteiras, você sabe que cada passo/instante de tempo vai ser correspondente aos números 6,7,8,9 e 10. Essa medida serve como a altura do retângulo e dessa forma pode-se calcular a área através da multiplicação da base pela altura.

Um detalhe importante refere-se ao valor h . Quanto menor for esse valor, mais retângulos serão formados em um mesmo intervalo de tempo o que permite uma melhor aproximação da área da função, que nesse caso corresponde a velocidade do corpo.

Por fim basta somar todas as velocidades obtidas ao longo do intervalo integrado e pronto, é possível obter uma aproximação para a integral da função. Esse mesmo processo pode então ser repetido em função da velocidade para achar a posição. Ao final será possível obter todas as informações necessárias ao simulador para que esse represente o movimento de Objetos.

1.4.6 Aplicação da Matemática – Circunferências e Verificação da Colisão entre Objetos e outros elementos

Já familiarizado com a movimentação de um corpo na simulação, faz-se necessário então refletir no caso de dois corpos se movimentarem e acabarem colidindo entre si. Em caso de nenhum tratamento acontecer para esses casos, os corpos simplesmente continuarão suas trajetórias. Fisicamente isso não está correto e o simulador perderia coesão. Portanto faz-se necessário considerar a colisão e aplicar algum efeito quando detecta-se que essa ocorreu.

Entretanto antes de abordar os possíveis efeitos da colisão descritos posteriormente no próximo tópico, é necessário compreender matematicamente como detectar as colisões

que ocorrem na simulação. Sendo assim, antes de mais nada é necessário definir os tipos de colisões que podem ocorrer durante uma simulação. Logo abaixo é possível conferir as colisões e como essas são consideradas no projeto.

- Colisão entre Objetos: Os objetos referem-se aos corpos da nossa simulação física. São eles que movimentam-se pelo plano através do cálculo apresentado anteriormente. Para facilitar a simulação considera que todos os objetos devem possuir a forma de uma circunferência, o que implica no fato da colisão entre objetos limitar-se a colisão de circunferências;
- Colisão entre Objeto e Linha: As linhas são os elementos que definem a área em que os objetos podem movimentar-se. Um objeto que colide com uma linha não pode a transpassar e para isso é necessário que haja cálculo de colisão entre esses dois elementos. Considerando que um Objeto pode ser visto sob a forma de uma circunferência e uma linha como um segmento de reta, a colisão pode ser vista como algo limitado a reta e circunferência.

Os cálculos relacionados a esses dois tipo de colisão é bem simples e pode ser efetuado através do cálculo de distância entre circunferências e entre uma circunferência e uma reta. Confira logo abaixo o cálculo realizado para descobrir se houve colisão entre duas circunferências e logo em seguida o cálculo para a colisão de uma circunferência com uma linha.

Para a colisão entre círculo considere: $c1$ e $r1$ como sendo respectivamente o centro da circunferência número 1 e o valor de seu raio, já $c2$ e $r2$ como sendo respectivamente o centro circunferência número 2 e o valor de seu raio. O cálculo de colisão entre esses pode ser efetuado através da análise da distância entre seus centros. Para calcular a distância basta aplicar $d = \sqrt{(c2.x - c1.x)^2 + (c2.y - c1.y)^2}$.

Tendo calculado a distância, pode-se realizar a seguinte análise:

$$d > r1 + r2 \implies 0$$

$$d \leq r1 + r2 \implies 1$$

Onde têm-se que 1 refere-se a existência de uma colisão e 0 refere-se a inexistência de colisão.

Já no que tange a colisão de uma circunferência com uma linha ao invés de usar o cálculo tradicional da distância do centro dessa circunferência até a reta/linha, pode-se realizar um cálculo simplificado considerando que as colisões com linhas estão presente em um contexto limitado. Para compreender essa abordagem considere: t sendo a linha que delimita a área máxima superior que objeto pode movimentar-se, b sendo a linha que

delimita a área máxima inferior, l sendo a linha que delimita a área máxima à esquerda e r sendo a linha que delimita a área máxima à direita que objeto pode movimentar-se. Agora considere C sendo o centro da circunferência e r como raio dessa circunferência.

A título de especificação quando aparecer *elemento.x* ou *elemento.y* entenda que busca-se referir as coordenadas naquele eixo daquele elemento matemático. Dessa forma, $C.x$ refere-se a coordenada x do centro da circunferência declarada anteriormente.

Diz-se que ocorre colisão quando um dos seguintes casos acontece:

$$C.y + r \geq t.y$$

$$C.y - r \leq b.y$$

$$C.x - r \leq l.x$$

$$C.x + r \geq r.x$$

Onde têm-se respectivamente a colisão da circunferência com as linhas: superior, inferior, à esquerda e à direita da área de movimento dos Objetos. Observe que aparentemente usou-se as coordenadas de uma reta para informar se houve ou não colisão. O fato é que essas retas são paralelas a algum eixo e então seus pontos terão alguma coordenada fixa. Essa coordenada fixa é que está sendo utilizada para a comparação das equações anteriores.

1.4.7 Aplicação da Física – Efeitos da Colisão entre Objetos, Colisões Elásticas, Conservação de Quantidade de Movimento e Reflexão Especular

Uma vez sendo capaz de calcular se houve ou não colisão na simulação, é necessário definir algum efeito que decorre desse acontecimento. Para o simulador em questão foi definido considerar as colisões como sendo elásticas. Nesse tipo de colisão os objetos não são deformados com o choque e existe a conservação da quantidade de movimento, uma grandeza vetorial física que relaciona a massa e a velocidade de um corpo e extremamente importante para definir o estado dinâmico de um sistema físico. Na 8 apresentada abaixo é possível conferir uma representação do que ocorre em colisões consideradas elásticas.

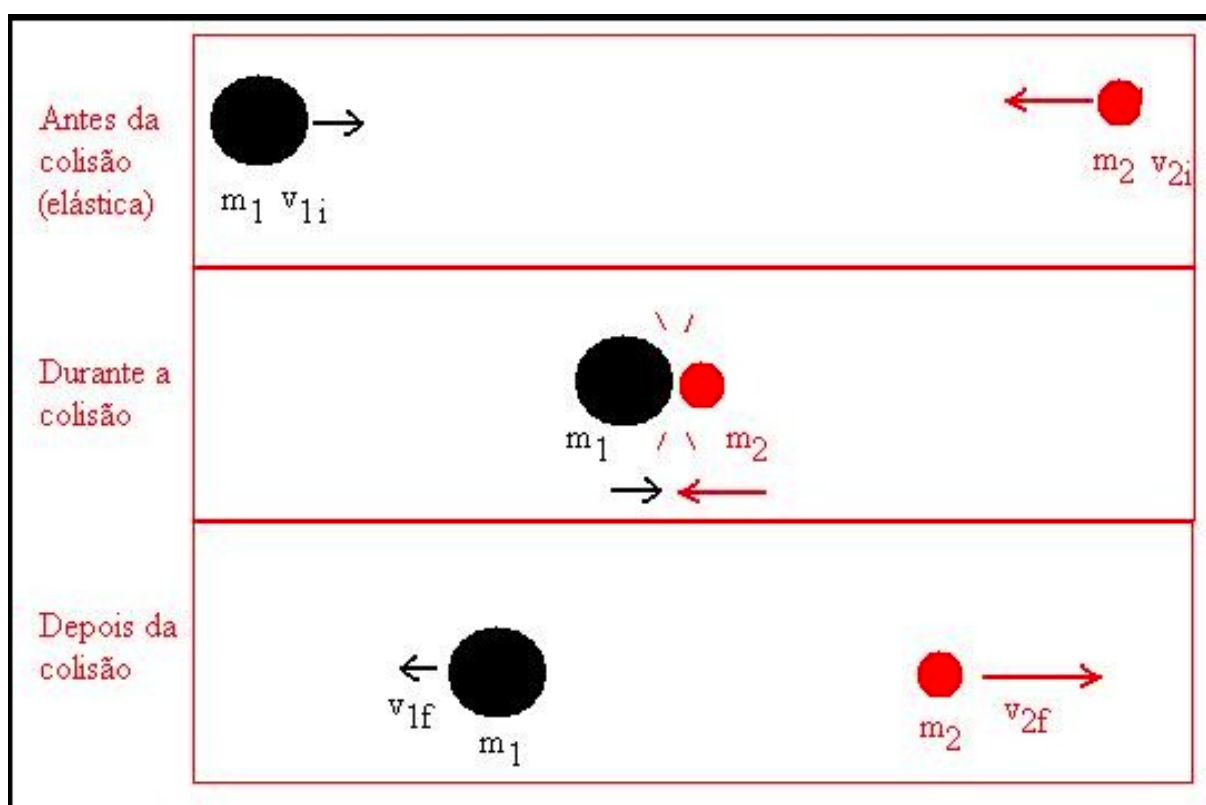


Figura 8 – Colisão elástica em uma direção de movimento

Fonte: <https://en.wikipedia.org/wiki/Elastic_collision>

A título de especificação logo em sequência são apresentada as fórmulas e os passos que permitem a simulação de um sistema de colisão elástica. Essas fórmulas podem ser exploradas e obtidas através da premissa de conservação de energia cinética em sistemas elásticos, e o cálculo das quantidades de movimento para cada corpo presente na colisão. Esses cálculos não serão aprofundados devido aspectos de tempo, mas fica aqui destacada a importância de conhecê-los.

$$\mathbf{v}'_1 = \mathbf{v}_1 - \frac{2m_2}{m_1 + m_2} \frac{\langle \mathbf{v}_1 - \mathbf{v}_2, \mathbf{x}_1 - \mathbf{x}_2 \rangle}{\|\mathbf{x}_1 - \mathbf{x}_2\|^2} (\mathbf{x}_1 - \mathbf{x}_2),$$

$$\mathbf{v}'_2 = \mathbf{v}_2 - \frac{2m_1}{m_1 + m_2} \frac{\langle \mathbf{v}_2 - \mathbf{v}_1, \mathbf{x}_2 - \mathbf{x}_1 \rangle}{\|\mathbf{x}_2 - \mathbf{x}_1\|^2} (\mathbf{x}_2 - \mathbf{x}_1)$$

Figura 9 – Fórmula da velocidade dos corpos após uma Colisão elástica para simulações bidimensionais

Fonte: <https://en.wikipedia.org/wiki/Elastic_collision>

Na imagem acima é possível observar os seguintes elementos: V'_1 e V'_2 sendo a nova velocidade após a colisão. V_1 e V_2 sendo as velocidades pré-colisão dos objetos levados em consideração. X_1 e X_2 sendo o vetor posição de cada objeto, e por fim m_1 e m_2 sendo as respectivas massas desses objetos.

Entretanto deve-se esclarecer que esse tipo de colisão deve ser aplicada quando ocorre o choque entre dois objetos do sistema. Dessa forma para múltiplas colisões entre objetos deve-se estruturar matematicamente um novo sistema, algo que não é abordado no projeto e por consequência não é descrito no relatório. Além disso deve-se observar que essa colisão não é ideal para descrever o encontro entre um objeto e uma linha, visto que não deseja-se modificar a velocidade de um corpo após a colisão com uma linha, pois essa é um elemento estático e que não representa quantidade de movimento.

Nesse caso é necessário aplicar outro conhecimento físico. Um efeito que torna a simulação mais real e coerente fisicamente e replicar o efeito de reflexão sofrida pela luz conforme apresentado no esquema abaixo, onde P refere-se a trajetória da luz e Q refere-se a sua trajetória após a reflexão especular causada na colisão com o espelho.

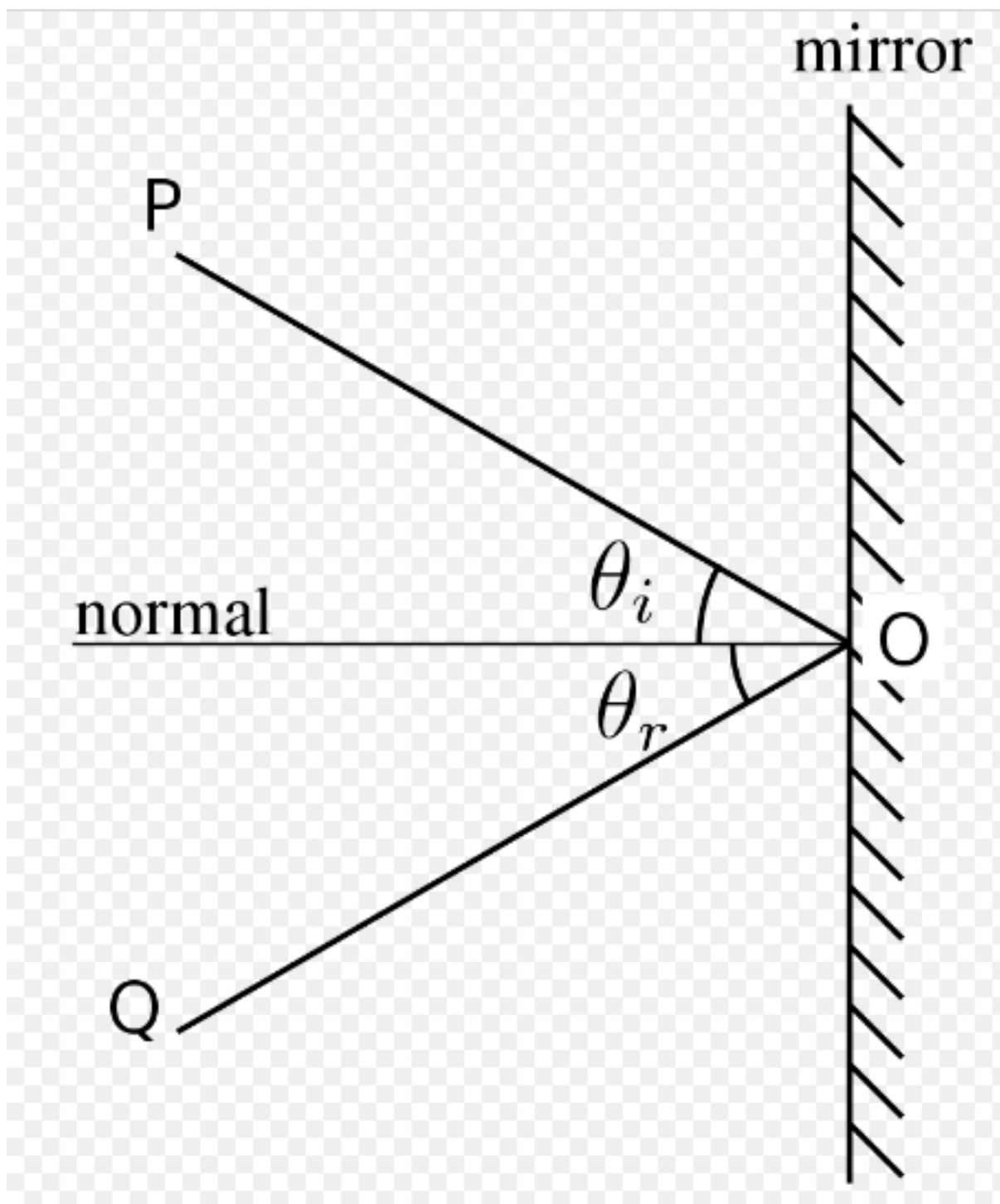


Figura 10 – Apresentação do efeito de reflexão especular

Fonte: <https://en.wikipedia.org/wiki/Specular_reflection>

Utilizando-se desse recurso é possível aplicar um efeito aos corpos que colidem com as linhas de forma a não afetar a energia cinética do sistema, mas apenas alterar o vetor velocidade no que diz respeito a sua direção e sentido para simular um efeito de colisão. Para calcular essa colisão pode-se utilizar a fórmula apresentada abaixo.

$$\vec{v}' = \vec{v} - 2(\vec{v} * \vec{n})\vec{n}$$

Onde têm-se que: \vec{v}' refere-se a nova velocidade do objeto que colidiu com uma linha l . \vec{v} refere-se a velocidade do objeto pré-colisão e \vec{n} refere-se a um vetor normal a linha l .

1.4.8 Aplicação Matemática – Reposicionamento do Objeto após colisão com uma Linha

Diferentemente do que ocorre na vida real, na programação do simulador um Objeto pode acabar transpassando em certo nível uma Linha e detectar a colisão quando essa linha já está “dentro” do objeto. Isso é um problema, pois caso sejam aplicados os cálculos já apresentados nessa situação, existe a grande possibilidade do resultado final não representar a realidade, algo que não deve acontecer no simulador.

Para evitar esse tipo de problema foi necessário o desenvolvimento de um cálculo para reposicionar o Objeto na posição exata onde aconteceria o primeiro contato com a Linha para que todo o processo de reflexão especular fosse aplicado. Esse cálculo é bem simples e sua formulação pode ser observado logo abaixo, bem como um exemplo desenvolvido no software Geogebra que demonstra o resultado obtido a partir desse cálculo.

Para realizar o reposicionamento do Objeto Obj após a colisão com a linha L deve-se considerar alguns aspectos sendo eles:

- \vec{v} : vetor velocidade de Obj ;
- \vec{n} : vetor normal a L ;
- α : ângulo entre \vec{v} e \vec{n} ;
- d : Distância entre o ponto mais externo da Circunferência de Obj em relação a linha L ;
- t : $\tan \alpha$;
- \vec{c} : Vetor para cálculo do Reposicionamento, onde deve-se considerar:
 - Caso a Circunferência transpassar a linha horizontal superior $\vec{c} = -(d * t, d)$
 - Caso a Circunferência transpassar a linha horizontal inferior $\vec{c} = (d * t, d)$

- Caso a Circunferência transpassar a linha vertical esquerda $\vec{c} = (d, d * t)$
- Caso a Circunferência transpassar a linha vertical direita, $\vec{c} = -(d, d * t)$

Uma vez considerado esses aspectos, têm-se que o reposicionamento do Objeto para a posição exatamente em que a circunferência de *Obj* toca *L* sendo:

$$\vec{r}' = \vec{r} - \vec{c}$$

Onde \vec{r}' é a nova posição de *Obj* e \vec{r} é a sua posição no momento em que o cálculo de reposicionamento ocorre.

