UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE ENGENHARIA ELÉTRICA
CURSO DE GRADUAÇÃO EM ENGENHARIA DE
CONTROLE E AUTOMAÇÃO

SIMULAÇÃO INTERATIVA DE UM DRONE NO NAVEGADOR WEB UTILIZANDO WEBASSEMBLY E WEBGL

MIGUEL RAVAGNANI DE CARVALHO

UBERLÂNDIA, MG

2023

MIGUEL RAVAGNANI DE CARVALHO

SIMULAÇÃO INTERATIVA DE UM DRONE NO NAVEGADOR WEB UTILIZANDO WEBASSEMBLY E WEBGL

Trabalho de Conclusão de Curso da Engenharia de Controle e Automação da Universidade Federal de Uberlândia - UFU - Campus Santa Mônica, como requisito parcial para a obtenção do título de Bacharel em Engenharia de Controle e Automação.

UBERLÂNDIA, MG

2023

Miguel Ravagnani de Carvalho

Simulação interativa de um drone no navegador Web utilizando WebAssembly e WebGL

Trabalho de Conclusão de Curso da Engenharia de Controle e Automação da Universidade Federal de Uberlândia - UFU - Campus Santa Mônica, como requisito parcial para a obtenção do título de Bacharel em Engenharia de Controle e Automação.

Trabalho aprovado em 3 de Fevereiro de 2023.

COMISSÃO EXAMINADORA

Prof. Éder Alves de Moura
Orientador

Prof. Dr. Renato Santos Carrijo
Membro Avaliador

Laura Ribeiro
Membro Avaliador

Uberlândia, MG
2023

Agradecimentos

Agradeço aos professores da Universidade Federal de Uberlândia que cruzaram meu caminho, pelo esmero ao lecionar, pela sabedoria compartilhada e pela paciência.

Agradeço à toda minha família, mas principalmente a meu pai, Edvaldo, por ser meu exemplo absoluto de caráter e honestidade; à minha mãe, Luciana, por todo o apoio emocional e carinho incondicionais, e por ter me emprestado seu senso de humor; à minha irmã, Beatriz, por todo o amor e companheirismo que nunca deixou de oferecer, e pelos conselhos que nunca ei de esquecer; à minha maior companhia, Sofia, pela força de vontade e propósito que deu ao meu futuro; à Matheus, pelo conhecimento que apenas horas de conversa poderiam gerar.

Por fim, agradeço a todo Grupo do RU, pela companhia nos corredores da faculdade, pelas risadas de perder o fôlego e por fazerem de cada fase difícil apenas um problema passageiro, e estendo, em especial, estes agradecimentos aos meus amigos, Enivaldo e Victoria, que estiveram ao meu lado desde o começo do curso.

"Toda coisa no mundo feita por alguém começou com uma ideia. Então apanhar uma poderosa o suficiente para te apaixonar, é uma das experiências mais belas. É como ser atingido por eletricidade e conhecimento ao mesmo tempo."
(David Lynch)

Resumo

JavaScript é a linguagem padrão para a implementação da lógica em páginas web e permite a manipulação dinâmica dos elementos da página. No entanto, como é uma linguagem interpretada, o desempenho pode ser limitado. Com o crescente interesse em aplicações web, muitos programas precisam ser convertidos para o formato *JavaScript*, *HTML* e *CSS*, o que pode ser um desafio, especialmente para programas mais complexos. Neste cenário surge a proposta do padrão *WebAssembly*: uma linguagem de baixo nível que define um formato de código binário destinado à execução em navegadores, de forma portátil, com alto desempenho e segurança. A proposta deste projeto é elaborar um estudo sobre a tecnologia *WebAssembly* de modo prático, realizando a portabilidade de uma aplicação nativa para a *Web*. A aplicação que será o objeto de estudo é a simulação da dinâmica e do controle posição de um drone, com navegação simplificada e restrita a um plano vertical, onde uma primeira versão, escrita em C++/*OpenGL*, será convertida em uma *Web App*, utilizando o conjunto *WebAssembly*/*WebGL*. Esse estudo objetivou entender os passos necessários para conversão de aplicações nativas em *Web Applications* e, com resultado, apresentamos os passos necessários ao processo de conversão, a partir do conjunto C++/*OpenGL*, com o detalhamento das ferramentas de software, bem como, das configurações requeridas e a simulação em suas duas versões.

Palavras-chave: WebAssembly; WebGL; simulação de drone; C++; OpenGL.

Abstract

JavaScript is the standard language for implementing the logic in web pages and allows for dynamic manipulation of page elements. However, as an interpreted language, performance may be limited. With growing interest in web applications, many programs must be converted to the JavaScript, HTML, and CSS format, which can be a challenge, especially for more complex programs. This scenery arises the proposal of the WebAssembly standard: a low-level language that defines a binary code format intended for execution in browsers, in a portable, high-performance and secure way. The proposal of this project is to elaborate a study on the WebAssembly technology practically, performing portability of a native application to the Web. The application that will be the object of study is the simulation of the dynamics and position control of a drone, with simplified navigation and restricted to a vertical plane, where a first version, written in C++/OpenGL, will be converted into a Web App, using the WebAssembly/WebGL set. This study aimed to understand the steps necessary for conversion of native applications to Web Applications and, as a result, we present the necessary steps in the conversion process, starting from the C++/OpenGL set, with the detailing of the software tools, as well as the required configurations and the simulation in its two versions.

Key-words: WebAssembly; WebGL; drone simulation; C++; OpenGL.

Lista de ilustrações

Figura 1 – Modelo de forças aplicadas no drone. 25
Figura 2 – Forças de torque resultante. 25
Figura 3 – Arquitetura de alto nível de compilação e execução da simulação. . . . 30
Figura 4 – Diretórios com a totalidade do código-fonte escrito para este trabalho. 31
Figura 5 – Código main.cpp. 31
Figura 6 – Trecho de código da aplicação gráfica para *sprites* estáticas. 33
Figura 7 – Trecho de código da aplicação gráfica para *sprites* animadas. 34
Figura 8 – Organização de dependências externas em repositório GIT. 36
Figura 9 – Laço de execução Emscripten. 38
Figura 10 – Servidor HTTP. 39
Figura 11 – Arquivo docker-compose.yml. 39
Figura 12 – Aplicação: Menu. 40
Figura 13 – Aplicação: Controle manual. 41
Figura 14 – Aplicação: Controle automático. 41
Figura 15 – Aplicação: Navegador Google Chrome. 42
Figura 16 – Aplicação: Navegador Mozilla Firefox. 42

Lista de abreviaturas e siglas

API	Application Programming Interface
CAD	Computer-Aided Design
CAM	Computer-Aided Manufacturing
CSS	Cascading Style Sheets
CPU	Central Processing Units
DOM	Document Object Model
GCC	GNU Compiler Collection
GLEW	OpenGL Extension Wrangler Library
GPU	Graphics Processing Units
GLUT	The OpenGL Utility Toolkit
HTTP	Hypertext Transfer Protocol
HTML	Hypertext Markup Language
JIT	Just-In-Time
JVM	Java Virtual Machine
LLVM	Low Level Virtual Machine
NaCl	Native Client
PD	Proporcional-Derivativo
QUIC	Quick UDP Internet Connections
RK4	Runge-Kutta de quarta ordem
SDL	Simple DirectMedia Layer
TI	Tecnologia da Informação
VAO	Vertex Array Object
VBO	Vertex Buffer Object