Para finalizar esse tópico, na figura 11 abaixo é possível conferir um exemplo da aplicação desse cálculo para reposicionar uma circunferência representando um objeto que transpassou a linha limite à direita. Inclusive esse foi o exemplo que permitiu originalmente a formulação desse cálculo.

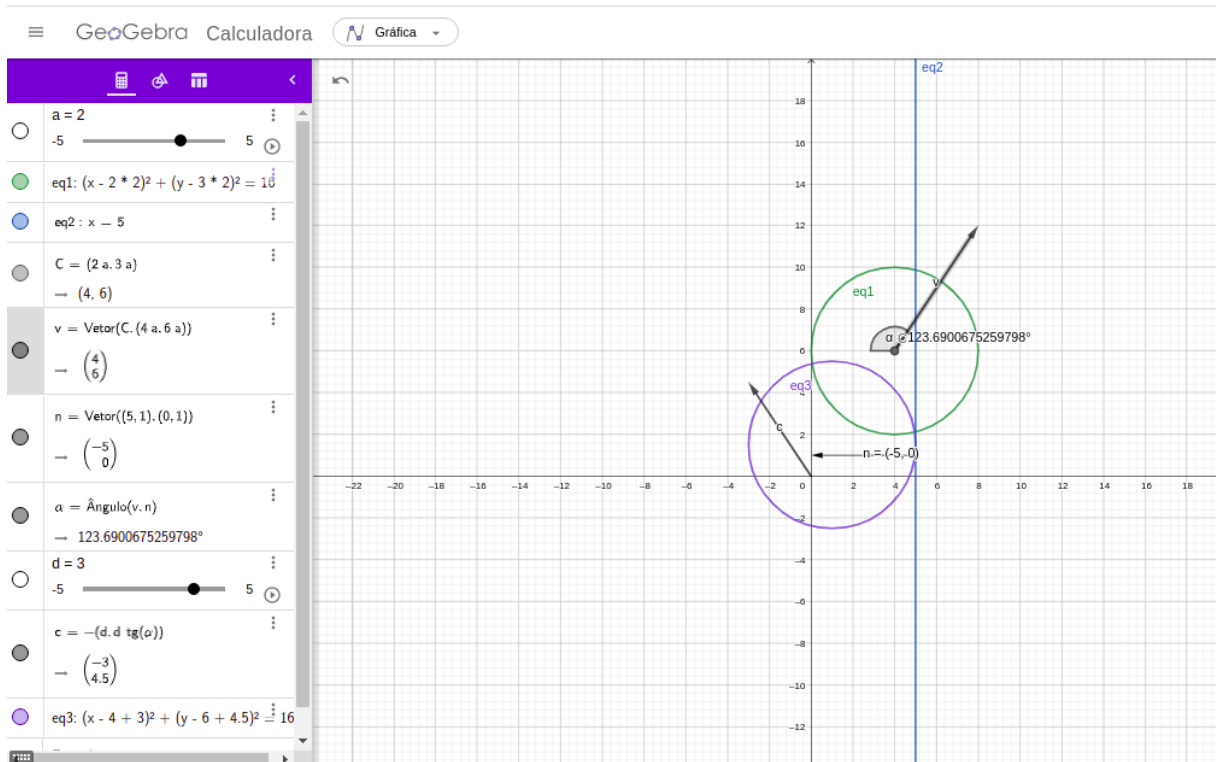


Figura 11 – Exemplo da aplicação do equacionamento de reposicionamento da circunferência verde resultando na circunferência azul usando-se o software Geogebra

Fonte: Produção própria

1.4.9 Aplicação Matemática – Matrizes e Transformações Geométricas

As matrizes são uma estrutura matemática similares a tabelas, uma vez que possuem linhas e colunas. O seu uso é muito abrangente, todavia para o trabalho utilizou-se matrizes para realizar transformações geométricas para o processo de renderização. Imagine que deseja-se renderizar diversas figuras geométricas, todavia deve-se considerar que as coordenadas de seus vértices estão sendo calculadas a partir do centro da tela, e não do canto superior direito como geralmente acontece na computação gráfica. Para resolver esse problema seria possível construir uma Matriz que representa os vértices da figura e multiplicar essa por uma matriz de transformação de translação que reposicionaria os vértices da figura para a posição desejada permitindo a renderização conforme especificado.

Para compreender como é possível realizar essas transformações deve-se analisar alguns aspectos sendo eles:

- A Matriz de Transformação é uma matriz 3x3 onde a mudança dos seus valores é capaz de alterar uma outra matriz no processo de multiplicação. Essa alteração pode ser compreendida e interpretada graficamente;
- A matriz que será multiplicada pela matriz de transformação contém os pontos que formam a figura geométrica. Deve ser uma matriz com 3 linhas para que a multiplicação seja possível, e cada coluna deve representar um ponto da figura geométrica. A representação de um ponto em forma de matriz é muito simples e pode ser dada na forma apresentada abaixo.

Considere o ponto P com as coordenadas x e y . Sua representação na forma de uma matriz pode-ser dada como:

$$M = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Para representar figuras geométricas ou outros elementos como vetores na forma de matriz basta acrescentar os pontos que os formam nessa matriz de modo a formar uma matriz $3 \times N$ onde N é a quantidade de pontos desse elemento.

Compreendido esses detalhes básicos, existem certas matrizes de transformação já conhecidas que podem ser aplicadas para modificar a representação de figuras geométricas, algo útil ao processo de renderização de figuras. Logo em seguida na FIGURAN é possível observar uma imagem que apresenta algumas matrizes de transformação, bem como o resultado obtido quando essas multiplicam uma matriz contendo os vértices de um quadrado.

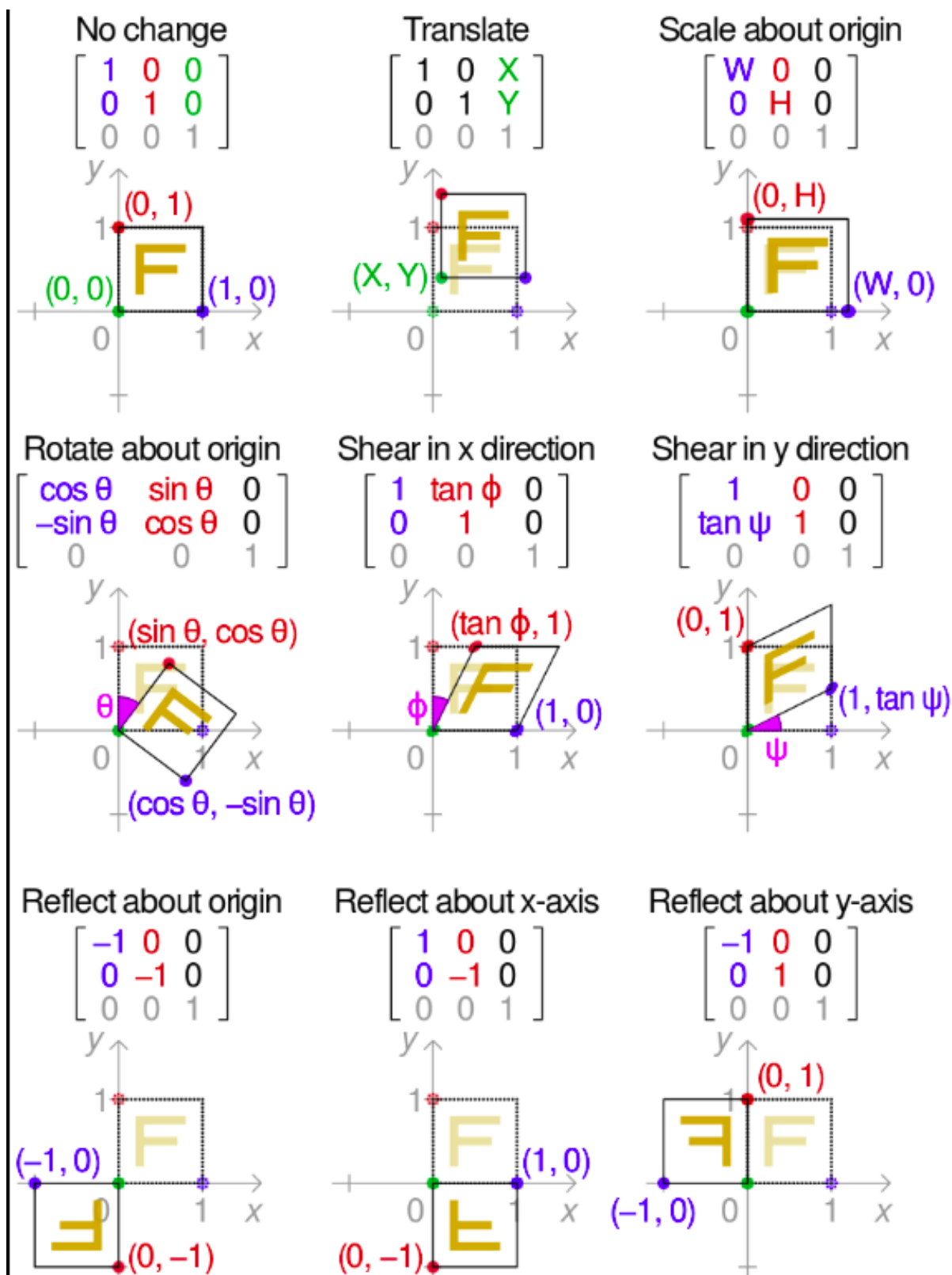


Figura 12 – Apresentação de algumas matrizes de transformação, bem como o resultado causado na multiplicação por uma matriz representando os vértices de um quadrado

1.5 Desenvolvimento do Pacote Base

Essa etapa do projeto pode ser considerada a mais importante para a concretização do simulador físico, uma vez que é nela onde foram realizadas as implementações da maior parte dos tópicos discutidos na seção anterior sobre aplicações matemáticas e físicas presentes no trabalho. O intuito do desenvolvimento desse pacote era gerar um conjunto de classes capazes de representar a base estrutural do simulador físico de modo que não fosse necessária a implementação de uma interface para poder montar uma simulação – sendo essa totalmente numérica e sem representação gráfica.

Abordando o aspecto organizacional dessa etapa de desenvolvimento, pode-se apresentar que ela ocorreu inicialmente isolada de outros processos de implementação de código, mas com o avançar do tempo e consequentemente o avançar da complexidade dos componentes implementados, essa etapa passou a ocorrer de maneira simultânea ao desenvolvimento do pacote Interface.

Uma vez compreendido esse aspecto introdutório dessa etapa de desenvolvimento, é possível apresentar alguns detalhes acerca desse conteúdo. Como ponto inicial, faz-se necessário e interessante comentar algumas das principais necessidades envolvidas no desenvolvimento desse pacote, já que são esses elementos que configuraram o andamento dessa etapa. Logo abaixo é possível observar a listagem das principais funcionalidades necessárias desse pacote.

- Representar um Objeto da simulação: um Objeto da simulação nada mais é do que o coração dela. São os elementos que estão tendo seus comportamentos simulados. São eles que irão possuir uma massa, uma aceleração, uma velocidade e uma posição. São eles que irão sofrer colisão. São eles que serão representados graficamente como corpos físicos da simulação;
- Detectar, calcular e aplicar a colisão: Um simulador físico interessante é aquele que é capaz de detectar e aplicar colisão (em certo nível) aos Objetos que movimentam-se pelo plano. Essa necessidade está relacionada com o desejo de criar simulações com um critério físico mais rigoroso. Se os objetos simplesmente se deslocassem pelo plano atravessando uns aos outros, a simulação não pareceria nada concreta e fisicamente incoerente;
- Representar elementos matemáticos usadas durante a simulação: Durante o desenvolvimento do projeto diversos aspectos matemáticos são necessários para construir a base da simulação, seja essa referente a aspectos básicos ou até mesmo referente a elementos de interface. Dessa forma, como foi possível observar na seção anterior, elementos como pontos, vetores e matrizes são essências para o projeto.

Tendo sido compreendido a estruturação básica do pacote Base é possível dar sequência ao relato do trabalho aplicado nessa etapa através da apresentação das classes finais criadas. Na tabela abaixo é possível observar o nome da classe, suas funcionalidades e algumas observações interessantes relacionadas a ela.

Tabela 3 – Componentes do Pacote Base - Parte 1

Nome	Funcionalidades	Observações
Ponto	Representar um ponto no plano cartesiano através de Coordenadas x e y.	Essa classe é uma das mais simples de todas, sendo comumente utilizada para compor outras classes como as que representam figuras geométricas.
Vetor	Representar um vetor no plano cartesiano e permitir operações comuns aos vetores.	Essa classe é importante pois permite que certas propriedades físicas sejam atribuídas a um objeto, como é caso da velocidade.
Matriz	Representar uma matriz NxN e permitir operações comuns às matrizes.	Essa classe é muito utilizada para realizar transformações geométricas, uma vez que conforme fora visto pode-se utilizar de uma matriz de transformação para alterar figuras geométricas. A aplicação dessa classe pode ser observada na classe Renderizador do pacote Interface, apresentado em seções posteriores do trabalho.
Retangulo	Representar a figura geométrica de um retângulo no plano.	Essa classe foi criada com o intuito de possibilitar que objetos da simulação possuíssem o formato retangular.
Triangulo	Representar a figura geométrica de um triângulo no plano.	Essa classe foi criada com o intuito de possibilitar que objetos da simulação possuíssem o formato triangular.
Circunferencia	Representar a figura geométrica de uma circunferência no plano.	Essa classe foi criada com o intuito de possibilitar que objetos da simulação possuíssem o formato circular. Ao fim, foi definida como o formato padrão para a realização das simulações.
Linha	Representar um segmento de reta no plano.	Essa classe foi criada principalmente para delimitar a área em que os objetos da simulação podem ocupar, uma vez que ao tocar uma linha o objeto sofre o efeito de reflexão especular

Fonte: Produção própria

Tabela 4 – Componentes do Pacote Base - Parte 2

Nome	Funcionalidades	Observações
Aceleracao	Representar funções de aceleração que definem o movimento uniformemente variado e permitir que a aceleração seja calculada em qualquer instante de tempo para qualquer um dos eixos de direção.	Essa classe armazena em sua estrutura interna ponteiros para funções C++. Dessa forma é possível criar uma estrutura capaz de calcular a aceleração em um determinado tempo abstraindo os detalhes e o comportamento da função programada. Diz-se que essa classe permite o cálculo de aceleração em qualquer um dos eixos devido ao fato de que em sua estrutura está prevista a definição de duas funções de aceleração, uma para cada eixo. Dessa forma um objeto não necessariamente precisa possuir acelerações iguais em ambos eixos de movimento
Objeto	Representar o elemento fundamental da simulação. Além de definir todas as propriedades físicas de um objeto para a simulação, também é responsável por realizar os cálculos de movimentação dos corpos na simulação.	Essa classe realiza os cálculos de movimentação através do processo de integração da aceleração, onde é possível descobrir a velocidade do corpo em um determinado período, bem como a sua posição pelo mesmo processo levando-se em conta a velocidade descoberta. Conforme fora citado, seu formato padrão é representado por uma circunferência.
DetectorColisao	Detectar a colisão entre objetos da simulação; Detectar colisão entre objetos e linhas definidas no sistema; Aplicar cálculos como conservação de movimento e reflexão especular a um objeto após a sua colisão.	Essa classe realiza todos os cálculos envolvendo conservação de movimento em um sistema elástico. Também é ela que calcula a reflexão especular quando um Objeto colide com uma Linha. Para realizar os cálculos, usa-se constante equações vetoriais.

Fonte: Produção própria

Visando dar sequência ao conteúdo dessa seção, logo em seguida é possível observar algumas imagens da implementação das classes focando em alguns elementos para demonstrar o trabalho aplicado nesse processo. Devido ao fato desse pacote ser consideravelmente extenso, fica difícil de apresentar a implementação completa de todas as classes. Caso seja desejado tal verificação, é possível acessar o repositório do projeto e conferir a documentação do pacote ou visualizar diretamente o código comentado para mais detalhes.

Entretanto, visando uma abordagem mais profunda do pacote, algumas das classes serão abordadas em maiores detalhes em seguida. As classes escolhida para serem detalhadas são: Objeto e DetectorColisao, uma vez que são as elas que fundamentam todo o aspecto físico da simulação. Sem essas classes não seria possível realizar se quer algum tipo de simulação.

1.5.1 Discussão sobre a classe Objeto

Conforme explicitado anteriormente, a classe Objeto é fundamental para a estruturação do trabalho, uma vez que refere-se a componente que representa o corpo físico que faz parte da simulação. Para compreender melhor a importância e o funcionamento básico dessa classe, sua estrutura pode ser observada na imagem do diagrama de classes logo abaixo e em seguida é possível visualizar uma breve listagem dos pontos mais relevantes.

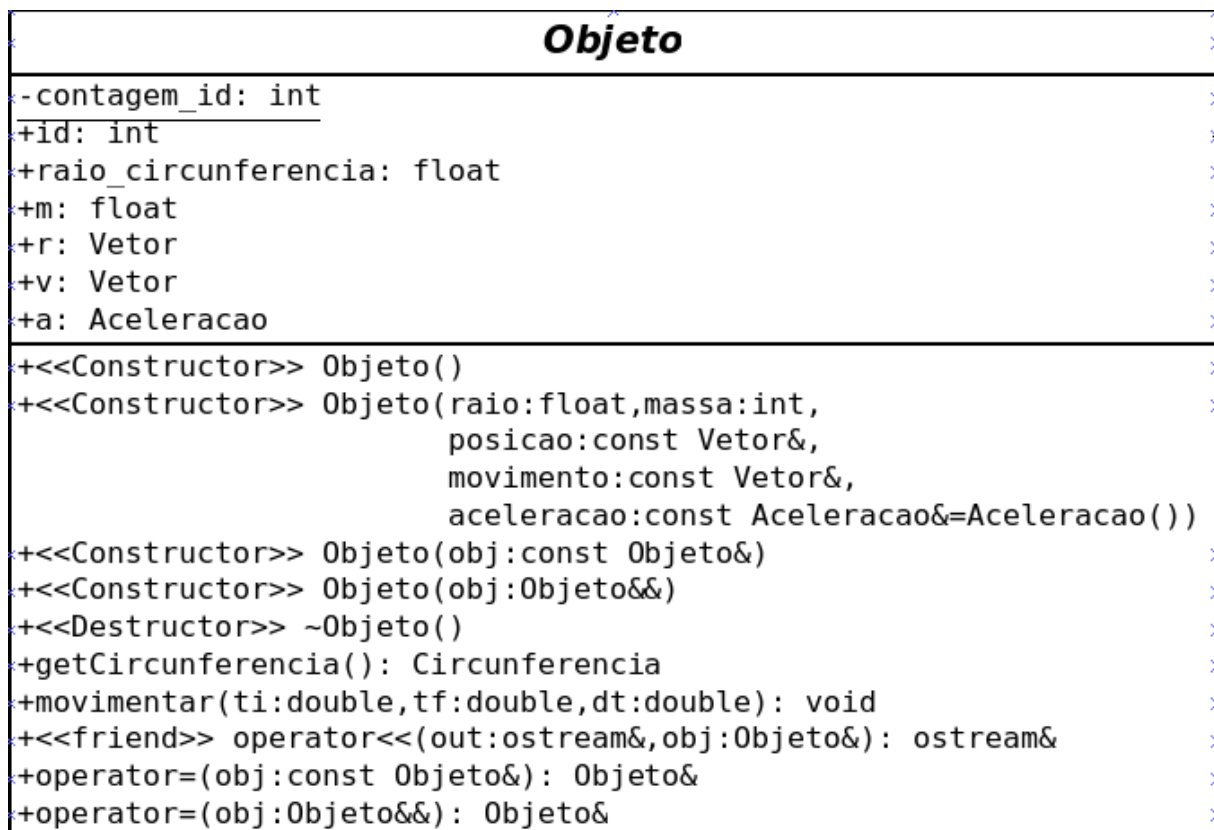


Figura 13 – Diagrama de classes da componente Objeto do Pacote Base

Fonte: Produção Própria

Uma vez observado o recorte do diagrama de classes da classe Objeto, faz-se interessante discutir alguns de seus atributos e métodos. Confira abaixo um detalhamento sobre alguns desses aspectos.

- Atributo `raio_circunferencia`: Indica o raio da circunferência que conforme já fora abordado, é a forma geométrica adotada para representar os Objetos da simulação;
- Atributo `m`: Refere-se a massa do Objeto. Assim como já foi tratado na seção anterior que discute os princípios físicos presentes no trabalho, a massa é essencial para o cálculo da conservação de movimento quando ocorre uma colisão entre Objetos;
- Atributos `r`, `v`, `a`: Referem-se respectivamente a posição do Objeto (representada vetorialmente), a velocidade desse objeto (também representada vetorialmente) e a sua aceleração (representada por um conjunto de funções definidas em `Aceleracao`). Esses atributos são a base fundamental do Objeto, uma vez que são eles que definem como esse desloca-se pelo plano. É com esses atributos que é possível trabalhar a questão já discutida da integração numérica para descobrir informações como velocidade e posição a partir da aceleração.
- Método `movimentar`: Esse é o método responsável pela implementação do integrador numérico e por consequência de toda a movimentação e física intimamente relacionada ao Objeto. Abaixo tem-se a apresentação de um trecho do código da classe Objeto correspondente a implementação desse código para que essas questões possam ser discutidas.

```
152 void movimentar(double ti, double tf, double dt){  
153     // Uso do Integrador de Euler  
154  
155     // Definição da quantidade de passos, bem como o incremento em cada passo  
156     int n_passos = 10000;  
157     double h = (1.f/n_passos) / (1.f/dt);  
158  
159     // Armazenamento dos dados da posição e velocidade iniciais em cada eixo  
160     double r0_x = r.x;  
161     double r0_y = r.y;  
162     double v0_x = v.x;  
163     double v0_y = v.y;  
164  
165     // Integração da Velocidade e da Posição  
166     for (ti; ti<tf; ti+=h){  
167         v0_x = v0_x + h*(a.func_x(ti));  
168         v0_y = v0_y + h*(a.func_y(ti));  
169  
170         r0_x = r0_x + h*(v0_x);  
171         r0_y = r0_y + h*(v0_y);  
172     }  
173  
174     // Atualização da Velocidade com o valor calculado por Integração  
175     v.x = v0_x;  
176     v.y = v0_y;  
177  
178     // Atualização da Posição com o valor calculado por Integração  
179     r.x = r0_x;  
180     r.y = r0_y;  
181 }
```

Figura 14 – Código do método movimentar presente no Objeto que realiza os processos de Integração numérica

Fonte: Produção Própria

Observe a partir dos comentários efetuados na classe, os elementos discutidos anteriormente quando foi tratado sobre o integrador de Euler. Nessa implementação é possível observar a definição da quantidade de passos em uma integração. Além disso é possível visualizar na linha 166, no laço de repetição o processo de integração, onde inicialmente integra-se a velocidade e logo em seguida integra-se a posição – isso considerando a integração individual em cada eixo do plano.

Um dos pontos mais importantes de se notar, é a chamada da função aceleração contida no objeto referenciado pelo atributo `a`. No código pode-se observar esse aspecto nas linhas 167 e 168. Essas duas linhas nada mais são que a integração da aceleração para descoberta de velocidade pelo método de integração de Euler. Além disso, nas próximas duas linhas de códigos é possível observar o uso dessa mesma velocidade recém calculada para achar pelo mesmo procedimento a posição do objeto naquele instante de tempo.

É dessa forma que é possível a partir de uma informação como aceleração, calcular a velocidade e posição do objeto e atualizar essas propriedades com o passar do tempo o que gera a movimentação dos Objetos na simulação.

1.5.2 Discussão sobre a classe `DetectorColisao`

O `DetectorColisao` é a classe responsável por toda a detecção e cálculos relacionados às colisões que ocorrem no simulador. Os principais cálculos que compõem essa classe e seus métodos já foram detalhados na seção que aborda a teoria matemática e física, sendo assim logo abaixo é apresentado o diagrama de classes, bem como alguns breves detalhamentos sobre o que refere-se os elementos mais interessantes dessa classe.

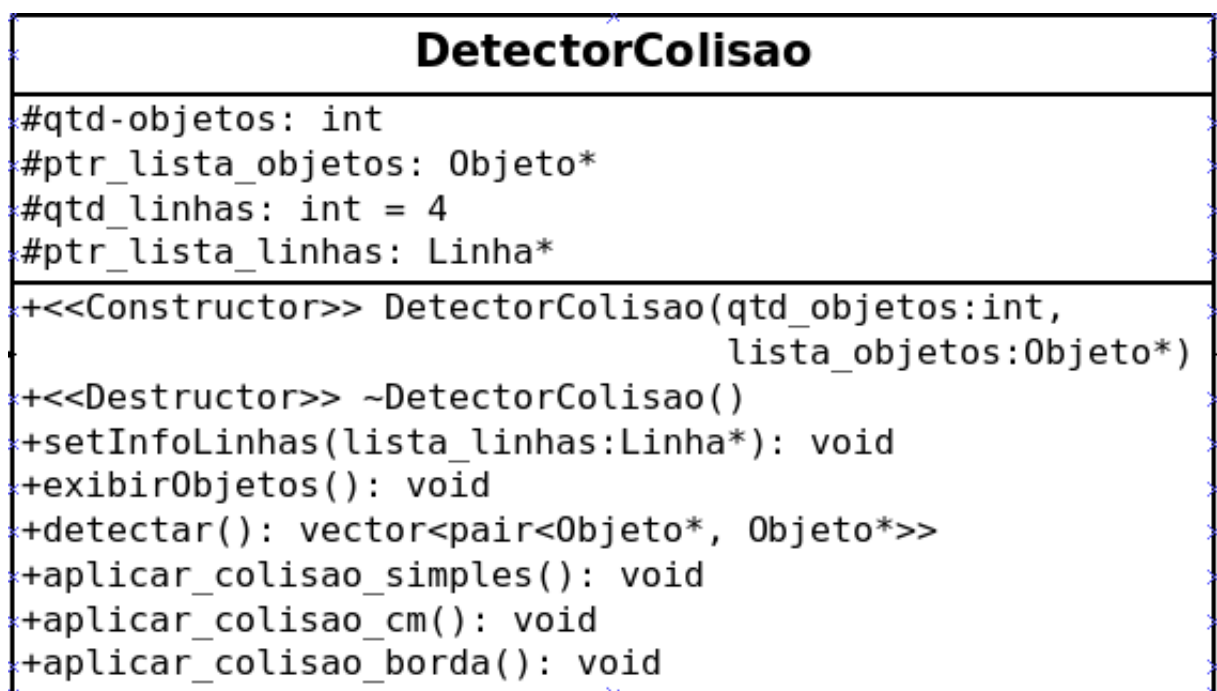


Figura 15 – Diagrama de classes da componente DetectorColisao do Pacote Base

Fonte: Produção Própria

Observado o seu diagrama, abaixo é possível conferir uma breve listagem dos aspectos essenciais dessa classe.

- Atributo `ptr_lista_objetos`: Atributo que refere-se a um array dos Objetos que serão levados em consideração para o cálculo da colisão, e aplicação dos efeitos após as colisões serem detectadas;
- Atributo `ptr_lista_linhas`: Atributo que refere-se a um array de Linhas. Essas linhas são usadas para delimitar a área em o que os Objetos podem locomover-se. Ao colidir com uma dessas linhas o objeto sofre o efeito de reflexão especular já detalhado anteriormente.
- Método `detectar`: Esse é o método responsável por detectar as colisões e gerar um vetor C++ que possui a relação de colisão entre Objetos para que esses possam sofrer os efeitos definidos pelo choque entre Objetos. O cálculo realizado nesse método segue o esquema já apresentado anteriormente;
- Método `aplicar_colisao_simples`: Define um efeito para a colisão entre Objetos. Esse método é mais usado para testes e não representa uma situação extremamente realística, uma vez que os Objetos que colidiram apenas possuem a direção de suas velocidades invertidas;
- Método `aplicar_colisao_cm`: Esse é o método responsável por aplicar o cálculo de conservação de movimento após uma colisão para simular um sistema inelástico conforme discutido em seções anteriores;
- Método `aplicar_colisao_borda`: Esse método aplica o efeito de reflexão especular aos Objetos após eles colidirem com uma Linha delimitadora. A operação desse método funciona exatamente como já fora descrito anteriormente, todavia existe uma etapa adicional executada nesse método que não havia sido mencionada anteriormente. Trata-se do reposicionamento do objeto, aspecto que já foi abordado em tópico próprio na seção de teoria anteriormente.

Antes de encerrar a discussão a cerca da classe `DetectorColisão`, faz-se interessante apresentar alguns trechos do código principalmente relacionados às funcionalidades discutidas anteriormente, uma vez que esses códigos permitem um melhor entendimento de como alguns processos mais importantes da simulação física ocorre no projeto, bem como serve o propósito de apresentação de resultados.

```

85     vector<pair<Objeto*,Objeto*>> detectar()
86     // Criação do vetor que será retornado contendo os Objetos que colidiram
87     vector<pair<Objeto*,Objeto*>> v;
88     //Variáveis auxiliares
89     Objeto ob1, ob2;
90
91     // Percorrer o array de Objeto pegando um a um para analisar com os demais Objeto
92     // a fim de verificar colisão
93     for (int i=0; i<(qtd_objetos-1); i++){
94         for (int j=i+1; j<qtd_objetos; j++){
95             ob1 = *(ptr_lista_objetos+i);
96             ob2 = *(ptr_lista_objetos+j);
97
98             // Cálculo da distância
99             float distancia = pow((pow((ob1.r.x - ob2.r.x), 2) + pow((ob1.r.y - ob2.r.y), 2)),(1/2.f));
100
101             // Caso houve colisão, adicionar referências para os Objeto no vetor de retorno
102             if (distancia <= (ob1.raio_circunferencia + ob2.raio_circunferencia)){
103                 v.push_back(pair<Objeto*,Objeto*>(ptr_lista_objetos+i, ptr_lista_objetos+j));
104             }
105         }
106     }
107     return v;
108 }

```

Figura 16 – Código para a detecção de colisão entre circunferências

Fonte: Produção Própria

```

141 void aplicar_colisao_cm(){
142     // Detecção das Colisões
143     vector<pair<Objeto*,Objeto*>> colisoes = detectar();
144
145     // Declaração de Variáveis Auxiliares
146     Vetor v1, v2, r1, r2;
147     float m1, m2;
148
149     // Iteração sobre o vetor para acessar as duplas de Objetos que colidiram entre si
150     for(int i=0; i<colisoes.size(); i++){
151         // Dados do primeiro Objeto da dupla de Colisão
152         v1 = move(colisoes[i].first->v);
153         r1 = move(colisoes[i].first->r);
154         m1 = colisoes[i].first->m;
155
156         // Dados do segundo Objeto da dupla de Colisão
157         v2 = move(colisoes[i].second->v);
158         r2 = move(colisoes[i].second->r);
159         m2 = colisoes[i].second->m;
160
161         // Cálculo do novo vetor Velocidade através do cálculo vetorial de colisão bidimensional
162         // entre dois Objetos
163         Vetor v1_ = v1 - ((2*m2)/(m1+m2)) * (((v1-v2)*(r1-r2)) * (1/pow((r1-r2).getModulo(),2))) * (r1-r2);
164         Vetor v2_ = v2 - ((2*m1)/(m1+m2)) * (((v2-v1)*(r2-r1)) * (1/pow((r2-r1).getModulo(),2))) * (r2-r1);
165
166         // Alteração das velocidades dos Objeto pelo novo valor calculador
167         colisoes[i].first->v = move(v1_);
168         colisoes[i].second->v = move(v2_);
169     }

```

Figura 17 – Código para a aplicação da conservação da quantidade de movimento

Fonte: Produção Própria

```
264         if (colidiu){
265             // Cálculo do ângulo formado entre o Vetor normal da Linha e o Vetor Velocidade do Objeto
266             float angulo = acos(1.f * (v * normal) / (v.getModulo() * normal.getModulo()));
267
268             // Cálculo do Vetor Reposicionamento, conforme descrito anteriormente
269             Vetor reposicionamento;
270             if (j == 0 || j == 1)
271                 reposicionamento = Vetor(d*tan(angulo), d);
272             else
273                 reposicionamento = Vetor(d, d*tan(angulo));
274
275             // Soma/Subtração do Vetor reposicionamento ao Vetor posição do Objeto
276             if (j == 0 || j == 3)
277                 (ptr_lista_objetos+i)->r = (ptr_lista_objetos+i)->r - reposicionamento;
278             else
279                 (ptr_lista_objetos+i)->r = (ptr_lista_objetos+i)->r + reposicionamento;
280
281             // Aplicação da reflexão especular através de cálculo vetorial
282             (ptr_lista_objetos+i)->v = v - 2*(v*l.normal)*l.normal;
283         }
284     }
285 }
286 }
```

Figura 18 – Código para o cálculo de reposicionamento do Objeto na colisão com uma Linha

Fonte: Produção Própria

Para encaminhar essa seção para a sua finalização, pretende-se abordar o fato do pacote Base implementar uma serie de testes para conferir se todas as funcionalidades das classes desenvolvidas estão operando corretamente. Devido ao fato de existirem diversos testes realizados e esses podem ser relativamente extensos, bem como observado o fato desses elementos fazerem parte da simulação final, apenas a implementação e resultado referente a classe Objeto são apresentados logo em seguida visando demonstrar o trabalho aplicado e o principal resultado obtidos nessa etapa do desenvolvimento do trabalho.

```
10  int main(){
11      // Criação dos elementos que compõem um Objeto
12      Vetor posicao = Vetor(0,0);
13      Vetor movimento = Vetor(10,10);
14      Aceleracao aceleracao = Aceleracao(constante_0, constante_0);
15
16      // Criação de Objetos
17      Objeto o1 = Objeto(20, 10, posicao, movimento, aceleracao);
18      cout << o1 << endl;
19
20      Objeto o2 = move(o1);
21
22      // Teste a chamada do método de movimentação do Objeto
23      int cont = 0;
24      int max = 20;
25      double ti = 0;
26      double escala = 1;
27
28      for (cont; cont<=max; cont++){
29          cout << o2 << endl;
30
31          o2.movimentar(ti, ti+escala, escala);
32      }
33
34      return 0;
35 }
```