WebGL	Web Graphics Library
W3C	World Wide Web Consortium
WASM	WebAssembly
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language

Sumário

1 Introdução **12**
1.1 Justificativas 12
1.2 Objetivos 13
1.3 Organização 13
2 Referencial Teórico **14**
2.1 Condições históricas para o surgimento da tecnologia WebAssembly 14
2.2 A tecnologia WebAssembly 17
2.2.1 Arquiteturas de virtualização 17
2.2.2 Módulos 18
2.2.3 Memória 18
2.2.4 Tabela 19
2.3 OpenGL e WebGL 20
2.3.1 GLEW 20
2.3.2 GLFW 20
2.3.3 GLM 21
2.3.4 WebGL 21
2.4 Compilação cruzada 22
2.4.1 Gnu Compiler Collection 22
2.4.2 Emscripten 23
2.4.3 CMake 23
2.5 Estratégia de virtualização de recursos em contêineres 24
2.5.1 Docker e docker-compose 24
2.6 Modelagem dinâmica de um Drone 25
3 Metodologia **30**
3.1 Simulação C++ e OpenGL 31
3.1.1 Sistema de render de sprites 32
3.1.2 Simulador de um sistema físico com métodos numéricos 34
3.1.3 Recursos visuais 35
3.2 Configuração para desenvolvimento em WebAssembly 35
3.3 Conversão entre C++/OpenGL e WebAssembly/WebGL 37
3.3.1 Laços de execução Emscripten 38
3.4 Execução 38
4 Resultados **40**
5 Conclusão **43**

Referências Bibliográficas 44
ANEXO A Anexos 46
ANEXO B Anexos 49
ANEXO C Anexos 53
ANEXO D Anexos 56
ANEXO E Anexos 61

1 Introdução

Este trabalho apresenta uma simulação interativa da dinâmica de um drone desenvolvido como uma aplicação nativa em C++/*OpenGL* e, posteriormente, foi portada para ser executada nos navegadores *Web* utilizando a tecnologia *WebAssembly*/*WebGL*.

1.1 Justificativas

Aplicações para plataformas *Web* estão cada vez mais complexas — exigentes não apenas em desempenho, mas também em segurança, portabilidade e estabilidade — nutrem a demanda por linguagens de programação de baixo nível (*low-level code*) que possam ser executadas em navegadores, tal como descorre Haas et al. (2017). As primeiras soluções propostas envolviam tradução de código escrito em linguagens compiladas para código em *JavaScript*, mas esse processo introduzia diversos problemas. Um destes obstáculos é diferença de sintaxe e semântica, já que as linguagens compiladas podem possuir sintaxes, e até mesmo divergência de paradigma com o JavaScript, o que pode tornar o processo de reescrita mais complexo e propenso à erros. Há também a limitação de desempenho de uma linguagem interpretada, como o *JavaScript*, e falta de compatibilidade, já o código escrito em linguagens compiladas pode depender de bibliotecas ou recursos que não são nativamente compatíveis com o *JavaScript*.

WebAssembly é uma resposta à tal demanda, pois consegue produzir um formato de código binário leve, de fácil interpretação e não dependente de *Hardware* ou plataforma. Além disso, essa tecnologia viabiliza aplicações com alto grau de segurança para aplicações naturalmente pouco protegidas, pois impõe tipos e limita o fluxo de controle — além de oferecer certo grau de proteção para linguagens compiladas (ainda que limitado) e que, nativamente, não possuem *garbage-collection*, ou coleta de lixo de memória: recurso responsável pelo gerenciamento automático da alocação de memória, e recuperam áreas alocadas durante a execução do programa, mas já não são mais utilizadas.

Este trabalho tem como motivação o estudo da tecnologia *WebAssembly*, sua aplicação em um simulador de um sistema físico, desenvolvido em C++ e OpenGL, a execução deste simulador em navegadores, e a documentação de ferramentas e práticas de desenvolvimento em computação gráfica para a *Web*.

1.2 Objetivos

Este trabalho propõe a transposição de uma simulação de um drone que se move no espaço bidimensional, inicialmente desenvolvido como uma aplicação nativa, para sua execução em navegadores *Web*, usando a mesma base de código para o *bytecode* do *WebAssembly*.

A aplicação consiste em um sistema interativo, na qual o do usuário opta por dois modos de controle da trajetória do drone. Por meio deste desenvolvimento, é almejado estabelecer uma proposta de esqueleto para um ambiente de desenvolvimento WebAssembly, que permita gerenciamento de dependências, versionamento e independência do sistema operacional nativo.

O simulador foi originalmente desenvolvido em C++, com o uso da *API* gráfica OpenGL — especificamente, *Modern OpenGL*, para haver proveito do desempenho inerente de uma ferramenta gráfica de baixo nível. Então, essa aplicação for portada para *WebAssembly* e *WebGL*, e ao fim do desenvolvimento uma extensa do desenvolvimento da aplicação foi elaborada.

1.3 Organização

O trabalho será organizado pela seguinte estrutura:

- Capítulo 2: Referencial teórico para introdução dos conceitos necessários para compreensão das tecnologias utilizadas, bem como definir o fio condutor motriz do projeto. Também há uma contextualização histórica e revisão bibliográfica;
- Capítulo 3: Descrição da metodologia aplicada no desenvolvimento e documentação detalhada dos trechos de código cruciais par compreensão da proposta do trabalho;
- Capítulo 4: Resultados alcançados;
- Capítulo 5: Conclusão;
- Anexos: Trechos extensos de código fundamentais para compreensão do projeto, mas que eram grandes demais para incorporar o capítulo 3.

2 Referencial Teórico

O objetivo desse capítulo é apresentar conceitos-chave das principais tecnologias utilizadas na proposta de projeto. Serão abordados conceitos de compilação *WemAssembly* (conhecida como WASM), computação gráfica com OpenGL e a simulação de um drone no espaço 2D.

2.1 Condições históricas para o surgimento da tecnologia WebAssembly

A característica de acelerada demanda por adaptabilidade de sistemas *Web* foram a base para a forma que assumiria o desenvolvimento de *software* para navegadores. É esperado de um sistema *Web* que seja executável em qualquer plataforma, independente de sistema operacional e que o esforço do usuário seja mínimo para sua execução, já que os critérios de usabilidade e experiência de usuário estão entre os mais determinantes. Isso acarreta necessidade de estratégias de manutenção ágil, robusta e constante. De tal forma, as tecnologias que cumpriam com tais requisitos dominaram o mercado do desenvolvimento de *software* para *Web*. Esta seção aborda a evolução da tecnologia *Web* e como se deu o contexto histórico para o surgimento do *WebAssembly* (SLETTEN, 2022).

Para compreender as tecnologias que formam a base do desenvolvimento *Web*, é preciso definir o protocolo "*The Hypertext Transfer Protocol*", ou HTTP. A história do HTTP pode ser rastreada até 1989, quando Tim Berners-Lee, um cientista da computação britânico, propôs um sistema para compartilhar documentos científicos usando hipertexto. O sistema foi batizado de *World Wide Web* e o protocolo usado para transferir documentos era o HTTP. A primeira versão do HTTP, HTTP/0.9, apenas suportava a recuperação de um único documento por vez. A introdução do HTTP/1.0 se dá em 1992 — nesta, há o suporte à múltiplos pedidos e respostas, bem como cabeçalhos e cache integraram o protocolo. As consequências do mais recente avanço foram transferências de dados de forma mais eficiente e maior velocidade ao navegar por páginas da *Web*. Em 1999, o HTTP/1.1 foi lançado, o que aumentou o desempenho, adicionando suporte a conexões persistentes, uso de solicitações condicionais e capacidade de incluir múltiplas mensagens em uma única solicitação. Essa versão do HTTP é amplamente usada hoje e se tornou o padrão para a comunicação na *Web* (GOURLEY et al., 2002). O HTTP/2, foi lançado em 2015 e introduziu novos recursos como multiplexagem, compressão de cabeçalho e *push* de servidor. Atualmente, a versão proposta mais recente é o HTTP/3. Originalmente batizado de "HTTP/2 *Semantics Using The QUIC Transport Protocol*", o protocolo substitui o

TCP como camada de transporte em favor do QUIC (*Quick UDP Internet Connections*), um protocolo de comunicação baseado em UDP (PERNA et al., 2022).

Porém, o aspecto principal que descreve o comportamento do HTTP expõe uma de suas principais limitações. As interações sejam modeladas como *requests* que devem chegar ao servidor para, só então, retornar ao usuário. Em uma comunicação de alta interatividade e fluxo de informações, como em aplicações modernas, este modelo introduz latência significativa, e pode comprometer a experiência de usabilidade. Portanto, uma tecnologia que permitisse execução de código por parte do usuário poderia resolver parte do problema (KEITH, 2005).

JavaScript é uma linguagem de programação de alto nível, criada em 1995 por Brendan Eich, enquanto ele trabalhava na Netscape Communications Corporation. Originalmente, foi projetada para ser usada principalmente para a criação de roteiros de cliente, ou seja, códigos executados no navegador do usuário, e não no servidor.

A linguagem já foi brevemente chamada de *Mocha*, antes que fosse renomeada para *LiveScript*. Mais tarde, foi renomeada novamente para JavaScript, como uma tentativa de capitalizar a popularidade da linguagem de programação Java, que estava ganhando popularidade na época, e não tardou a conquistar espaço como uma das principais linguagens de programação utilizadas na web. Até então, outras linguagens propostas para desenvolvimento *front-end*, como *Dart* (desenvolvido pelo Google) e o *TypeScript*, não ultrapassaram o território dominado pelo *JavaScript* (PEREIRA et al., 2017). Com sua populariade crescendo, a Netscape decidiu apresentar a linguagem como uma especificação padrão aberta, e em 1996, a ECMAScript foi criada. ECMAScript é uma especificação formal de JavaScript, mantida pela Ecma International. Em novas versões da linguagem foram acrescentadas novas funcionalidades, como suporte para orientação a objetos (ECMAScript 3) e melhorias na manipulação de eventos (ECMAScript 5).

Em conjunto com o *JavaScript*, o desenvolvimento de páginas *Web* é baseado no uso do HTML, ou "*Hypertext Markup Language*", e do CSS, ou "*Cascading Style Sheets*)", como determinado pelo W3C (*World Wide Web Consortium*), uma organização internacional que tem como objetivo desenvolver tecnologias para a *Web*. Fundado em 1994 por Tim Berners-Lee, o W3C trabalha com empresas, organizações governamentais e indivíduos para criar padrões abertos e interoperáveis para a *Web*. Esses padrões incluem HTML, CSS e *JavaScript*, bem como outros formatos de dados e protocolos. O objetivo do W3C é garantir que a *Web* seja acessível para todos, independentemente de plataforma ou dispositivo (COMPUTER. . . , 2005). O HTML é uma linguagem de marcação utilizada para criar páginas da web. A tecnologia HTML permite criação de estruturas e *layout* de uma página *Web*, adicionando elementos como títulos, parágrafos, links, imagens e outros recursos multimídia. A combinação do HTML com o CSS (uma linguagem de estilo composta por regras, que permitem que desenvolvedores especifiquem como os elementos

de uma página web devem ser exibidos, incluindo suas cores, fontes, tamanhos e posições) e o *JavaScript* teceram o que é entendido como desenvolvimento *Web* moderno (KEITH, 2005).

Porém, as limitações intrínsecas da linguagem tornaram-se mais aparentes com o tempo, e para casos em que recursos de baixo nível eram necessários, como em *codecs* de transmissão de vídeo, aplicações gráficas e códigos com múltiplas *threads*.

Segundo Sletten (2022), uma das primeiras propostas de nativização de código para navegadores foi feita pelo Google, em 2011. O projeto de código aberto era chamado de *Native Client* (NaCl). O projeto estabelecia um ambiente controlado e de recursos limitados que permitia a execução de código em navegadores, com velocidade próxima à execução de *bytecode* em ambiente nativo. Era baseado na tecnologia da *toolchain* LLVM (*Low Level Virtual Machine*). Esta trata-se de uma infraestrutura de compilação de código aberto que fornece ferramentas e bibliotecas que otimizam e produzem *bytecode* para diferentes arquiteturas de processador. Pode ser definido como uma plataforma flexível e escalável que pode ser usada para a compilação de uma variedade de linguagens de programação, incluindo C, C++, Fortran, e outras (LATTNER; ADVE, 2002). Mesmo que o NaCl possuísse foco na portabilidade de códigos desenvolvidos em C e C++, ainda possibilitava extensão do suporte da tecnologia para outras linguagens.

Apesar do desempenho, NaCl ainda possuía limitações significativas. O código compilado produzido estava atrelado a uma plataforma alvo e não funcionava em outros navegadores que não o Google Chrome. Havia também uma gama de problemas relacionados à alocação dinâmica de memória, e os códigos estavam restritos por rígidos padrões de desenvolvimento.

Outra tecnologia proposta para melhorar o desempenho e portabilidade de *software* para navegadores foi o projeto ASM.js. Trata-se uma linguagem de programação baseada em *JavaScript*, criada pelo desenvolvedor Mozilla Alon Zakai em 2013, projetada para fornecer um desempenho de computação mais elevada em navegadores, permitindo que código C ou C++ seja executado nativamente, sem a necessidade de *plugins* ou aplicativos adicionais. A criação do ASM.js surge em um contexto de crescente importância dos navegadores como plataforma para aplicativos e jogos, bem como uma demanda crescente por aplicativos mais avançados e interativos. A Mozilla, em particular, notou a oportunidade de ampliar seu ecossistema de desenvolvimento de aplicativos e jogos, e investiu em tecnologias como o ASM.js para atender a essa demanda (NOAH et al., 2017).

ASM.js permitiu que a Mozilla ampliasse sua base de desenvolvedores e aumentasse sua receita com a venda de ferramentas de desenvolvimento e suporte técnico. Além disso, a linguagem também atraiu outras empresas e desenvolvedores para o ecossistema Mozilla, aumentando a competição e inovando no campo de aplicativos e jogos para navegadores.

NaCl teve um papel pioneiro ao abrir portas para natividade de código para navegadores, e AMS.js apresentou as primeiras soluções para compatibilidade estendida, mas a crescente demanda por desenvolvimento flexível e portável, com mínimo retrabalho. As condições histórias montavam um ambiente propício para o surgimento orgânico do *WebAssembly*.