Figura 19 – Teste da classe Objeto com foco para a movimentação

Fonte: Produção Própria

```
Lima001@LimaNotebook:~/Area de Trabalho/bcc-projeto-poo/src/Base$ ./teste objeto.out
Objeto(20;10;Vetor(0,0);Vetor(10,10);Aceleracao(0x7ffc3f1f09b0,0x7ffc3f1f09b8))
Objeto(20;10;Vetor(0,0);Vetor(10,10);Aceleracao(0x7ffc3f1f09e0,0x7ffc3f1f09e8))
Objeto(20;10;Vetor(11.0013,11.0013);Vetor(12.0002,12.0002);Aceleracao(0x7ffc3f1f09e0,0x7ffc3f1f09e8))
Objeto(20;10;Vetor(24.003,24.003);Vetor(14.0004,14.0004);Aceleracao(0x7ffc3f1f09e0,0x7ffc3f1f09e8))
Objeto(20;10;Vetor(39.0051,39.0051);Vetor(16.0006,16.0006);Aceleracao(0x7ffc3f1f09e0,0x7ffc3f1f09e8))
Objeto(20;10;Vetor(56.0076,56.0076);Vetor(18.0008,18.0008);Aceleracao(0x7ffc3f1f09e0,0x7ffc3f1f09e8))
Objeto(20;10;Vetor(75.0105,75.0105);Vetor(20.001,20.001);Aceleracao(0x7ffc3f1f09e0,0x7ffc3f1f09e8))
Objeto(20;10;Vetor(96.0138,96.0138);Vetor(22.0012,22.0012);Aceleracao(0x7ffc3f1f09e0,0x7ffc3f1f09e8))
Objeto(20;10;Vetor(119.018,119.018);Vetor(24.0014,24.0014);Aceleracao(0x7ffc3f1f09e0,0x7ffc3f1f09e8))
Objeto(20;10;Vetor(144.022,144.022);Vetor(26.0016,26.0016);Aceleracao(0x7ffc3f1f09e0,0x7ffc3f1f09e8))
Objeto(20;10;Vetor(171.026,171.026);Vetor(28.0018,28.0018);Aceleracao(0x7ffc3f1f09e0,0x7ffc3f1f09e8))
Objeto(20;10;Vetor(200.031,200.031);Vetor(30.002,30.002);Aceleracao(0x7ffc3f1f09e0,0x7ffc3f1f09e8))
Objeto(20;10;Vetor(231.036,231.036);Vetor(32.0022,32.0022);Aceleracao(0x7ffc3f1f09e0,0x7ffc3f1f09e8))
Objeto(20;10;Vetor(264.042,264.042);Vetor(34.0024,34.0024);Aceleracao(0x7ffc3f1f09e0,0x7ffc3f1f09e8))
Objeto(20;10;Vetor(299.048,299.048);Vetor(36.0026,36.0026);Aceleracao(0x7ffc3f1f09e0,0x7ffc3f1f09e8))
Objeto(20;10;Vetor(336.055,336.055);Vetor(38.0028,38.0028);Aceleracao(0x7ffc3f1f09e0,0x7ffc3f1f09e8))
Objeto(20;10;Vetor(375.061,375.061);Vetor(40.003,40.003);Aceleracao(0x7ffc3f1f09e0,0x7ffc3f1f09e8))
Objeto(20;10;Vetor(416.069,416.069);Vetor(42.0032,42.0032);Aceleracao(0x7ffc3f1f09e0,0x7ffc3f1f09e8))
Objeto(20;10;Vetor(459.076,459.076);Vetor(44.0034,44.0034);Aceleracao(0x7ffc3f1f09e0,0x7ffc3f1f09e8))
Objeto(20;10;Vetor(504.085,504.085);Vetor(46.0036,46.0036);Aceleracao(0x7ffc3f1f09e0,0x7ffc3f1f09e8))
Objeto(20;10;Vetor(551.093,551.093);Vetor(48.0038,48.0038);Aceleracao(0x7ffc3f1f09e0,0x7ffc3f1f09e8))
Objeto(20;10;Vetor(600.102,600.102);Vetor(50.004,50.004);Aceleracao(0x7ffc3f1f09e0,0x7ffc3f1f09e8))
Lima001@LimaNotebook:~/Area de Trabalho/bcc-projeto-poo/src/Base$
```

Figura 20 – Resultado obtido com a execução do código anterior

Fonte: Produção Própria

1.6 Desenvolvimento do Pacote Interface

Um dos elementos mais fundamentais do trabalho e quesito elencando como requisito é a criação de uma interface gráfica para o projeto de simulações físicas. Conforme é possível observar em seções anteriores, foi estipulado ao trabalho o uso da biblioteca SDL2 para a construção dos elementos de interface de usuário do projeto. Tendo observado que um primeiro contato já havia sido realizado com o SDL conforme especificado na seção Primeiro Contato com o SDL, o processo de desenvolvimento de uma interface constituía na tarefa de melhorar e adicionar mais recursos da biblioteca através da sua abstração pela modelagem de classes.

Sendo assim pode-se sumarizar essa etapa do projeto como o processo de desenvolvimento de uma camada de abstração ao SDL através da separação dos recursos em classes concretas e modulares capazes de interagir entre si para criar uma interface gráfica para o projeto.

No que diz respeito a organização e processo de desenvolvimento desse pacote, pode-se dizer que inicialmente fora dedicado um tempo especial para a sua modelagem e construção, todavia ao passar do tempo alguns dos recursos da interface tornaram-se dependentes de outros elementos do projeto, como é o caso do processo de renderização de Objetos da simulação que só era possível com o desenvolvimento da própria classe Objeto. Sendo assim conforme o avançar dessa etapa o desenvolvimento tornou-se principalmente simultâneo ao desenvolvimento do pacote Base.

Tomado nota desses fatos, é possível detalhar o processo de desenvolvimento das componentes da interface. Para dar início é interessante apontar algumas das principais necessidades envolvidas no desenvolvimento desse pacote, uma vez que são esses aspectos que ditaram o andamento dessa etapa. Logo abaixo é possível observar a listagem das principais funcionalidades necessárias desse pacote.

- Deve ser capaz de importar e permitir o uso das funcionalidades SDL;
- Deve ser capaz de criar e exibir desenhos simples em uma tela;
- Deve ser capaz de lidar com eventos de usuário como o uso do teclado;

Uma vez observado esses aspectos, deu-se início ao processo de desenvolvimento das classes para o pacote. Ressalta-se aqui o aspecto modular do pacote, uma vez que a total usufruição dos recursos só é possível através do uso das diversas componentes criadas. Isso deve-se ao fato de interfaces serem elementos complexos que dependem diversos elementos operando em conjunto para funcionar. Por exemplo, não é possível renderizar uma figura se não existir uma tela em que essa ação possa ser efetuada.

Tendo sido compreendido a estruturação básica do pacote Interface é possível dar sequência ao relato do trabalho aplicado nessa etapa através da apresentação das classes finais criadas em um modelo já visto na seção anterior que trata sobre o pacote Base.

Tabela 5 – Componentes do Pacote Interface - Parte 1

Nome	Funcionalidades	Observações
Controlador_SDL	Controlar a inicialização e finalização do SDL permitindo o correto uso das funcionalidades disponíveis pela biblioteca; Pausar a aplicação SDL	Essa é a classe mais básica do pacote Interface. Sem a instanciação de um objeto desse tipo não é possível utilizar as outras classes do pacote corretamente, pois essas dependem da inicialização do SDL.
Objeto	Representar o elemento fundamental da simulação. Além de definir todas as propriedades físicas de um objeto para a simulação, também é responsável por realizar os cálculos de movimentação dos corpos na simulação.	Essa classe realiza os cálculos de movimentação através do processo de integração da aceleração, onde é possível descobrir a velocidade do corpo em um determinado período, bem como a sua posição pelo mesmo processo levando-se em conta a velocidade descoberta. Conforme fora citado, seu formato padrão é representado por uma circunferência.
Relogio	Controlar a taxa de FPS da simulação.	Essa classe é responsável por controlar e permitir que os resultados gráficos sejam exibidos de maneira julgada harmoniosa. A título de especificação, o controle do FPS ocorre através do atraso da aplicação em casos do FPS atual ser superior ao desejado.
CorRGBA	Representar uma cor e permitir que essa seja usada por outros elementos do pacote em um formato específico	As cores são basicamente criadas a a partir da representação rgba comum, todavia é um dos métodos declarados na classe que permite que essa cor seja convertida no formato Uint32, usado principalmente para funções de renderização do SDL.
Janela	Criar uma janela gráfica a partir de onde obtêm-se a área de renderização.	A janela é a área propriamente dita da interface, onde o processo de renderização ocorre e a captura de eventos acontece.

Fonte: Produção própria

Tabela 6 – Componentes do Pacote Interface - Parte 2

Nome	Funcionalidades	Observações
Renderizador	Renderizar figuras básicas definidas no pacote Base na área de renderização da janela gráfica SDL.	Essa classe é uma das mais importantes do pacote Interface, uma vez que é a responsável por “desenhar” todas as figuras suportadas pela simulação. Sem usar essa classe não seria possível representar graficamente os Objeto da simulação. Conforme já fora citado em seções anteriores, essa classe permite a transformação geométrica das figuras que serão renderizadas através do uso de matrizes.
PlanoCartesiano	Representar e desenhar o plano cartesiano da simulação.	Essa classe faz o uso da classe Renderizador para traçar diversas linhas pela tela visando formar um plano cartesiano conforme especificações do desenvolvedor. Essa classe foi criada de forma a permitir que os eixos possuam diferentes escalas.
Evento	Representar um Evento de teclado em um formato mais amigável do que o apresentado pelo SDL	Essa classe é o componente mais básico no que tange a captura e gerenciamento de eventos, uma vez que seus objetos podem ser usados para representarem um evento que ocorreu no SDL, bem como quem o ativou. O formato de representação é definido pela própria classe e um conjunto de estruturas de enumeração. Esse fato permite que os eventos sejam mais facilmente armazenados e interpretados posteriormente pela classe GerenciadorEvento
GerenciadorEvento	Capturar, armazenar e interpretar os eventos do SDL	Conforme fora feito uma observação anteriormente, essa classe faz uso da classe Evento, sendo responsável por transformar os eventos SDL em um objeto Evento para que esse possa ser interpretado por um método interno do gerenciador.

Fonte: Produção própria

Visando dar sequência ao conteúdo dessa seção, logo em seguida é possível observar algumas imagens da implementação das classes focando em alguns elementos para demonstrar o trabalho aplicado nesse processo. Devido ao fato desse pacote ser consideravelmente extenso, fica difícil de apresentar a implementação completa de todas as classes. Caso seja desejado tal verificação, é possível acessar o repositório do projeto e conferir a documentação do pacote ou visualizar diretamente o código comentado para mais detalhes.

Além disso, visando uma abordagem mais profunda do pacote, uma das classes será abordada em maiores detalhes em seguida. A classe escolhida para ser detalhada é a classe *Renderizador*, uma vez que essa é responsável por um aspecto extremamente importante e interessante da interface gráfica, que é a representação de figuras e possibilidade de representar os Objeto da simulação.

1.6.1 Discussão sobre a classe *Renderizador*

Conforme já foi explicitado anteriormente nessa seção, a classe *Renderizador* é muito importante para a composição da interface devido ao fato de ser o componente responsável pelo desenho de figuras geométricas na tela que permitem, por exemplo, representar um Objeto da simulação que possui formato circular. De maneira geral no que trata-se de implementação, essa classe não é extremamente complexa. Sua estrutura pode ser observada na imagem do diagrama de classes logo abaixo.

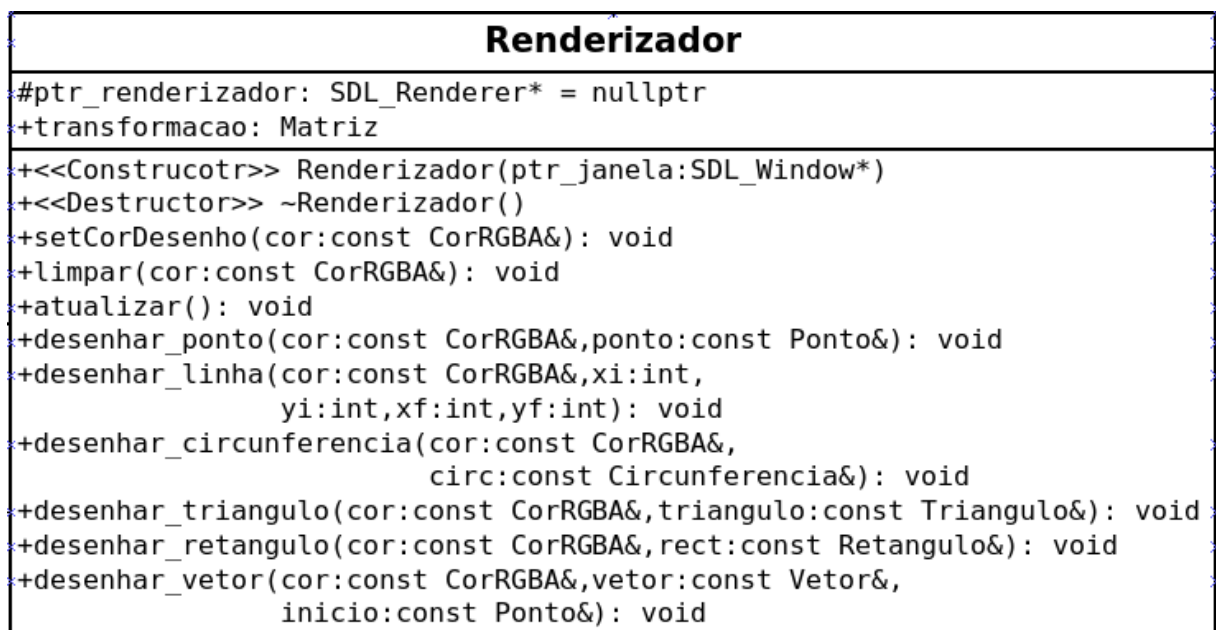


Figura 21 – Diagrama de classes da componente *Renderizador* do Pacote Interface

Fonte: Produção Própria

Como é possível observar essa classe possui apenas dois atributos e uma série de métodos, em sua maioria referente ao processo de renderização de algum elemento geométrico como segmentos de retas, circunferências e vetores. Visando apresentar os principais elementos presentes nessa classe, abaixo estão listados alguns de seus atributos e métodos julgados mais interessantes seguidos por uma explicação da operação desses elementos.

- Atributo Matriz transformação: Conforme explicitado na seção de que trata-se da teoria matemática e física aplicada ao trabalho, esse atributo representa a matriz 3x3 de transformação. Através dela é possível alterar a maneira na qual um elemento geométrico é renderizado. Essa possibilidade permite que todas as renderizações sejam por exemplo, desenhadas considerando o centro da tela como origem, diferentemente do que naturalmente ocorre na computação gráfica onde a origem encontra-se no canto superior esquerdo.
- Método `desenhar_circunferencia`: Esse é um dos muitos métodos de renderização de figuras geométricas implementado pelo renderizador. Todavia dá-se destaque em específico para esse método pelo fato dos Objetos da simulação possuírem formato de circunferência. Assim como fora falado em seções anteriores, esse método consiste em achar diversos pontos pertencentes a circunferência para que retas possam ser traçadas e a circunferência possa ser aproximada graficamente.
- Método `desenhar_vetor`: Outro método de renderização muito utilizado é para renderizar vetores. Conforme fora comentado, os vetores são usados para representar grandezas físicas de natureza não escalar como velocidade. Ao permitir que vetores sejam renderizados, é possível representar a magnitude, direção e sentido da velocidade de um Objeto, o que torna a simulação mais interessante.

Logo em sequência é possível observar algumas imagens acerca da implementação dessa classe, bem como alguns resultados obtidos através do uso dos recursos implementados.

```

171 void desenhar_circunferencia(const CorRGBA &cor, const Circunferencia &circ){
172     setCorDesenho(cor);
173
174     Ponto* demarcacao = circ.demarcacao;
175     Matriz tmp = Matriz(3,circ.precisao);
176
177     for (int i=0; i<circ.precisao; i++){
178         tmp[0][i] = demarcacao[i].x;
179         tmp[1][i] = demarcacao[i].y;
180         tmp[2][i] = 1;
181     }
182
183     tmp = transformacao * tmp;
184
185     for(int i=1; i<circ.precisao; i++){
186         SDL_RenderDrawLine(ptr_renderizador,
187                             tmp[0][i-1],
188                             tmp[1][i-1],
189                             tmp[0][i],
190                             tmp[1][i]);
191     }
192
193     SDL_RenderDrawLine(ptr_renderizador,
194                         tmp[0][circ.precisao-1],
195                         tmp[1][circ.precisao-1],
196                         tmp[0][0],
197                         tmp[1][0]);
198 }
199

```

Figura 22 – Código da classe Renderizador - Método par desenhar circunferências

Fonte: Produção Própria

```

271 void desenhar_vetor(const CorRGBA &cor, const Vetor &vetor, const Ponto &inicio){
272     float xi, xf, yi, yf;
273
274     const float tam_setas = vetor.getModulo()/5;
275     const float angulo = vetor.getAngulo();
276
277     xi = static_cast<int>(inicio.x);
278     yi = static_cast<int>(inicio.y);
279     xf = static_cast<int>(xi + vetor.x);
280     yf = static_cast<int>(yi + vetor.y);
281
282     // Linha Principal
283     desenhar_linha(cor, xi, yi, xf, yf);
284
285     // Linha Secundária 1
286     xi = static_cast<int>(xf + cos(angulo + M_PI*3/4) * tam_setas);
287     yi = static_cast<int>(yf + sin(angulo + M_PI*3/4) * tam_setas);
288     desenhar_linha(cor, xi, yi, xf, yf);
289
290     // Linha Secundária 2
291     xi = static_cast<int>(xf + cos(angulo + M_PI*5/4) * tam_setas);
292     yi = static_cast<int>(yf + sin(angulo + M_PI*5/4) * tam_setas);
293     desenhar_linha(cor, xi, yi, xf, yf);
294 }

```

Figura 23 – Código da classe Renderizador - Método par desenhar vetores

Fonte: Produção Própria

```
24 class Renderizador{
25     protected:
26         SDL_Renderer* ptr_renderizador = nullptr;    //!< Ponteiro para o elemento Renderizador SDL que permite a chamada de funções de renderização
27     public:
28
29         Matriz transformacao;                        //!< Matriz 3x3 usada para Transformações Geométricas nas Figuras
30 }
31
```

Figura 24 – Código da classe Renderizador - Atributos

Fonte: Produção Própria

```
24 // Definição de uma Matriz de Transformação para o processo de Renderização
25 render.transformacao = Matriz(3,3);
26 render.transformacao[0][0] = 1;
27 render.transformacao[0][1] = 0;
28 render.transformacao[0][2] = 400;
29 render.transformacao[1][0] = 0;
30 render.transformacao[1][1] = -1;
31 render.transformacao[1][2] = 300;
32 render.transformacao[2][0] = 0;
33 render.transformacao[2][1] = 0;
34 render.transformacao[2][2] = 1;
35
36 janela.preencherFundo(CorRGBA(25,25,25));
37
38 // Chamada aos métodos do Renderizador
39 render.desenhar_ponto(azul, Ponto(50,50));
40 render.desenhar_ponto(CorRGBA(255,0,0), Ponto(-50,-50));
41 render.desenhar_ponto(verde, Ponto(0,0));
42
43 render.desenhar_linha(CorRGBA(0,255,0), -400,0,400,0);
44
45 render.desenhar_circunferencia(azul, Circunferencia(50, Ponto(0,10), 180));
46
47 render.desenhar_triangulo(branco, Triangulo(Ponto(-50,0), Ponto(50,0), Ponto(0,100)));
48
49 render.desenhar_retangulo(branco, Retangulo(Ponto(0,150),25,25));
50
51 // Alteração da Matriz de Transformação para mudança no processo de renderização
52 render.transformacao[0][0] = 2/10.f;
53 render.transformacao[1][1] = -2/10.f;
54 render.desenhar_vetor(branco, Vetor(60,80), Ponto(0,0));
55
56 render.atualizar();
57 janela.atualizar();
58
```

Figura 25 – Código teste do Renderizador - Chamada dos métodos de desenho

Fonte: Produção Própria

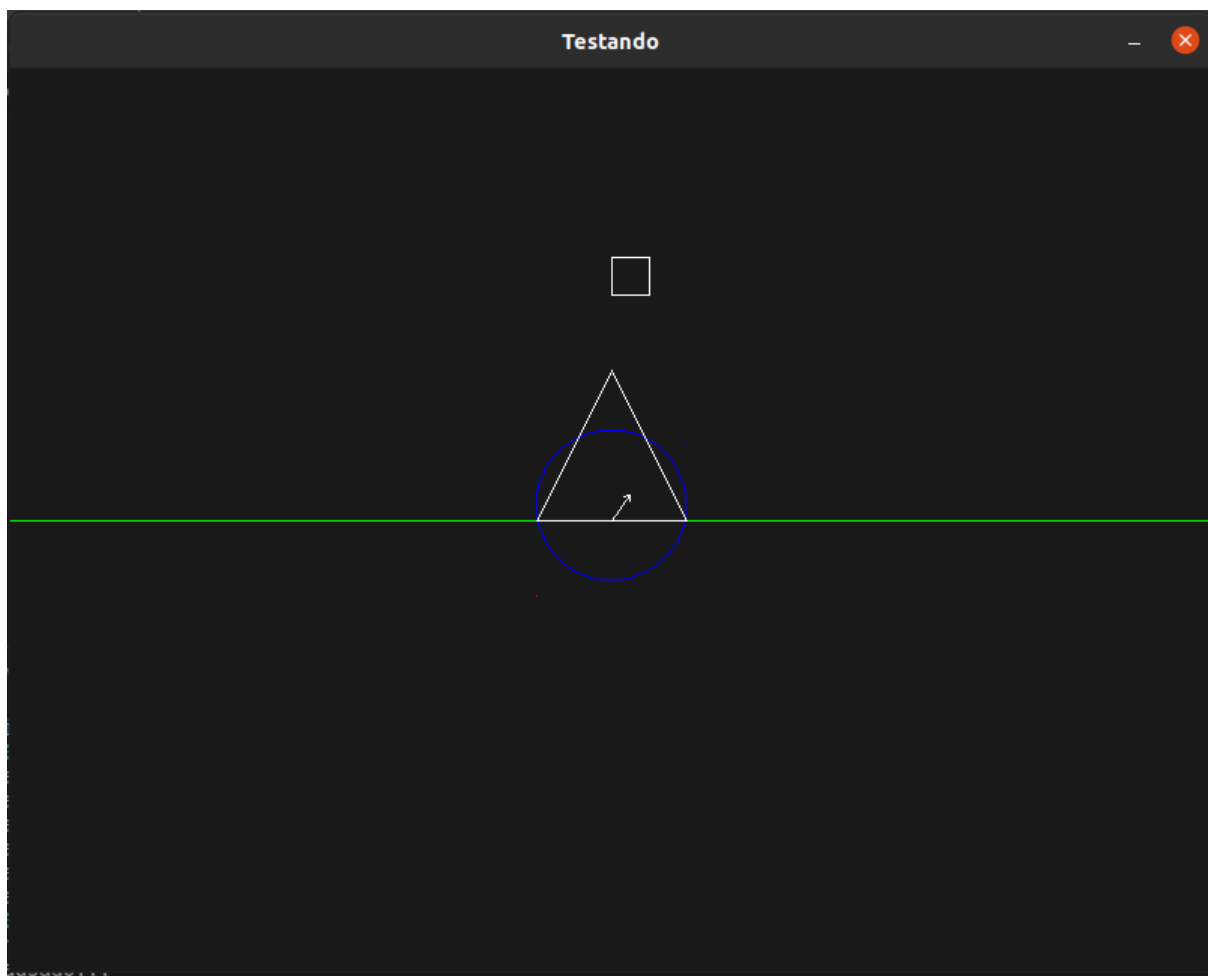


Figura 26 – Resultado da execução do código apresentado na figura anterior

Fonte: Produção Própria

Para encaminhar essa seção para a sua finalização, assim como ocorre no pacote Base o pacote Interface implementa uma serie de testes para averiguar as funcionalidades das classes desenvolvidas. Devido ao fato desses testes muitas vezes serem evoluções de testes de elementos mais simples, juntamente do fato de que mais resultados serão passíveis de observação em seções posteriores, logo abaixo não será apresentado um exemplo de teste, visto que uma das classes mais importante já foi comentada – o Renderizador.

Antes de concluir essa parte do trabalho, é extremamente importante destacar a fonte de embasamento para a construção dos códigos desse pacote. Muito do que foi construído principalmente no que tange às classes mais básicas e comuns, foi baseado nas classes construídas pelo discente responsável pela disciplina. Esses códigos auxiliaram a concretização do pacote Interface e podem ser visualizados no repositório do Gitlab (<<https://gitlab.com/oederaugusto/airhockey>>).

1.7 Desenvolvimento do Pacote Arquivo

Outro processo fundamental do projeto refere-se a manipulação de arquivos para implementação de um mecanismo de persistência de dados para as simulações que fossem desenvolvidas. Devido ao teor introdutório da disciplina, não foi apresentado nem cobrado persistência de dados fazendo-se uso de uma Banco de Dados por exemplo, mas sim por uma abordagem muito mais direta e simples que refere-se à manipulação de arquivos.

Para essa parte do projeto tinha-se como ideia fundamental seguir os mesmos padrões já vistos no desenvolvimento de pacotes citados anteriormente, isto é, desenvolver uma série de classes capazes de resolver o problema proposto, podendo esse problema ser definido como gerenciar arquivos e realizar operações comuns a persistência de dados, como processos de leitura e escrita de dados em um arquivo.

Sendo observada essa necessidade, partiu-se para a criação das classes a partir de um conhecimento já prévio do autor em manipulação de arquivos advindos de experiências com outras linguagens como Python, o que possibilitou elencar as necessidades básicas da classe responsável por gerenciar os arquivos, como seus atributos e métodos. Todavia, devido ao fato de tratar-se de uma outra linguagem foi necessário também a realização do estudo dirigido propriamente para C++ de forma simultânea a todo o processo de desenvolvimento do pacote.

Ao fim do processo de estudo e modelagem da resolução do problema, foi possível realizar a implementação da classe GerenciadorArquivo. Devido ao fato da persistência de dados referir-se a algo relativamente menos complexo no que tange a necessidade de diferentes componentes, a classe supracitada é a única constituinte de seu pacote, sendo responsável por abstrair a camada de manipulação de arquivos ofertada pela linguagem C++ em um conjunto de métodos simples e suficientes para o domínio do projeto.

Tendo sido compreendido a ideia e estruturação básica do pacote denominado arquivo, logo abaixo é possível conferir um leve detalhamento da classe GerenciadorArquivo e suas funcionalidades, bem como algumas imagens de sua implementação que demonstram os resultados obtidos ao longo dessa fase de desenvolvimento.

Tabela 7 – Componentes do Pacote Arquivo

Nome	Funcionalidades	Observações
GerenciadorArquivo	Criar um arquivo conforme especificado ao objeto; Realizar a leitura de uma única linha, ou do conteúdo inteiro do arquivo; Salvar e Recuperar Objetos em um arquivo;	Essa classe foi criada com o propósito principal de permitir ao desenvolvedor salvar Objetos (Componentes que movimentam-se pela tela) da simulação, bem como reutilizar dados de um determinado arquivo para inicializar um conjunto de objetos para uma nova Simulação. Ressalta-se que os objetos de outra classe além de Objeto não devem ser salvos/recuperados usando-se dessa classe.

Fonte: Produção própria

Conforme é possível observar nas observações apresentadas na tabela anterior, a persistência de dados foi focada em objetos da classe Objeto devido ao fato desses serem um dos elementos mais fundamentais presentes no projeto, bem como o desejo de evitar aspectos complexos de diferenciação entre uma linha que refere-se a um objeto da classe X e outra linha que refere-se ao objeto da classe Y.

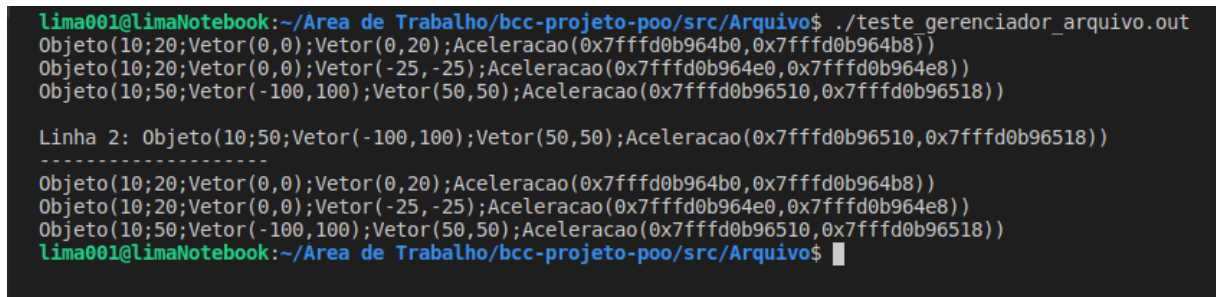
Juntamente da modelagem e implementação da classe, era necessário também realizar a implementação de pequenos testes para averiguar e demonstrar de maneira simplificada o modo de operação de objetos criados a partir da classe GerenciadorArquivo. Da mesma forma que ocorreu nos outros pacotes e implementações de outras classes, esse teste é simples e minimamente comentado possibilitando que aquele que visualize o código possua uma visão do tema e possua a capacidade de utilizar e explorar os códigos ali presentes.

```
Arquivo > C: teste_gerenciar_arquivo.cpp > main()
1  #include "gerenciador_arquivo.h"
2
3  using namespace std;
4
5  double f_const(double t){
6      return 0;
7  }
8
9  int main(){
10     // Criação de um objeto do tipo GerenciadorArquivo para manipular o arquivo teste.txt
11     string nome_arquivo = "teste.txt";
12     GerenciadorArquivo g(nome_arquivo);
13
14     // Criação do Arquivo
15     g.criarArquivo();
16
17     // Criação dos Objetos que serão salvos e recuperados do arquivo
18     Objeto lista_objetos[3];
19     lista_objetos[0] = Objeto(10, 20, Vetor(0,0), Vetor(0,20), Aceleracao(f_const,f_const));
20     lista_objetos[1] = Objeto(10, 20, Vetor(0,0), Vetor(-25,-25), Aceleracao(f_const,f_const));
21     lista_objetos[2] = Objeto(10, 50, Vetor(-100,100), Vetor(50,50), Aceleracao(f_const,f_const));
22
23     g.salvarObjetos(3,lista_objetos);
24
25     g.lerArquivo();
26     cout << endl << "Linha 2: " << g.getLinhaN(2) << endl;
27
28     g.recuperarObjetos(lista_objetos);
29     cout << "-----" << endl;
30     cout << lista_objetos[0] << endl;
31     cout << lista_objetos[1] << endl;
32     cout << lista_objetos[2] << endl;
33
34     return 0;
35 }
```

Figura 27 – Código do teste realizado para a classe Gerenciador Arquivo

Fonte: Produção Própria

Através da execução desse código é possível obter como resultados finais um arquivo de texto conforme apresentado nas figuras 28 e 29, um array de Objetos é recuperados desse mesmo arquivo. Dessa forma observa-se que a implementação da persistência de dados ocorreu com sucesso.

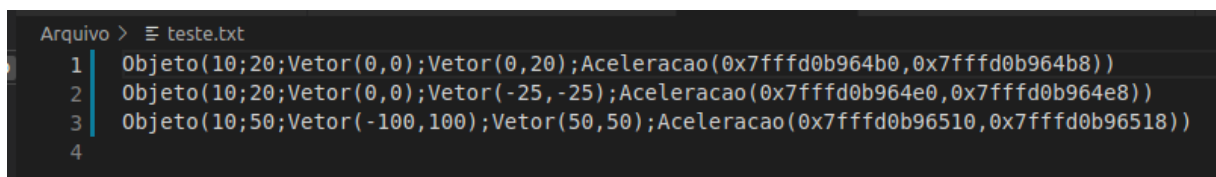


```
Lima001@LimaNotebook:~/Area de Trabalho/bcc-projeto-poo/src/Arquivo$ ./teste gerenciador_arquivo.out
Objeto(10;20;Vetor(0,0);Vetor(0,20);Aceleracao(0x7fffd0b964b0,0x7fffd0b964b8))
Objeto(10;20;Vetor(0,0);Vetor(-25,-25);Aceleracao(0x7fffd0b964e0,0x7fffd0b964e8))
Objeto(10;50;Vetor(-100,100);Vetor(50,50);Aceleracao(0x7fffd0b96510,0x7fffd0b96518))

Linha 2: Objeto(10;50;Vetor(-100,100);Vetor(50,50);Aceleracao(0x7fffd0b96510,0x7fffd0b96518))
-----
Objeto(10;20;Vetor(0,0);Vetor(0,20);Aceleracao(0x7fffd0b964b0,0x7fffd0b964b8))
Objeto(10;20;Vetor(0,0);Vetor(-25,-25);Aceleracao(0x7fffd0b964e0,0x7fffd0b964e8))
Objeto(10;50;Vetor(-100,100);Vetor(50,50);Aceleracao(0x7fffd0b96510,0x7fffd0b96518))
Lima001@LimaNotebook:~/Area de Trabalho/bcc-projeto-poo/src/Arquivo$
```

Figura 28 – Resultado da execução do código apresentado na figura anterior - exibição no console

Fonte: Produção Própria



```
Arquivo > teste.txt
1 Objeto(10;20;Vetor(0,0);Vetor(0,20);Aceleracao(0x7fffd0b964b0,0x7fffd0b964b8))
2 Objeto(10;20;Vetor(0,0);Vetor(-25,-25);Aceleracao(0x7fffd0b964e0,0x7fffd0b964e8))
3 Objeto(10;50;Vetor(-100,100);Vetor(50,50);Aceleracao(0x7fffd0b96510,0x7fffd0b96518))
4
```

Figura 29 – Resultado da execução do código apresentado na figura anterior - exibição do arquivo onde os Objetos foram salvos

Fonte: Produção Própria

1.8 Testes e Integração da Pacotes

Essa etapa refere-se a um processo experimental que ocorreu simultaneamente às etapas de desenvolvimento de todos os Pacotes do projeto descritos anteriormente. Nesse processo o intuito era juntar todos os elementos até então desenvolvidos para a produção de simulações testes visando averiguar o funcionamento correto de todas as componentes no momento em que essas operassem em conjunto. Pode-se entender essa etapa como período pré-desenvolvimento da simulação final detalhada em seção futura.

De maneira geral o trabalho era só juntar as partes conforme essas já haviam sido implementadas e testadas. Os testes de cada classe foi extremamente importante, pois além de garantir que os elementos possuem o funcionamento íntegro quando isolados, também proporcionava uma ideia de como integrar todos os pacotes.

Logo em seguida estão elencados os testes de integração realizados. Também é possível observar algumas imagens a cerca dos resultados obtidos. Entretanto, antes dessa apresentação faz-se necessário comentar um ponto de ressalva. Nem todos os elementos foram integrados nesse período de teste. O pacote responsável pela persistência de dados somente foi utilizado em conjunto dos outros recursos para o desenvolvimento da simulação final. Isso ocorre devido ao fato desse ter sido o último elemento desenvolvido no que tange implementação no trabalho antes de construir a simulação final e não fazia sentido criar um teste que se comportaria muito similar ao resultado final;

Compreendida a ressalva, é possível dar início a apresentação dos testes realizados. Abaixo é possível conferir uma tabela que conta com uma relação entre testes realizados e recursos utilizados a mais em comparação com o último teste implementado conforme sequência apresentada na tabela.