2.2 A tecnologia WebAssembly

Segundo Lehmann, Kinder e Pradel (2020), *WebAssembly* é um formato de instrução binário para uma máquina virtual com arquitetura orientada em pilha. Como apontado por Haas et al. (2017), a tecologia é uma arquitetura de instrução virtual ("*virtual instruction set architecture*", ou ISA virtual). O objetivo desta seção é detalhar como funcionam suas estruturas básicas.

2.2.1 Arquiteturas de virtualização

Uma introdução às arquiteturas de máquinas virtuais será elaborada para poderem ser desenvolvidos os tópicos base de WebAssembly. Posteriormente, estes conceitos serão abordados novamente, a fim de definir os conceitos de contêineres.

Virtualização é um processo pelo qual se cria uma representação virtual de um recurso físico, como um computador, sistema operacional, dispositivo de armazenamento ou rede. Isso permite que várias instâncias virtuais de um recurso sejam criadas e gerenciadas a partir de um único recurso físico, o que pode melhorar a eficiência, flexibilidade e escalabilidade das operações de TI (Tecnologia da Informação), de modo que seja mantida independência e isolamento entre tais instâncias. A virtualização é amplamente utilizada em nuvens públicas e privadas, data centers e ambientes de desenvolvimento de *software* para aumentar a eficiência, flexibilidade e escalabilidade. A capacidade de criação de camadas de abstração permite que máquinas sejam virtualizadas e que ferramentas de desenvolvimento possuam compatibilidade abrangente, flexível e de alta portabilidade (SHI et al., 2008).

A trajetória do desenvolvimento da linguagem *Java* é um exemplo de virtualização aplicada ao desenvolvimento de linguagens de ferramentas. Diferente de linguagens que produzem *bytecode* altamente dependentes de arquitetura, como C e C++, o *bytecode* produzido por um programa em *Java* é interpretado por uma máquina virtual especializada, a *Java Virtual Machine* (JVM), que traduz, em tempo de execução, a linguagem de máquina compilada para a linguagem de máquina da arquitetura do sistema. Com o desenvolvimento da tecnologia, estratégias de otimização foram propostas e a implementação de sistemas *Just-In-Time* (JIT) permitiram com que o interpretador de uma JVM identifique possíveis seções de código reutilizáveis, uma vez que o sistema JIT permite que o código seja

compilado já em tempo de execução. A otimização ocorre quando o sistema analisa o código-fonte, e o divide em blocos de código, e então, são selecionados os blocos executados com maior frequência. Em seguida, estes são compilados para código de máquina nativo para a arquitetura específica do sistema, aumentando significativamente a velocidade de execução, em comparação com os blocos de código que serão executados como código interpretado. Além disso, o sistema JIT consegue identificar trechos de código não utilizado, portanto, descartável (OLIVEIRA; SILVA, 2013).

Na técnica de virtualização em pilha, na qual se baseia a JVM, há partilha de um mesmo núcleo entre máquinas virtuais, mas há uma estratégia de isolamento de recursos individuais, como memória e CPU. Tal abordagem utiliza de uma pilha central para organizar instruções de utilização de recursos. O formato de instrução WASM também é descrito como um formato de instruções para máquinas virtuais baseadas em pilha.

2.2.2 Módulos

Módulos WASM são representações estáticas de binários compilados para código de máquina executável em navegadores. Contém definições de funções, tabelas, conjuntos de variáveis globais e de memória. As definições de um módulo podem ser exportadas e importadas, como é descrito por Haas et al. (2017). Através de definições importadas, chamadas *imports*, é possível definir os parâmetros necessários para declarar uma instância dinâmica de um módulo. Já as definições exportadas, ou *exports*, determinam quais conteúdos estão disponíveis para acesso em uma instância (MDN, 2022).

O código interno a um módulo é organizado em funções, que podem receber e retornar sequências de valores e parâmetros.

2.2.3 Memória

Módulos são capazes de alocar um *array* de *bytes*, utilizados como um espaço de memória linear. A manipulação deste *array* se dá por uma série de instruções reservadas, que permitem acesso, escrita, e até expansão do valor alocado no momento em que um módulo é instanciado (ROSSBERG, 2022).

Essa memória é geralmente alocada de forma segura, isolada e limitada para cada instância de uma aplicação, de forma que o sistema possua estabilidade e robustez. A arquitetura de memória WASM é aliada com a flexibilidade de uma linguagem dinâmica, como nas operações de leitura e escrita de espaços de memória WASM com JavaScript.

Em desenvolvimento de software é comum que os desenvolvedores tenham que decidir sobre o comprometimento de recursos e funcionalidades, em diversos níveis, para garantir determinados requisitos de operação. Isso é refletido em aspectos como velocidade de execução, segurança e gerenciamento de memória. Linguagens compiladas de baixo

nível têm velocidades notáveis de execução, mas sacrificam verificações do uso de dados na memória. Por outro lado, linguagens interpretadas que possuem ferramentas próprias de coleta de lixo de memória oferecem garantias e tratamento de dados em tempo de execução, e permitem operações dinâmicas que podem acelerar desenvolvimento, porém, neste caso, sacrificam desempenho de execução em tempo real, assim como a previsibilidade do comportamento ao realizar operações de alocação de memória em máquinas que possuem recursos computacionais limitados.(SLETTEN, 2022)

A comunicação realizada pela máquina virtual que interpreta o código de máquina WASM ocorre quando há interface com o DOM (*Document Object Model*). O DOM é uma API para documentos HTML ou XML (*Extensible Markup Language*) que permite *software* acesse e manipule o conteúdo, estrutura e estilo de um documento eletrônico. Este documento é convertido em uma estrutura de objetos — chamada de árvore de objetos do DOM, onde cada elemento HTML ou XML é representado por um objeto e seus atributos e conteúdos são propriedades desse objeto — que pode ser manipulada por *scripts*, como a linguagem *JavaScript* (HILBIG; LEHMANN; PRADEL, 2021). A interface entre WASM o DOM ocorre através de uma API específica chamada *JavaScript-WebAssembly API*. Essa API fornece uma série de funções que permitem que o código *JavaScript* acesse e manipule o código WASM, bem como forneça acesso às funcionalidades do navegador, como o DOM e os recursos do dispositivo. Além disso, o código WASM também pode acessar e manipular o DOM através de *JavaScript*. Isso é feito através de uma técnica chamada "*JS-WASM interop*", onde o código WASM chama funções *JavaScript* que, por sua vez, acessam e manipulam o DOM.

ArrayBuffer é uma classe do *JavaScript* que representa uma memória temporária de bytes de tamanho fixo. Ele é usado para armazenar dados binários, como imagens, vídeos e arquivos de áudio. Ele pode ser compartilhado com uma instância de *WebAssembly*, permitindo que a aplicação WASM acesse e manipule esses dados binários.

Uma vez compartilhado, o *ArrayBuffer* é convertido em uma memória linear no *WebAssembly*, que pode ser acessada através de endereços de memória absolutos. Isso permite que a aplicação WASM execute operações de baixo nível, como acesso aos bytes individuais do *ArrayBuffer*, o que pode ser útil para tarefas como processamento de imagens ou codificação de áudio.

2.2.4 Tabela

O mecanismo que habilita troca entre a memória linear WASM e o *JavaScript* é proveniente da implementação do sistema de tabelas. A instrução de tabela permite a definição um tipo de tabela WASM, e consiste na combinação de dois parâmetros: O tipo de elemento e o número máximo de elementos em uma tabela. Os tipos de elemento podem ser tanto ponteiros para funções (os quais são resultados de uma política de abstração de

endereçamento de memória, que evita a exposição da memória linear de forma insegura) como referências para objetos *JavaScript*. O segundo parâmetro, o número máximo de elementos, é opcional, e serve para alocar uma quantidade de memória para a tabela — entretanto, este valor pode ser alterado dinamicamente, em tempo de execução.

2.3 OpenGL e WebGL

OpenGL é uma tecnologia proposta para realizar interface entre *software* e *hardwre* gráfico. Por isso, é correto classificar a biblioteca como uma API gráfica. Através dela, é possível construir modelos em espaço bidimensional e tridimensional utilizando de primitivas geométricas: um conjunto de informações que descrevem vértices, arestas e coordenadas. A biblioteca conta com uma variedade de tipos — que possuem um tipo diretamente correspondente na linguagem C — utilizados para descrever os polígonos que compõe uma cena em espaço virtual Woo et al. (1999). A API é largamente utilizada no desenvolvimento de *softwares* de CAD (*Computer-Aided Design*) e CAM (*Computer-Aided Manufacturing*), bem como na construção de ferramentas para indústria de jogos(CHEN; CHENG, 2005).

2.3.1 GLEW

GLEW, ou "*OpenGL Extension Wrangler Library*", é uma biblioteca de *software open-source* que fornece acesso a extensões de *hardware* para a API de gráficos 3D *OpenGL*. Ele é projetado para lidar com a complexidade de gerenciar as diferentes extensões disponíveis em diferentes dispositivos e sistemas operacionais, proporcionando uma interface simplificada para os desenvolvedores de aplicativos. A tecnologia foi elaborada para resolver o problema de acessar extensões de hardware em diferentes sistemas operacionais e dispositivos. Cada dispositivo e sistema operacional tem suporte para um conjunto diferente de extensões, e gerenciar essas diferenças manualmente pode ser complicado e trabalhoso. A biblioteca resolve esses problemas fornecendo uma interface simples e consistente para acessar extensões, independentemente do dispositivo ou sistema operacional.

É escrita em C e é compatível com a maioria das plataformas, incluindo Windows, Linux e Mac OS.

2.3.2 GLFW

GLFW é uma biblioteca de *software open-source* que fornece uma interface para gerenciar janelas e entradas para aplicativos de computação gráfica. Seu projeto foi elaborado como alternativa simples e independente de plataforma para bibliotecas de janelas como GLUT (*The OpenGL Utility Toolkit*) e SDL (*Simple DirectMedia Layer*). A

biblioteca é desenvolvida em C e é compatível com a maioria das plataformas, incluindo Windows, Linux e Mac OS X.

A GLFW fornece uma interface para criar janelas e gerenciar entradas como teclado, mouse e *joystick*. Também fornece suporte para contextos de renderização, como *OpenGL* e *OpenGL ES* (API que surge como especialização do *OpenGL* para sistemas embarcados, como dispositivos móveis), permitindo que os desenvolvedores de aplicativos criem janelas e gerenciem entradas sem se preocupar com detalhes de plataforma específicos.

Além disso, a GLFW oferece recursos avançados, como suporte para monitorização de tela, gerenciamento de janela e suporte para *drag-and-drop*, tornando-o uma opção robusta e flexível para desenvolvimento de aplicativos gráficos.

2.3.3 GLM

GLM, ou "*OpenGL Mathematics*", é uma biblioteca de *software open-source* para matemática computacional em aplicativos de computação gráfica. É projetada para fornecer uma interface consistente e fácil de usar para operações matemáticas comuns, como cálculos de transformação e interpolação, além de suportar a API de computação gráfica *OpenGL*. É escrito em C++ e é compatível com a maioria das plataformas, incluindo Windows, Linux e Mac OS X.

A biblioteca oferece uma variedade de tipos de dados matemáticos, como vetores, matrizes e quatérnions, bem como operações para esses tipos de dados, e inclui funções para cálculos de transformação, como rotação, translação e escala, e funções para interpolação, como *spline* e números de Euler. Além disso, GLM tem uma compatibilidade direta com o GLSL, ou *GL Shading Language*, linguem utilizada para descrever *shaders* e materiais, e definem como o *OpenGL* deve manipular vértices, faces, normais e outros atributos de geometria da computação gráfica. As ferramentas fornecidas por GLM são úteis para desenvolvimento de renderizadores, manipuladores e motores 3D.

2.3.4 WebGL

WebGL (*Web Graphics Library*) é uma especificação de API *JavaScript*, baseado em *OpenGL*, que é utilizada para desenvolvimento de aplicações de computação gráfica para navegadores, em *JavaScript*. *WebGL* usa o contexto gráfico da *canvas HTML5* (uma área de desenho em uma página web utilizada para desenhar gráficos, imagens e animações por meio de *scripts*) para desenhar gráficos em tela. Tal como *OpenGL*, a *WebGL* fornece acesso ao hardware de gráficos do dispositivo, permitindo aos desenvolvedores desenhar primitivas gráficas, como triângulos, quadriláteros e linhas, bem como carregar e manipular imagens e texturas. Também permite a utilização de recursos avançados, como iluminação, sombras e sistemas de partículas (MATSUDA; LEA, 2013).

2.4 Compilação cruzada

Cross-compiling, ou compilação cruzada, é o processo que habilita compilar um *software* para uma plataforma diferente daquela em que um determinado compilador está sendo executado — é a capacidade de compilar um programa em um sistema (conhecido como sistema *host*) e gerar código executável para outro sistema (conhecido como sistema de *target*). A tecnologia habilita a produção de código independente de plataforma ou ambiente de desenvolvimento e concede flexibilidade para desenvolvedor de forma a estender a capacidade de compatibilidade de código para uma gama de sistemas operacionais e arquiteturas de processadores. Outro fator que argumenta a favor da compilação cruzada é a disponibilidade de recursos físicos da plataforma *target*: no caso do desenvolvimento *mobile*, o desenvolvedor não precisa estar em posse de todos os dispositivos ou arquiteturas para quais desenvolverá um *firmware*. Não obstante, caso o sistema *target* disponha de pouco poder computacional, a compilação cruzada pode acelerar o processo de compilação quando é executado em um dispositivo *host* de desempenho superior (GAY, 2018).

Para realizar a compilação cruzada é necessário ter o compilador, bibliotecas e ferramentas de desenvolvimento para a plataforma de destino, além de configurar corretamente as variáveis de ambiente e *flags* de compilação. Por mais que a compilação cruzada resolva problemas basais de compatibilidade, o desenvolver deve estar atento a requisitos específicos de *software* e disponibilidade de recursos particulares de cada sistema operacional ou arquitetura alvo. Ferramentas como o CMake permitem que cenários sejam previamente contemplados, e que o processo de compilação siga estratégias apropriadas para cada cenário automatizadamente.