Tabela 8 – Testes realizados na etapa de integração de Pacotes

Nome do Teste (Arquivo)	Recursos Explorados
movimento_simples.cpp	Integração inicial entre Base e Interface; Movimentação do Objeto.
movimento_aleatorio.cpp	Alteração de Velocidade de um Objeto de maneira direta.
movimento_colisao_simples.cpp	Deteção de colisão entre Objetos; Aplicação de efeitos simples de colisão – Inverter sentido da velocidade; Integração com o gerenciamento de eventos.
movimento_colisao_cmx.cpp	Aplicação de colisão com conservação de movimento em movimento em uma dimensão.
movimento_colisao_cmb.cpp	Aplicação de colisão com conservação de movimento em movimento em duas dimensões.
movimento_colisao_borda.cpp	Deteção de colisão entre um Objeto e uma Linha; Aplicação da reflexão especular; Aplicação do cálculo de reposicionamento do Objeto após a colisão com uma Linha.

Fonte: Produção própria

No que tange respeito aos códigos, esses podem ser visualizados no repositório do projeto. Os resultados da execução desses não serão exibidos uma vez que será possível observar o resultado final na próxima seção que assemelha-se aos resultados dessa etapa, considerada as evoluções de complexidade.

1.9 Elementos extras que não estão no Trabalho Final

Nem tudo o que fora pensado e até desenvolvido durante o trabalho chegou a fazer parte da versão finalizada do projeto. Todavia isso não significa que esses elementos desenvolvidos não expressam importância alguma, na verdade pode-se dizer que eles expressam exatamente o oposto, uma vez que o processo de desenvolvimento de um projeto, bem como do conhecimento está relacionado a tentativas e modelagens diversas.

Os recursos que são apresentados nesse seção demonstram linhas de pensamentos uma vez cogitadas para o trabalho. Esses pensamentos são capazes de expressar o nível de desenvolvimento do trabalho em um certo ponto, bem como podem servir de exemplos para futuras mudanças no próprio trabalho, ou até mesmo em outros trabalhos que façam uso de recursos similares.

O elemento julgado mais interessante que não faz-se presente no projeto final, e que é abordado nos próximos parágrafos refere-se a funções lambdas e seu uso no âmbito da definição de uma regra para movimentação com aceleração para um Objeto da simulação.

Recapitulando algumas discussões realizadas durante a abordagem da construção do Pacote Base, um Objeto da simulação deve possuir uma aceleração definida para que através de integração numérica seja calculada a sua velocidade e pelo mesmo processo seja calculada a sua posição no plano da simulação. Visando permitir que o desenvolvedor tivesse a liberdade de definir acelerações não necessariamente constantes, foi adotada a abordagem de calcular a aceleração a partir de funções.

Essa abordagem é extremamente interessante, pois como é possível observar na classe Aceleração é possível definir diferentes funções de aceleração para diferentes objetos em seus diferentes sentidos de movimentação. Todavia existe uma desvantagem ao adotar esse tipo de abordagem. Ao fazer necessária a declaração de funções, o usuário do sistema fica preso a necessidade de implementar por conta própria o seu código para a função, o que pode não ser adequado para diversas aplicações e torna, reduz a capacidade de abstração de certos elementos do código, e torna em certo nível a interação usuário-sistema menos interativa.

Para esclarecer esses pontos supracitados, imagine a ocasião em que almeja-se que o usuário informe certos valores para criar um objeto do tipo Vetor. Conforme já observado em seções anteriores a classe Vetor é muito simples, sendo que seus atributos resumem-se

a dois números que podem ser facilmente informados uma interface de entrada de dados simples – como um terminal.

Agora imagine uma situação similar onde busca-se que o usuário crie um Objeto de maneira interativa, informado os valores de seus atributos via alguma interface para entrada de dados, como por exemplo linha de comando. Uma vez tendo sido observado a necessidade de declaração de uma função C++ para a definição de aceleração, seria inviável ao usuário realizar tal tarefa. De modo geral, existem maneira de parcialmente contornar esse caso através do preestabelecimento de funções para serem usadas, todavia observa-se uma certa perda de flexibilidade do sistema.

Por esse motivo havia sido pensado a criação e uso de funções lambdas. Essas funções consideradas anônimas em C++ possibilitam a criação de funções de ordem superior hábeis a retornar uma nova função. Isso significa que utilizando-se desse recurso é possível construir “construtores de funções”, códigos capazes de construir funções conforme a necessidade via passagem de parâmetros específicos.

Nesse caso seria possível criar “construtores de funções” generalistas capazes de aceitar parâmetros para modelar certas funções de aceleração consideradas mais comuns. Ressalta-se que não seria possível construir toda e qualquer função, mas ao menos seria possível adicionar uma considerável camada de abstração sem afetar consideravelmente a flexibilidade das aplicações mais comuns do sistema.

A princípio havia sido criado um exemplo simples que adota a abordagem comentada anteriormente. O exemplo apresentado na 30 trata-se do uso de funções lambdas para a definição de funções de ordem superior. Nesse caso em questão o código está isolado e não foi aplicado ao contexto específico da simulação, todavia não seria necessário um grande empenho para a adaptação da classe Objeto e da classe Aceleração para a passar a trabalhar com essa abordagem.

```
test.cpp > gerar(int)
1  #include <iostream>
2  #include <functional>
3
4  using namespace std;
5
6  std::function<int (int)> gerar(int x){
7
8      auto func = [=](int valor){return valor * x;};
9      return func;
10
11 }
12
13 int main(){
14
15     std::function<int (int)> dobrar = gerar(2);
16     cout << dobrar(5) << endl;
17
18     return 0;
19 }
```

Figura 30 – Teste realizado com funções lambdas para criar uma função de ordem superior

Fonte: Produção Própria

```
lima001@limaNotebook:~/Area de Trabalho/Copia Poo$ ./a.out
10
lima001@limaNotebook:~/Area de Trabalho/Copia Poo$
```

Figura 31 – Resultado da execução do código apresentado na figura anterior - chamada às funções criadas a partir da função de ordem superior

Fonte: Produção Própria

Entretanto essa abordagem não foi adotada e não faz-se presente na versão finalizada do projeto. Isso ocorre devido ao fato do autor não possuir um elevado conhecimento sobre funções lambdas e a necessidade de tal flexibilização proporcionada por esse recurso não ser extremamente crítica para o projeto atualmente desenvolvido e finalizado. Maioria dos casos, inclusive da simulação final construída, faz-se uso de uma aceleração constante e o usuário que executa a simulação não está responsabilizado pela criação de objetos.

Dessa forma fica destacada nessa seção um outro recurso que poderia fazer parte do projeto, mas acabou de fora devido ao julgamento das reais necessidades de suas funcionalidades. Para aqueles que possuem interesse, o código relacionado ao tema esta também disponível no repositório do projeto para consulta.

Além das funções lambdas, outros elementos ficaram de fora do projeto final. Visando uma rápida apresentação desses, logo abaixo é possível conferir uma lista com alguns dos outros recursos pensados e que não necessariamente foram implementados em um estado mais concreto como o observado nas funções lambda, mas que de toda forma foram cogitados para serem possivelmente implementados na versão final do projeto.

- Desenvolvimento e uso de uma estrutura hashmap no DetectorColisão para armazenar os objetos que sofrem colisão. Essa abordagem permitiria que objetos fossem adicionados e removidos facilmente do motor de colisão. Esse recurso não foi aplicado devido a falta de necessidade de tais operações no contexto atual do trabalho;
- Definição de diversas linhas de controle de borda que não necessariamente representariam as bordas da janela gráfica. Como visto na seção que trata sobre o desenvolvimento do pacote Base, as linhas são usadas para demarcar os limites da janela gráfica e proporcionar colisão para com os objetos impedindo que eles saiam. A intenção inicial era permitir que diversas linhas fossem traçadas ao longo de uma janela gráfica em locais conforme a necessidade e desejo do desenvolvedor, permitindo criar um ambiente de colisão que não limitava-se ao formato retangular padrão da janela. Esse recurso não foi aplicado devido a complicações experimentadas no que tange ao processo de cálculo de colisão entre linhas e objetos. Somando esse fato ao tempo necessário para implementar uma devida solução, não seria possível construir uma implementação adequada desse recurso e dessa forma optou-se pelo método mais simples já discutido na seção sobre o Pacote Base;
- Uso de coordenadas polares ao invés de coordenadas planas;
- Sincronização do tempo de simulação com o tempo real. Esse recurso fora pensado nos momentos iniciais do projeto e consistia em realizar cálculos e correções para adequar o tempo de processamento da simulação com o tempo decorrido no ambiente real. A implementação dessa funcionalidade agregaria coerência a simulação e tornaria ela

mais robusta. Todavia esse recurso não foi implementado uma vez que escolhe-se dar mais atenção para outros aspectos da simulação e deixar o quesito de tempo algo mais simples.

1.10 Documentação

Desenvolver um software não resume-se à simples escrita de diversos códigos. O processo de desenvolvimento de um bom software está relacionado também com a capacidade de que o leitor do seu código possui em compreendê-lo, seja pelo simples desejo e curiosidade de entender como o programa foi escrito, ou até mesmo para uma possível reutilização de códigos. Dessa forma faz-se necessário a documentação dos códigos desenvolvidos.

A documentação do projeto em questão foi desenvolvida utilizando-se da ferramenta doxygen. Essa ferramenta permite que comentários, juntamente de outros recursos como sintaxe própria e sintaxe latex seja compreendida e compilada para uma série de arquivos de documentação. Sabendo-se disso, ao longo do desenvolvimento ocorreu também o processo de documentação do código que intensificou-se ao final do desenvolvimento do projeto, uma vez que os códigos já estavam todos finalizados e o tempo podia ser ministrados especificamente para essa tarefa.

Ao final do projeto todas os arquivos foram comentados, onde os arquivos das classes foram compilados pela ferramenta doxygen e resultaram na documentação final do projeto do simulador. Os comentários realizados são extremamente simples, mas ao mesmo tempo capazes de passar as principais ideias e detalhes de cada elemento essencial do código desenvolvido. A fim de exemplificar e apresentar os resultados obtidos através desse processo, logo em sequência é possível observar um conjunto de imagens que demonstram os comentários realizados em alguns códigos do projeto, bem como alguns resultados gerados pela compilação doxygen.

```
8  /*!  
9  \file objeto.h  
10 \class Objeto  
11 \author Gabriel Eduardo Lima  
12 \date Última Modificação: 27/02/2002  
13 \brief Classe para Representação de um Objeto com forma  
14      de Circunferência capaz de movimentar-se  
15 */  
16  
17 class Objeto {  
18     private:  
19         static int contagem_id;          //!< Contador de id - Atributo da Classe para gerar automaticamente o id de cada Objeto  
20  
21     public:  
22         int id;                          //!< Atributo de identificação única de cada Objeto  
23         float raio_circunferencia;      //!< Raio da circunferencia do Objeto  
24         float m;                        //!< Massa do Objeto  
25         Vetor r;                        //!< Vetor posição do Objeto  
26         Vetor v;                        //!< Vetor velocidade do Objeto  
27         Aceleracao a;                  //!< Aceleração do Objeto - Conjunto de Funções que definem aceleração nos eixos x e y  
28  
29         //!< Construtor Default  
30         /*!  
31         Inicializa o Objeto com todos os valor numéricos nulos, e  
32         faz uso dos construtores default das outras classes para  
33         criar os objetos membros em um estado padrão.  
34  
35         @see Vetor  
36         @see Aceleracao  
37         */  
38         Objeto():  
39             raio_circunferencia(0),  
40             m(0),  
41             r(Vetor()),  
42             v(Vetor()),  
43             a(Aceleracao())  
44     {
```

Figura 32 – Comentário efetuado na classe Objeto que será compilado pelo Doxygen

Fonte: Produção Própria

Atributos Públicos

int	id	Atributo de identificação única de cada Objeto .
float	raio_circunferencia	Raio da circunferencia do Objeto .
float	m	Massa do Objeto .
Vetor	r	Vetor posição do Objeto .
Vetor	v	Vetor velocidade do Objeto .
Aceleracao	a	Aceleração do Objeto - Conjunto de Funções que definem aceleração nos eixos x e y.

Amigas

std::ostream &	operator<< (std::ostream &out, Objeto &objeto)	Sobrecarga do Insertion Operator para Saída de Dados. Mais...
----------------	--	---

Descrição detalhada

Classe para Representação de um Obejto com forma de Circunferência capaz de movimentar-se.

Autor

Gabriel Eduardo Lima

Data

Última Modificação: 27/02/2002

Figura 33 – Resultado da documentação compilada apresentada na figura anterior

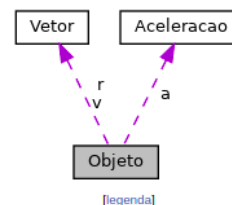
Fonte: Produção Própria

Referência da Classe Objeto

Classe para Representação de um Obejo com forma de Circunferência capaz de movimentar-se. [Mais...](#)

```
#include <objeto.h>
```

Diagrama de colaboração para Objeto:



Membros Públicos

Objeto ()

Construtor Default. [Mais...](#)

Figura 34 – Elemento introdutório da documentação da classe Objeto gerado pelo Doxygen

Fonte: Produção Própria

Aqui estão as classes, estruturas, uniões e interfaces e suas respectivas descrições:

Aceleracao	
Acelerecao	Classe usada para Definir as leis de aceleração que regem a movimentação de um Objeto tanto no sentido vertical quanto horizontal
Circunferencia	Classe para Representação de uma Circunferência. É a forma considerada como padrão para os objetos Objeto da simulação
Controlador_SDL	Classe responsável pela inicialização e possibilitar a operação da biblioteca SDL
CorRGBA	Classe responsável por representar cores RGBA para uso em conjunto aos elementos gráficos da interface SDL
DetectorColisao	Classe para Cálculo e Aplicação de Colisão entre objetos do tipo Objeto e entre um Objeto e uma Linha
Evento	Classe Responsável pela representação de um evento único capaz de ser processado por um objeto do tipo GerenciadorEvento
GerenciadorArquivo	Classe para manipulação de arquivos visando a persistência de dados relacionados a objetos do tipo Objeto
GerenciadorEvento	Classe Responsável pelo processamento dos objetos do tipo Evento
Janela	Classe responsável pela criação e controle da Janela gráfica e seus elementos associados
Linha	Classe para Representação de um Segmento de reta no Plano Cartesiano Usada para definição de limites nos quais os objetos da simulação colidem, e principalmente para o cálculo da reflexão especular
Matriz	Classe para Representação de Matrizes
Objeto	Classe para Representação de um Obejo com forma de Circunferência capaz de movimentar-se
PlanoCartesiano	Classe responsável pela representação gráfica do Plano Cartesiano
Ponto	Classe para Representação de um Ponto no Plano Cartesiano
Relogio	Classe responsável pelo controle dos FPS da aplicação gráfica
Renderizador	Classe responsável pela abstração do processo de renderização de figuras na Área de renderização da janela
Retangulo	Classe para Representação de um Retângulo
Triangulo	Classe para Representação de um Triangulo
Vetor	Classe para Representação de um Vetor no Plano Cartesiano

Figura 35 – Menu de documentação listando todas as classes documentadas, bem como apresentando uma breve descrição conforme comentado no arquivo daquela classe

Fonte: Produção Própria

Além da documentação escrita do código, foi produzido um diagrama UML de classes que serve como representação gráfica da estrutura do projeto através da apresentação das classes presentes no sistema, bem como a ligação expressa entre si. No que tange respeito o desenvolvimento desse diagrama pode-se ressaltar alguns aspectos como:

- Para o seu desenvolvimento foi utilizado o software de diagramação denominado Dia;
- A construção do diagrama ocorreu ao final do projeto quando as classes já haviam sido finalizadas. Dessa forma, diferentemente do que comumente acontece em projetos mais robustos onde o sistema é desenvolvido a partir da modelagem em diagramas e outros elementos pré-desenvolvimento, o diagrama de classes foi construído com o propósito de relatar o programa construído, e não modelá-lo;

Tendo sido analisado todos esses aspectos, logo na continuidade do relatório é possível observar alguns recortes do diagrama de classes, uma vez que seu tamanho ficou consideravelmente grande para ser apresentado inteiramente. Todavia caso deseje-se analisar o diagrama como um todo é possível acessá-lo no repositório do projeto conforme especificações já citadas anteriormente.