2.4.1 Gnu Compiler Collection

A origem do GCC, ou "*Gnu Compiler Collection*", se deve ao movimento do software livre, liderado por Richard Stallman e a Fundação GNU, que buscava criar um sistema operacional livre e de código aberto baseado em Unix. A necessidade de um compilador livre e de código aberto para o projeto do sistema operacional GNU foi o ponto de partida para o desenvolvimento do GCC.

A partir da década de 1980, o GCC foi sendo desenvolvido e aperfeiçoado, com o objetivo de suportar várias arquiteturas de processadores e sistemas operacionais. Esse processo foi guiado pelas necessidades econômicas e políticas do movimento do software livre, que buscava criar uma alternativa livre e de código aberto aos compiladores proprietários dominantes da época.

Ao longo dos anos, o GCC tem sido amplamente utilizado em vários projetos de software livre, incluindo sistemas operacionais, bibliotecas e aplicativos. Sua popularidade cresceu a medida que o movimento do software livre se tornou cada vez mais relevante e

as necessidades econômicas e políticas do setor evoluíram (LERNER; TIROLE, 2001).

2.4.2 Emscripten

A portabilidade de código para a *Web* é um objetivo que antecede o *WebAssembly*. Diversos métodos para migrar aplicações para navegadores foram propostos, e um desses descreve a compilação de código-fonte diretamente para *JavaScript*. Há limitações consideráveis para tal método, que incluem problemas de incompatibilidade semântica entre as linguagens, alocação de memória e segurança (EMSCRIPTEN, 2015). O compilador *Emscripten* foi projetado em 2011 para converter códigos em C e C++ para JavaScript e ASM.js era um dos códigos produzidos. Foi só depois do anúncio do *WebAssembly* (em 2015) e início da integração da arquitetura de virtualização WASM em navegadores (em 2017) que o *Emscripten* tomou frente como um dos compiladores de *bytecode* WASM mais utilizados (ZAKAI, 2018).

O *Emscripten* é baseado no LLVM, uma ferramenta de compilação *open-source*, havendo sido projetado para fornecer uma forma de compilar código C/C++ para *JavaScript* de maneira fácil e eficiente. Com o *Emscripten*, é possível compilar aplicativos e jogos existentes escritos em C/C++ para funcionar em navegadores *Web*, sem a necessidade de reescrevê-los completamente em *JavaScript*.

2.4.3 CMake

CMake foi desenvolvido pela Kitware, Inc. como uma alternativa de plataforma cruzada para ferramentas de construção tradicionais, como o *make*. Ele foi projetado para fornecer uma maneira de construir software em diferentes sistemas operacionais e arquiteturas de processadores, sem a necessidade de modificar manualmente *scripts* de construção.

Sua criação foi impulsionada pelas necessidades econômicas e políticas do setor da tecnologia, especificamente a crescente necessidade de software que possa ser facilmente construído em diferentes plataformas. Com a proliferação de diferentes sistemas operacionais e arquiteturas de processadores, havia uma necessidade crescente de ferramentas que possibilitassem a construção de software em diferentes plataformas sem a necessidade de modificar manualmente *scripts* de construção. A tecnologia é baseada em *scripts* de configuração em formato de texto simples, que descrevem as dependências e configurações de construção de um projeto. Ele gera *scripts* de construção específicos para diferentes sistemas operacionais e compiladores, permitindo que o *software* seja construído em diferentes plataformas sem a necessidade de modificar manualmente *scripts* de construção (BAST; REMIGIO, 2018).

Ao longo dos anos, o CMake tem sido amplamente utilizado em projetos de *software*

open-source e comerciais, contribuindo para a popularidade crescente de ferramentas de construção cruzadas. Sua popularidade cresceu à medida que as necessidades econômicas e políticas do setor evoluíram, com o aumento da necessidade de *software* que possa ser facilmente construído em diferentes plataformas.

2.5 Estratégia de virtualização de recursos em contêineres

A virtualização através de contêineres surgiu como uma alternativa às técnicas de virtualização tradicionais, como a virtualização de máquinas completas. Ela permite a criação de ambientes isolados, conhecidos como contêineres, dentro de um sistema operacional existente, permitindo que diferentes aplicações e bibliotecas sejam executadas de forma independente e isolada.

A criação da virtualização através de contêineres foi impulsionada pelas necessidades econômicas e políticas do setor da tecnologia, especificamente a crescente necessidade de escalabilidade e eficiência na utilização de recursos. Com a crescente popularidade de aplicações baseadas na nuvem e a necessidade de lidar com grandes volumes de tráfego de dados, havia uma necessidade cada vez maior de técnicas que permitissem o uso eficiente de recursos em diferentes níveis de escalabilidade.

Como estabelecido previamente, a virtualização isola recursos enquanto compartilha a plataforma. No caso dos contêineres, cada instância é isolada das demais e do sistema operacional hospedeiro, e compartilham o *kernel* do sistema operacional — fato que explica a maior eficiência no uso de recursos. Isso permite que vários contêineres possam ser executados em um único sistema, aumentando a escalabilidade e a eficiência.

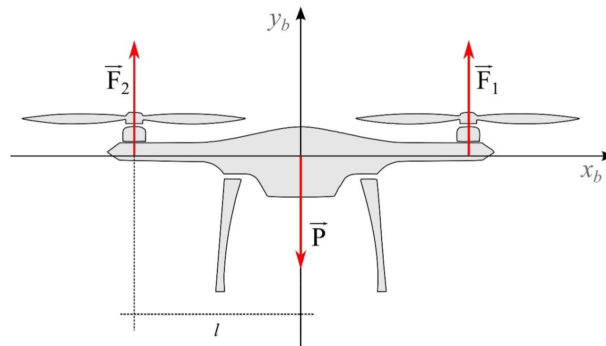
2.5.1 Docker e docker-compose

Docker é uma plataforma de virtualização de contêineres que permite a criação, implantação e execução de aplicações em contêineres isolados dentro de um sistema operacional existente. Ele foi criado pela Docker, Inc. com o objetivo de fornecer uma forma fácil e eficiente de gerenciar aplicações em contêineres. A criação do Docker foi impulsionada pelas necessidades econômicas e políticas do setor da tecnologia, especificamente a crescente popularidade de aplicações baseadas na nuvem e a necessidade de lidar com grandes volumes de tráfego de dados. Com a crescente popularidade de aplicações baseadas na nuvem, havia uma necessidade crescente de ferramentas que permitissem o gerenciamento eficiente de aplicações em contêineres. *Docker-compose* é uma ferramenta adicional para gerenciamento de contêineres no *Docker*, que permite a criação e gerenciamento de aplicações compostas por vários contêineres. Ele permite a configuração de vários contêineres como uma única aplicação e suporta o gerenciamento de dependências entre os contêineres (ANDERSON, 2015).

2.6 Modelagem dinâmica de um Drone

O sistema físico estudado é um modelo simplificado de um drone, que navega em um espaço de duas dimensões. A modelagem será restrita à dinâmica do sistema físico em um espaço 2D. Através de um modelo físico em espaço de estados, é possível simular as forças que atuam sobre o corpo de um bicóptero. A figura 1 ilustra como o drone foi modelado. Para um sistema de coordenadas F_b, que têm como referência o centro de massa do modelo: A força peso $\overrightarrow{P}$ atua diretamente sobre o centro de massa, enquanto as forças $\overrightarrow{F_1}$ e $\overrightarrow{F_2}$ atuam à uma distância l.

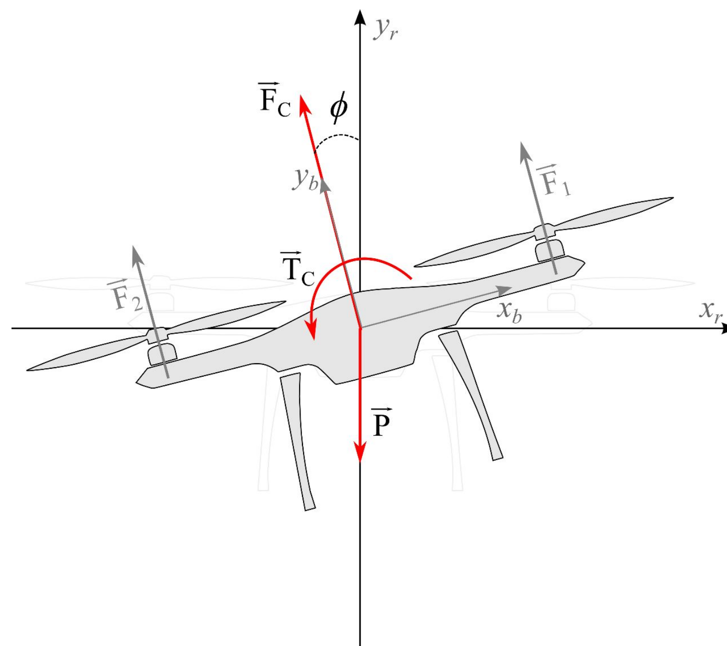
Figura 1 – Modelo de forças aplicadas no drone.



Fonte: Moura (2022).

O movimento aplicado pelos rotores produz torque que atua sobre a orientação do veículo. A Figura 2 ilustra as forças físicas resultantes.

Figura 2 – Forças de torque resultante.



Fonte: Moura (2022).

A cinemática do sistema é modelada pelas seguintes equações:

$$\dot{w} = \frac{1}{\tau} \cdot (-w + \bar{w}) \tag{2.1}$$

onde $w = \begin{bmatrix} w_1 & w_2 \end{bmatrix}^T \in \mathbb{R}^2$ é a velocidade de rotação dos rotores.

$$\dot{r} = v \tag{2.2}$$

onde $r = \begin{bmatrix} x_r & y_r \end{bmatrix}^T \in \mathbb{R}^2$ é a posição do drone.

$$\dot{v} = \frac{1}{m} \cdot (D^{R/B}(\phi) \cdot F_C + P) \tag{2.3}$$

onde $v = \begin{bmatrix} v_x & v_y \end{bmatrix}^T \in \mathbb{R}^2$ é a velocidade linear do drone.

$$\dot{\phi} = \omega \tag{2.4}$$

onde $\phi \in \mathbb{R}$ é a atitude.

$$\dot{\omega} = \frac{1}{I_Z} \cdot T_C \tag{2.5}$$

onde $\omega \in \mathbb{R}$ é a velocidade angular de rotação do conjunto rotor/hélice.

$$F_i = k_f \cdot w_i^2, i \tag{2.6}$$

onde $F_i \in \mathbb{R}$ é a força aplicada por cada um dos rotores, e i assume valores 1 e 2.

$$F_c = \begin{bmatrix} 0 \\ F_1 + F_2 \end{bmatrix} \tag{2.7}$$

onde $F_c \in \mathbb{R}$ é a força de controle.

$$T_c = l \cdot (F_1 + F_2) \tag{2.8}$$

onde $T_c \in \mathbb{R}$ é o torque de controle.

$$D^{R/B}(\phi) = \begin{bmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{bmatrix} \tag{2.9}$$

onde $D^{R/B}(\phi) \in \mathbb{R}^{2\times 2}$ é matriz de rotação.

Por fim, as seguinte relações devem ser definidas:

$$P = m \cdot g \tag{2.10}$$

onde $P \in \mathbb{R}$ é a força peso.

Além das equações definidas, algumas constantes devem ser consideradas:

- l é a distância entre um rotor e o centro de massa do drone;
- $_f$ é a — constante de proporcionalidade de força de um rotor;
- τ é a — constante de proporcionalidade de força;
- I_Z é o momento de inércia;
- g é a constante de aceleração gravitacional.

A simulação permitirá o controle da posição do drone através dos rotores. A atuação da velocidade de rotação do rotor, definidas como $\bar{W}_1$ e $\bar{W}_2$, é possível calcular as forças resultantes aplicadas em cada um dos rotores F_1 e F_2. Através da equação (2.7), a força F_c — que controla o movimento linear do drone — é obtida; já com equação (2.8), a força que controla o movimento angular do drone é obtida (MOURA, 2022).

A solução de controle deste sistema deve ser capaz manter o drone em uma posição desejada (posição de comando) e orientação desejada (ângulo de comando). Para cumprir tal tarefa, o dois tipos de controle são implementados: controle de posição e controle de atitude.

O controle de posição é feito através de um controlador proporcional-derivativo (PD) que utiliza as informações de posição do drone (posição atual) e posição desejada (posição de comando) para calcular a força de controle necessária para alcançar a posição desejada. (MOURA, 2022)

A ação de controle de posição é descrita pelas seguintes equações:

$$C_r x = k_p \cdot e_{r_x} + k_c \cdot e_{v_x} \tag{2.11}$$

onde $C_r x$ é a ação de controle de posição para o eixo x, kp_p e kd_p são os ganhos do controlador PD, e_{r_x} é o erro de posição o eixo x e e_{v_x} é o erro de velocidade o eixo x.

$$C_r y = k_p \cdot e_{r_y} + k_c \cdot e_{v_y} - F_g \tag{2.12}$$

onde $C_r y$ é a ação de controle de posição para o eixo y, kp_p e kd_p são os ganhos do controlador PD, e_{r_y} é o erro de posição o eixo y e e_{v_y} é o erro de velocidade o eixo y, F_g é a força exercida pela gravidade.

A ação de controle de atitude é determinada por um controlador proporcional que utiliza a informação de ângulo do drone (ângulo atual) e ângulo desejado (ângulo de comando) para calcular o torque de controle necessário.

A ação de controle de atitude é descrita pelas seguintes equações:

$$C_\phi = -\tan^{-1}(C_r x, C_r y) \tag{2.13}$$

onde C_ϕ é a ação de controle de atitude.

O modelo matemático para resolução das equações de controle será Runge-Kutta. Runge-Kutta é um método numérico utilizado para resolver equações diferenciais ordinárias (EDOs) em sistemas dinâmicos. Consiste em uma sequência de iterações, cada uma das quais aproxima a solução da EDO, portanto, pode ser integrado em uma simulação computacional.

No contexto de controle de uma simulação de um sistema físico, o método Runge-Kutta pode ser utilizado para modelar dinamicamente a posição, velocidade e orientação de um drone, a fim de calcular a ação de controle necessária para alcançar um determinado objetivo de posição ou orientação. Isso é feito resolvendo as equações de movimento do drone, que descrevem como a aceleração afeta a posição e velocidade, e como forças de torques afetam a orientação do drone.