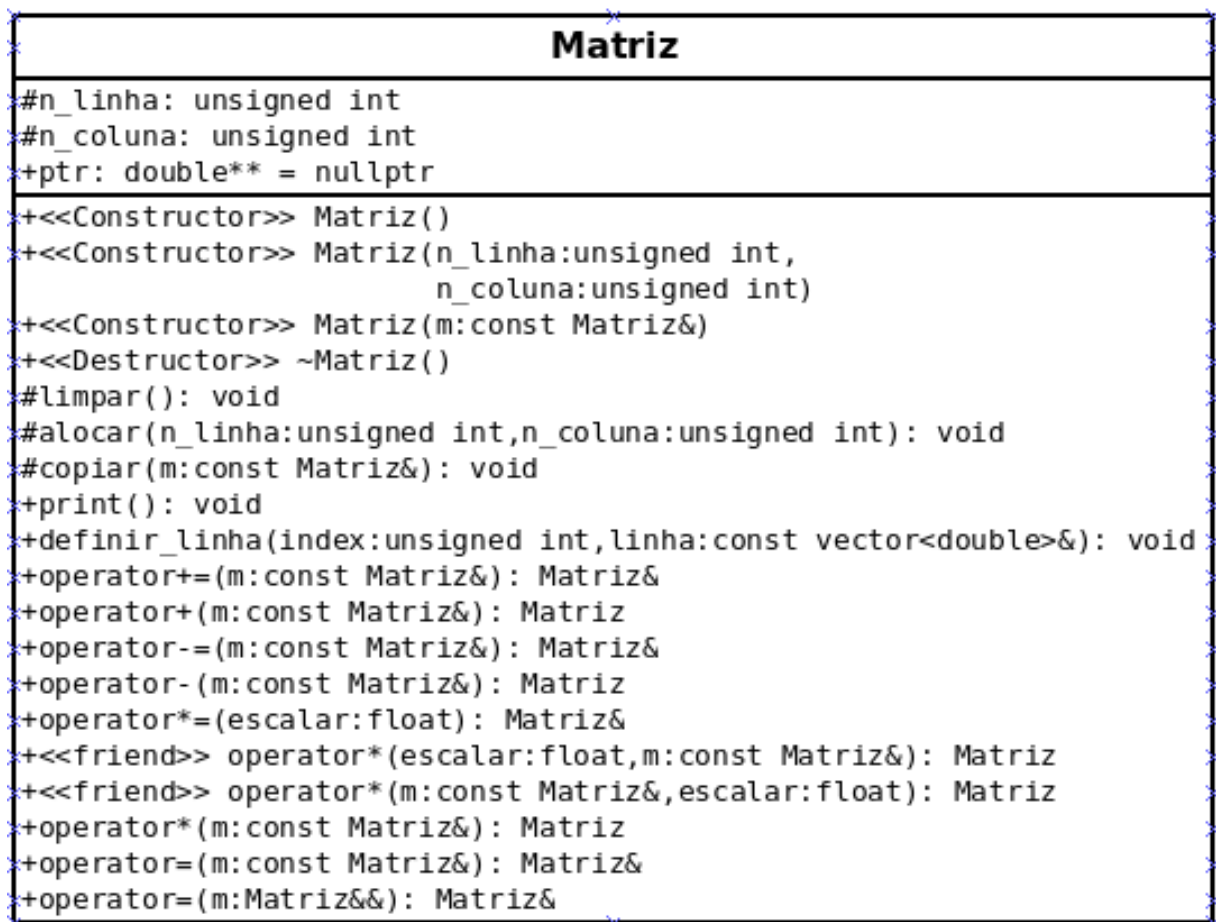


Figura 36 – Diagrama da classe Matriz - Classe muito importante e já citada no trabalho

Fonte: Produção Própria

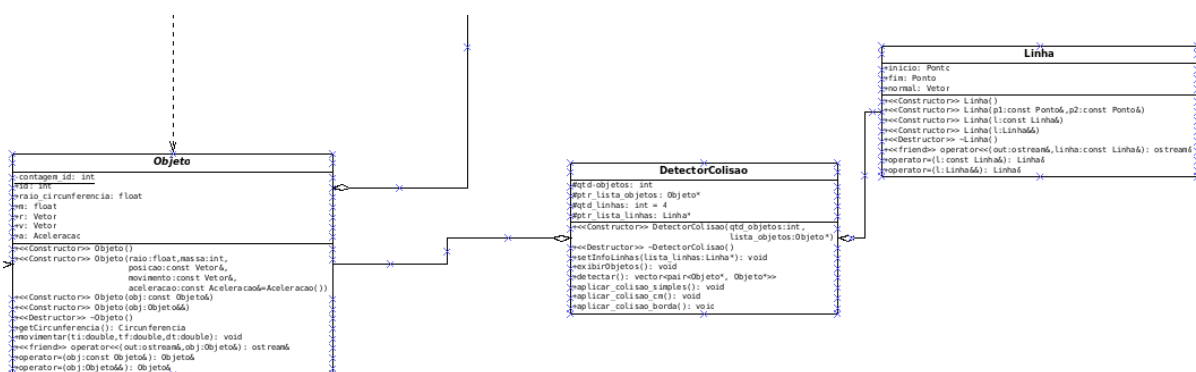


Figura 37 – Recorte das componentes responsável pela colisão e movimentação de corpos na simulação

Fonte: Produção Própria

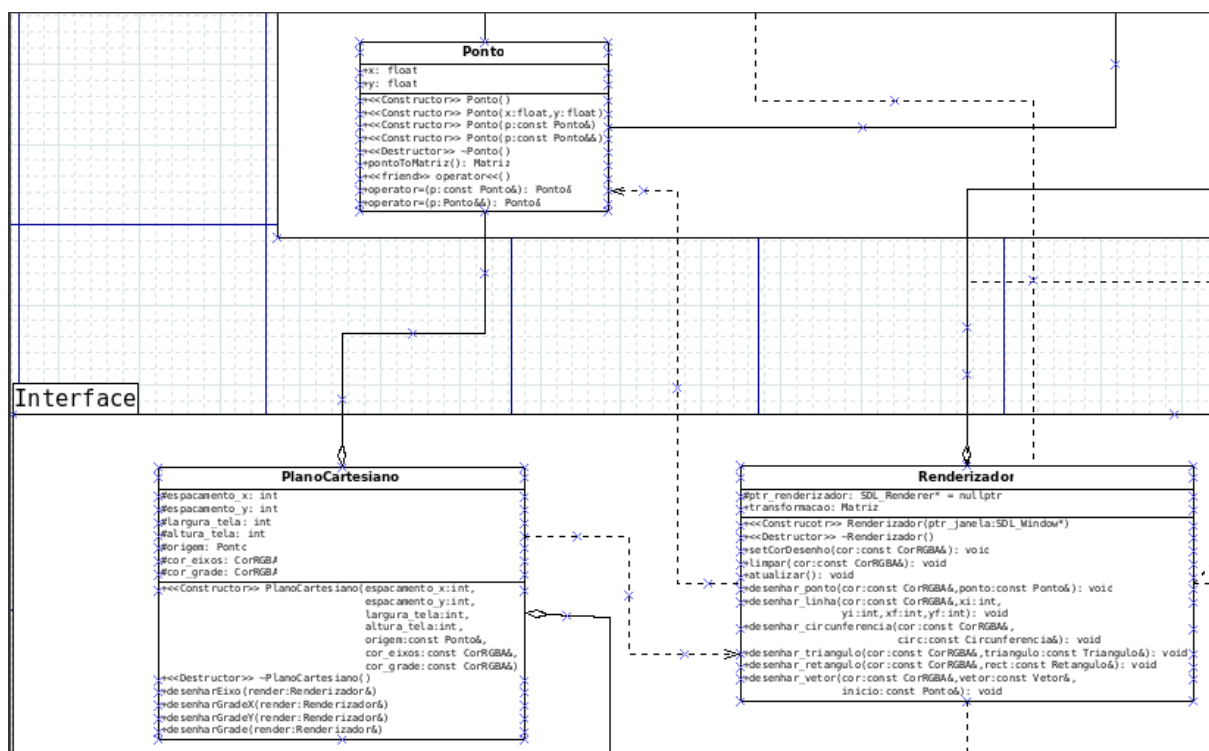


Figura 38 – Recorte da integração entre componentes de diferentes pacotes

Fonte: Produção Própria

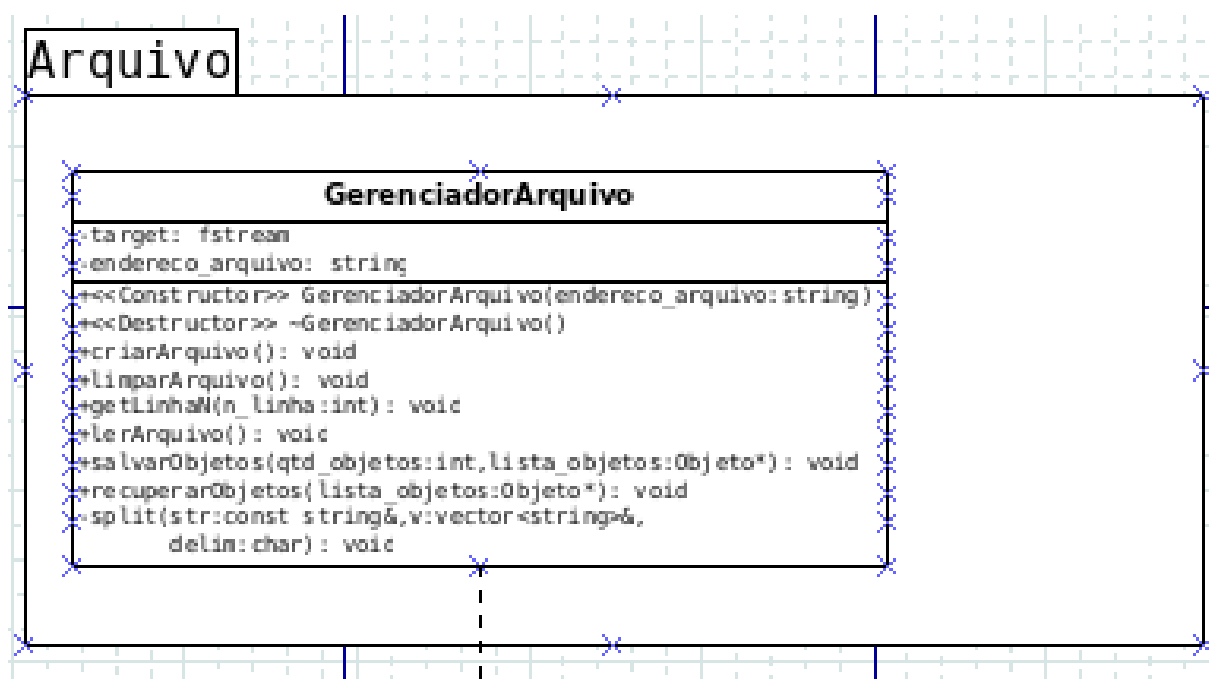


Figura 39 – Diagrama da classe GerenciadorArquivo responsável pela persistência de dados no sistema

Fonte: Produção Própria

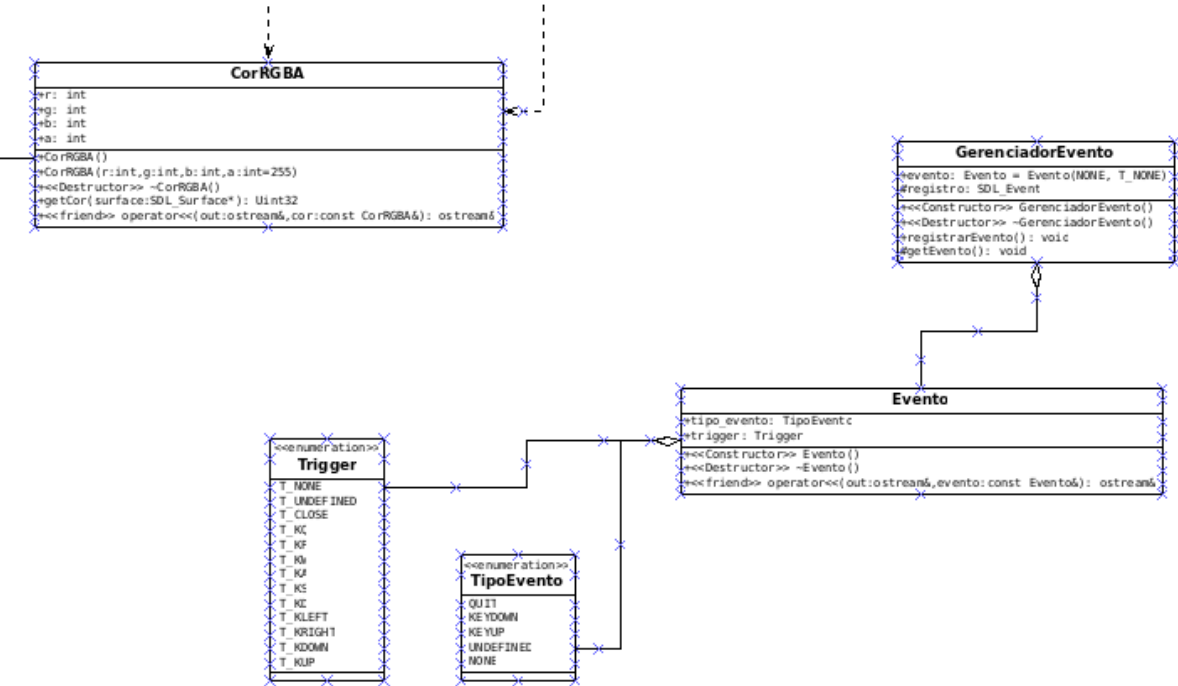


Figura 40 – Recorte das componentes (exceto CorRGBA) responsáveis pelo tratamento de eventos da interface

Fonte: Produção Própria

1.11 Repositório e Simulação Final

O resultado final obtido refere-se a produção de uma simulação contendo todas as componentes desenvolvidas ao longo do projeto. Essa simulação finaliza o projeto e demarca a criação de um repositório com códigos e documentação a cerca de um simulador físico. Todo o projeto e os elementos discutidos nesse relatório podem ser encontrados nesse local. O link para acesso já foi mencionado anteriormente sendo possível observar a simulação final na íntegra.

Abaixo estão elencadas as duas imagens relacionadas à produção final do projeto, sendo a figura 41 referente ao repositório criado na plataforma Gitlab com todos os elementos do trabalho. Já a figura 42 juntamente da figura 43 apresentam o resultado da simulação final conforme sua execução demonstrando o sucesso e conclusão do projeto do simulador.

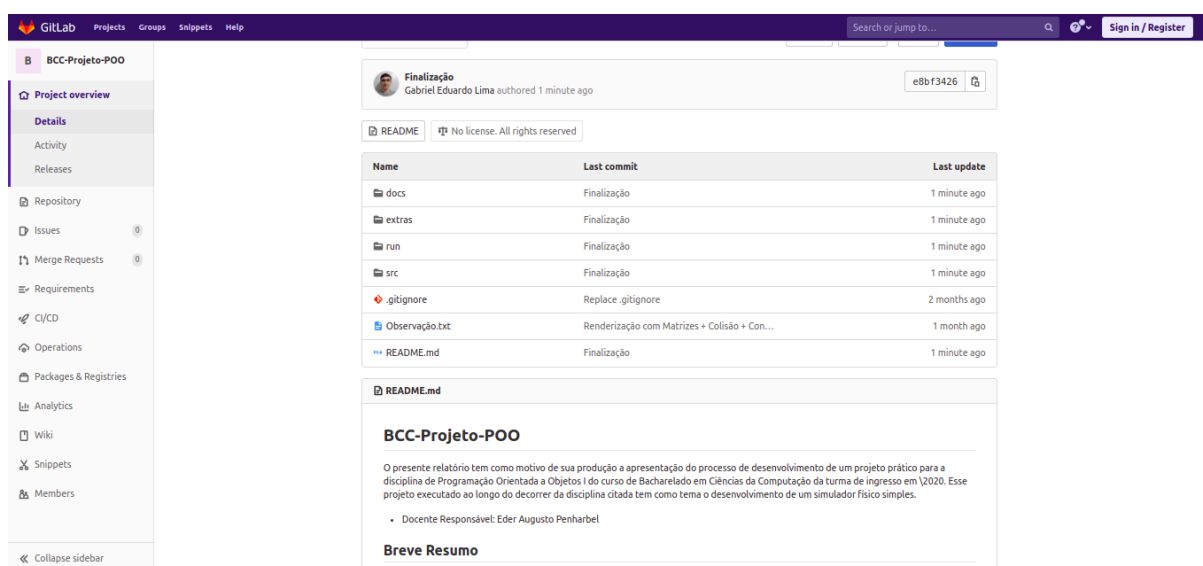


Figura 41 – Imagem do repositório Gitlab finalizado com todos os elementos desenvolvidos durante o projeto para exploração

Fonte: Produção Própria

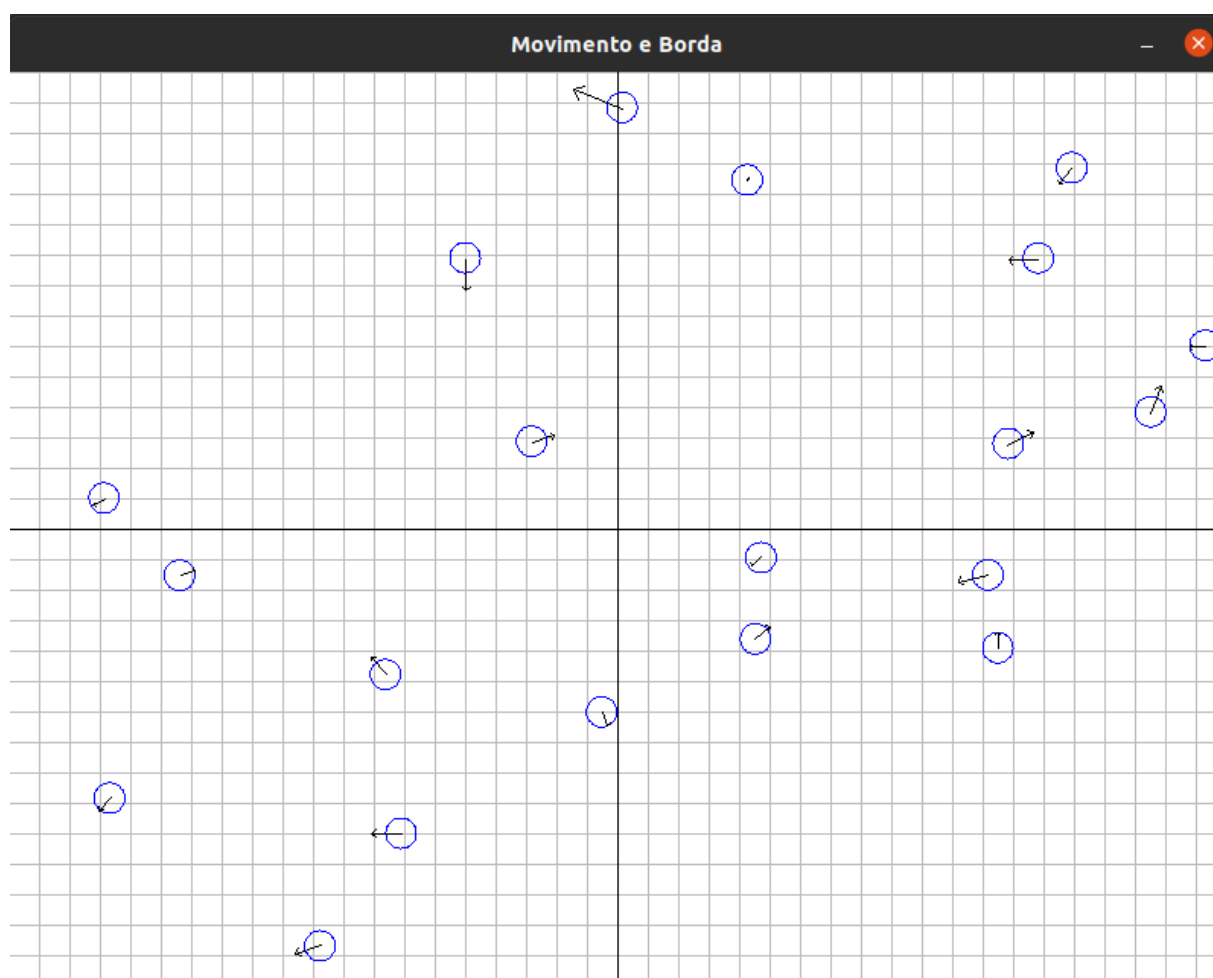


Figura 42 – Momento 1 da simulação final com diversos corpos movimentando-se aplicando tudo o que foi discutido anteriormente

Fonte: Produção Própria

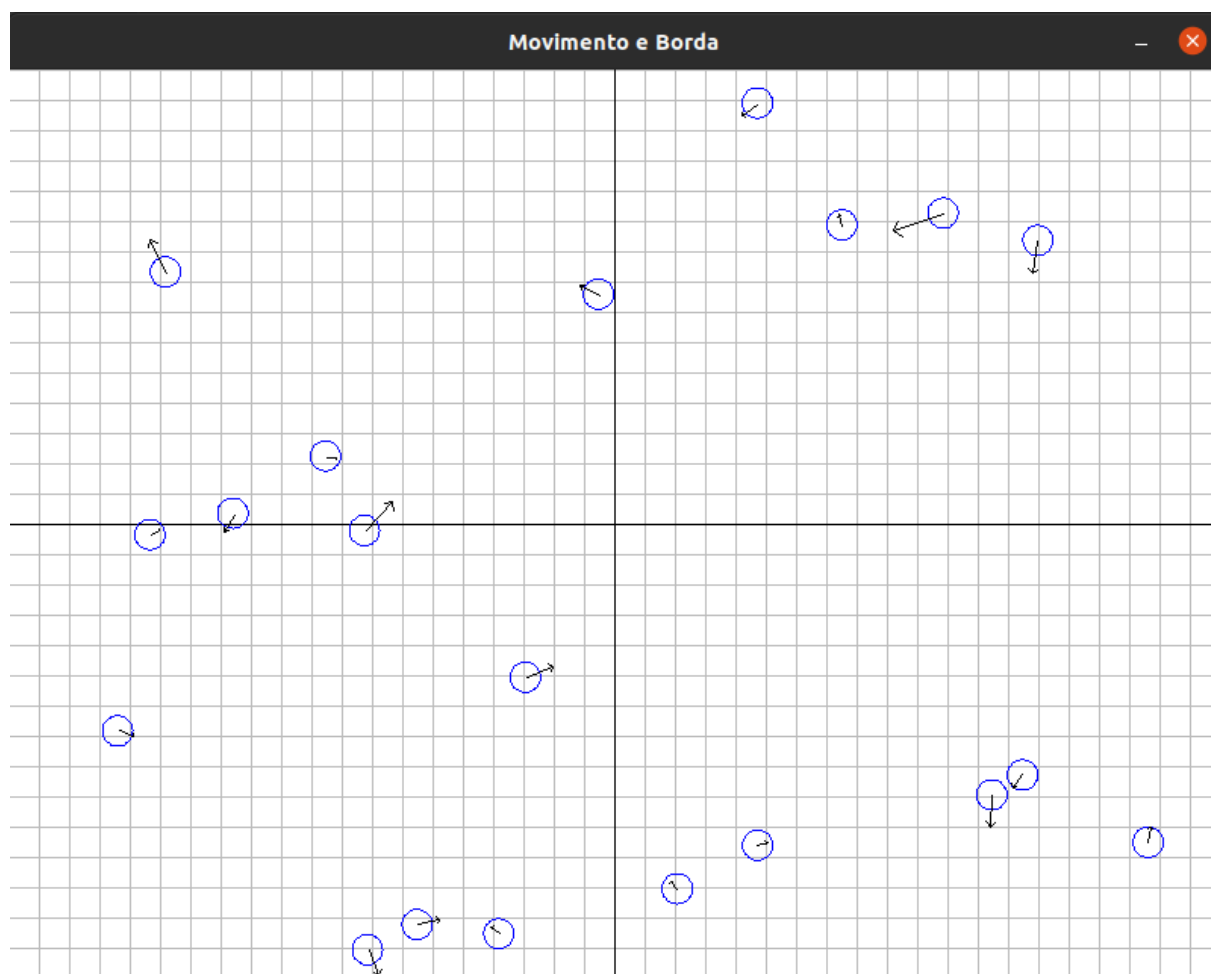


Figura 43 – Momento 2 da simulação final com diversos corpos movimentando-se aplicando tudo o que foi discutido anteriormente

Fonte: Produção Própria

2 Conclusão

Com o seguinte relatório é possível concluir que o desenvolvimento do trabalho consistiu em diversas etapas tanto teóricas quanto práticas, o que possibilitou não somente a aquisição de um conhecimento prático-teórico no que tange a programação orientada a objetos, mas também a diversos conceitos relacionados a base de funcionamento de um simulador físico. Outros tipo de conhecimento também foram possíveis de se desenvolverem ao logo do trabalho como recursos da linguagem C++, processo de documentação usando Doxygen e até mesmo aspectos da engenharia de software como levantamento de requisitos e diagramação do projeto.

Com esse trabalho também é possível observar a importância de áreas como a matemática e a física para a descrição de aspectos reais, bem como a imensa capacidade de interação entre essas áreas com a computação. Esse aspecto de correlação é fundamental para a aprendizagem, pois torna esse processo mais robusto e dinâmico.

Ao fim acredita-se que foi possível executar todos os aspectos previstos nos requisitos, assim como tem-se estabelecido que a execução do projeto prático foi um sucesso e atende as demandas definidas para a disciplina de Programação Orientada a Objetos. Visando finalizar o relatório e trazer algumas informações interessantes sobre o trabalho como um todo, logo em sequência é possível observar alguns tópicos de conteúdo.

2.1 Dificuldades

Como se sabe, nem todos elementos relacionados ao processo de desenvolvimento de um projeto acadêmico como o descrito por esse relatório podem ser considerados como apenas etapas bem sucedidas. O processo de geração do conhecimento e de aprendizagem está relacionado com as tentativas bem sucedidas, assim como está relacionado com as tentativas que resultam em falhas e na capacidade de tirar algum conhecimento a partir delas.

Essa seção da conclusão do trabalho visa apresentar alguns elementos adversos ao projeto que em algum momento causaram uma eventual dificuldade ao autor. De maneira geral pode-se dizer que não houveram ocorrências de muitas dificuldades e que os eventuais acontecimentos restringem-se a aspectos mais simples e não necessariamente comprometedores de todo o projeto. Visando relatar de maneira abrangente os elementos que compuseram o trabalho final, logo em seguida é possível observar as dificuldades vivenciadas e elencadas pelo autor.

- **Frequência de Documentação do Trabalho:** Ao longo do desenvolvimento do projeto tentou-se dar a devida atenção ao processo de documentação dos códigos. Todavia conforme o avançar do desenvolvimento e o considerável aumento de elementos componentes de códigos, bem como a frequente necessidade de alteração o processo de documentação tornou-se mais difícil. Ao final, foi necessário empenhar mais tempo contínuo para a produção da documentação do que necessitaria em caso de uma documentação condizente com o fluxo de desenvolvimento do trabalho;
- **Produção do Diagrama de Classe:** Apesar do autor já possuir algum conhecimento prévio em modelagem UML e diagramas de classes, uma certa dificuldade foi visível no que tange a esse aspecto do trabalho. A dificuldade refere-se a correta modelagem de elementos que nunca haviam sido tratados pelo autor, como por exemplo representar atributos e parâmetros que são ponteiros para funções, algo permitido em C++. Ao final, foi necessária uma dedicação extra no que tange ao estudo da diagramação UML e sua relação com C++.

De maneira geral pode-se concluir que as dificuldades encontradas referem-se a tarefas que o autor não era possuída uma frequência considerável de realização em projetos anterior. Dessa forma, mesmo possuindo conhecimentos básicos sobre o assunto certas dificuldades surgiram pela falta de certas práticas. Entretanto pode-se também afirmar que essas dificuldades não expressam bloqueio para o desenvolvimento do trabalho, uma vez que conforme o seu desenvolvimento o autor é capaz de desenvolver o hábito necessário por aquela atividade.

2.2 Trabalhos Futuros

Devido ao fato introdutório da disciplina, bem como a restrição dessa em um tempo delimitado, existem diversos aspectos do trabalho que possuem potencial de serem desenvolvidos e possivelmente resultarem em versões novas do projeto, bem como trabalhos próprios inteiramente novos. Nessa seção estão elencados alguns dos pontos julgados mais interessantes pelo o autor que podem ser abordados em momentos futuros e que não referem-se a simples correções e otimizações do projeto atual, mas sim adição de funcionalidades.

- **Cálculo de colisão múltipla simultaneamente:** Conforme fora visto em seções anteriores, o cálculo de colisão é efetuado entre 2 objetos por vez. Todavia existem casos em que um objeto colide com mais de um único objeto ao mesmo tempo. Nesse caso seria necessária, por exemplo, a modelagem de um sistemas de equações capaz de resolver essas diversas colisões de maneira correta;

- Simulação com Objetos em formatos diferentes: Os Objetos da simulação ficaram restritos ao formato de circunferência, todavia é interessante permitir simulações com outras formas geométricas como triângulos e retângulos, e inclusive entre Objetos de formatos diferentes. Nesse caso seria necessário, por exemplo, modificar a colisão e incluir efeitos como rotação de figuras;
- Ajuste do tempo da simulação em conformidade ao tempo real: Conforme fora citado anteriormente, existia uma ideia de fazer com que o tempo de simulação correspondesse ao tempo real. Para isso seria necessário a modificação e manipulação do tempo de processamento das instruções do simulador;
- Expansão dos conceitos físicos do simulador: Além de usar os conceitos apresentados e detalhados na seção A Teoria: Matemática e Física Envolvida, seria possível trabalhar com aspectos físicos como forças (de maneira mas explícita), sistemas inelásticos, corpos com velocidade angular, corpos com carga elétrica, campo elétrico e magnético etc.
- Expansão dos conceitos matemáticos do simulador: Além dos próprios conceitos físicos, muita exploração matemática poderia ser efetuada como trabalhar com coordenadas polares, descrição de elementos do sistema através de matrizes (de forma mais complexa), exploração de outros métodos de integração numérica, exploração de conceitos vetoriais etc.
- Simulação no espaço: Conforme pode-se observar todas as simulações são realizadas no plano. Entretanto uma das possibilidades de aprofundamento do trabalho é desenvolver uma simulação com características similares, mas que ocorre no espaço 3D. Nesse caso seria necessária toda uma reformulação do código tanto para a base matemática, como para os elementos de interface;
- Criação de uma interface interativa com o usuário: A simulação final criada consiste em um ambiente pré-configurado pelo autor do projeto. Para alterar a simulação é necessário modificar diretamente componentes dos códigos escritos. Todavia uma possibilidade seria criar uma interface interativa onde um usuário poderia informar valores para configurar certos elementos do código, sendo possível assim a criação interativa e simples de diversas simulações.

2.3 Avaliação do processo de Aprendizagem

Conforme já sabido, o projeto relatado por esse trabalho tem como objetivo ser apresentado para fins avaliativos da disciplina de Programação Orientada a Objetos I do curso em Bacharelado em Ciências da Computação. Entretanto não somente deve-se dar a atenção para aspectos avaliativos, uma vez que a relação entre uma nota considerada

alta e o conhecimento adquirido no desenvolvimento de uma atividade nem sempre são proporcionais. Sendo assim essa seção final do relatório visa apresentar um retorno do conhecimento adquirido independentemente do processo avaliativo.

Primeiramente faz-se extremamente importante citar novamente que no momento em que esse trabalho foi desenvolvido faz aproximadamente um ano que vive-se a pandemia de 2020. Dessa forma a maneira na qual o desenvolvimento do trabalho foi dado difere da habitual vivenciada em períodos anteriores, o que afeta e dita totalmente a forma pela qual o aprendizado é construído. A título de especificação, o desenvolvimento desse trabalho deu-se totalmente de maneira remota, onde o contato com o professor responsável pela disciplina dava-se através de encontros online síncronos e assíncronos.

De maneira geral é possível afirmar que o rendimento do trabalho, bem como o conhecimento adquirido estão de acordo com o esperado em uma disciplina de um curso levado a sério. Acredita-se que foi possível aprender os conceitos e detalhes previstos pela ementa da disciplina, bem como outros assuntos extras relacionados principalmente com o estudo da matemática e da física. Analisando o ponto de vista do autor do projeto, é possível afirmar que a disciplina não perdeu em nada a sua seriedade e compromisso para com o processo de aprendizagem mesmo sendo ministrada em formato incomum (ao seu padrão previsto).

Um dos aspectos que tornam esse aproveitamento positivo refere-se a flexibilidade e autonomia trazida pelo ambiente remoto no que tange ao desenvolvimento das atividades acadêmicas. O aluno passa de um papel comumente passivo para ativo no processo de aprendizagem, onde se bem aproveitado é capaz de produzir resultados extremamente satisfatórios naquilo que refere-se ao conhecimento e por consequência muito provavelmente o resultado é estendido para aspectos avaliativos.

Dessa forma conclui-se que a ideia balizadora do projeto pode ser vista como uma boa ferramenta de aprendizado se bem utilizada pelas partes que a compõem – dedicação discente e docente. Por de mais, deixa-se registrado uma visão satisfatória do término do projeto com a produção de um conhecimento sólido com um potencial considerável de desenvolvimento, bem como uma a mesma visão no que refere-se ao término da disciplina.

Referências

Cplusplus. **Cplusplus.com**. Disponível em: <<https://www.cplusplus.com/>>. Último acesso: 11/03/2021

Doxygen. **Doxygen**. Disponível em: <<https://www.doxygen.nl/index.html>>. Último acesso: 11/03/2021

Geogebra. **Geogebra Calculator**. Disponível em: <<https://www.geogebra.org/calculator>>. Último acesso: 11/03/2021

Gitlab. Disponível em: <<https://gitlab.com/>>. Último acesso: 11/03/2021

Gitlab. **BCC-Projeto-POO**. Disponível em: <<https://gitlab.com/Lima001/bcc-projeto-poo>>. Último acesso: 11/03/2021

Gitlab. **AirHockey**. Disponível em: <<https://gitlab.com/oederaugusto/airhockey>>. Último acesso: 11/03/2021

Learncpp. **Learncpp.com**. Disponível em: <<https://www.learncpp.com/>>. Último acesso: 11/03/2021

STROUSTRUP, B. **Programming: principles and practice using C++**. [s. l.]: Addison Wesley, 2009. ISBN 9780321543721.

STROUSTRUP, B. **The C++ programming language**. 4th ed. [s. l.]: Addison Wesley, 2013. ISBN 9780321563842.

STROUSTRUP, B. **A tour of C++**. [s. l.]: Addison Wesley, 2014. ISBN 0321958314.

UFSC Biblioteca Universitária. **Normalização de trabalhos acadêmicos**. Disponível em: <<https://portal.bu.ufsc.br/normalizacao/>>. Último acesso: 11/03/2021

Wikipedia. **Specular reflection**. Disponível em: <https://en.wikipedia.org/wiki/Specular_reflection>. Último acesso: 11/03/2021

Wikipedia. **Euler method**. Disponível em: <https://en.wikipedia.org/wiki/Euler_method>. Último acesso: 11/03/2021

Wikipedia. **Elastic collision**. Disponível em: <https://en.wikipedia.org/wiki/Elastic_collision>. Último acesso: 11/03/2021

Wikipedia. **Colisão elástica**. Disponível em: <https://pt.wikipedia.org/wiki/Colis%C3%A3o_el%C3%A1stica>. Último acesso: 11/03/2021

Wikipedia. **Movimento uniformemente variado**. Disponível em:
<https://pt.wikipedia.org/wiki/Movimentoz_uniformemente_variado>. Último
acesso: 11/03/2021