Existem vários tipos diferentes de métodos Runge-Kutta, cada um com suas próprias características e precisões. Alguns exemplos incluem:

- Método de Euler: é o método Runge-Kutta mais simples e um dos mais antigos. Ele divide o intervalo de tempo em passos pequenos e usa a derivada no ponto inicial para aproximar a solução na próxima iteração.
- Método de Runge-Kutta de 2ª ordem (RK2): também conhecido como método de Heun. Ele usa uma aproximação da derivada no meio do intervalo de tempo para melhorar a precisão em relação ao método de Euler.
- Método de Runge-Kutta de 4ª ordem (RK4): é um dos métodos Runge-Kutta mais populares e precisos. Ele usa quatro aproximações da derivada em diferentes pontos do intervalo de tempo para calcular a solução na próxima iteração.

$$k_1 = h \cdot f(t_n, y_n) \tag{2.14}$$

$$k_2 = h \cdot f(t_n + \frac{h}{2}, y_n + \frac{k_1}{2}) \tag{2.15}$$

$$k_3 = h \cdot f(t_n + \frac{h}{2}, y_n + \frac{k_2}{2}) \tag{2.16}$$

$$k_4 = h \cdot f(t_n + h, y_n + k_3) \tag{2.17}$$

$$y(t_f) = y_n + \frac{1}{6} \cdot (k_1 + 2 \cdot k_2 + 2 \cdot k_3 + k_4) \tag{2.18}$$

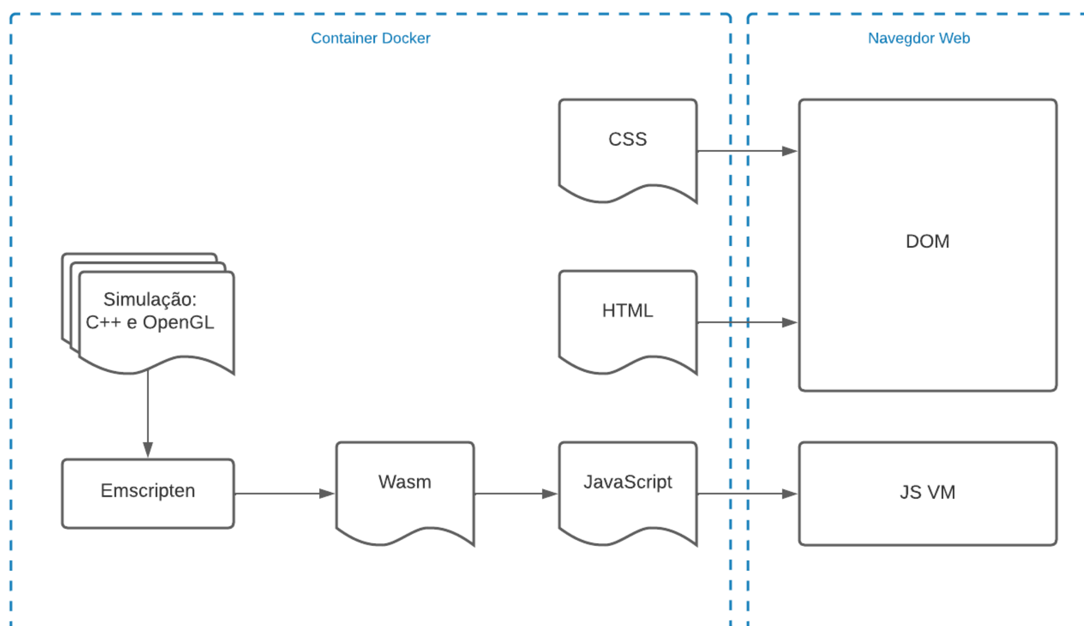
Nestas equações, t_n é o tempo atual, y_n é o valor da solução atual, h é o passo de tempo, $f(t, y)$ é a função diferencial que se deseja resolver, e k_1, k_2, k_3 e k_4 são denominadas como constantes de Runge-Kutta.

3 Metodologia

Este capítulo discorre sobre o processo de desenvolvimento, justifica as decisões técnicas de implementação e esclarece a conexão entre as tecnologias introduzidas no capítulo anterior. Serão destacados os trechos mais relevantes de código para compreensão do sistema, como o sistema de computação gráfica, simulação de um drone e produção de um ambiente automatizado para gerenciamento de desenvolvimento e gerenciamento de dependências. Os trechos relevantes de código muito extensos serão anexados e referenciados onde for relevante.

A Figura 3 abstrai a arquitetura do sistema que será descrito neste capítulo. Assim, a partir do que está apresentado nela, podemos entender o processo de interação e de ações para conversão da aplicação C++/*OpenGL* para uma aplicação *Web*, onde, a simulação de um sistema dinâmico, desenvolvida em C++ e com a API *OpenGL*, é compilada para código WASM através da *toolchain Emscripten*. Então, o código produzido é acessado por um *script JavaScript*, interpretado pelo sistema de virtualização do navegador, e com a combinação das tecnologias CSS e HTML, é carregado neste navegador. É importante notar que todo o código produzido está encapsulado em um contexto de um contêiner *Docker*.

Figura 3 – Arquitetura de alto nível de compilação e execução da simulação.

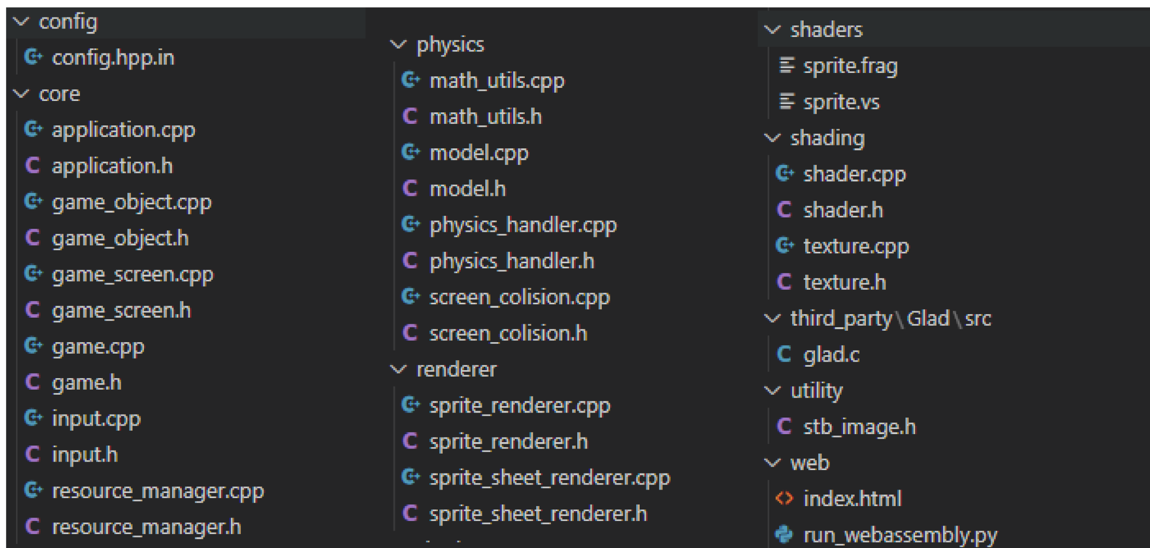


Fonte: Autoria própria (2022).

3.1 Simulação C++ e OpenGL

O código escrito em C++ foi repartido com a intenção de tornar a leitura da arquitetura mais simples, e o desenvolvimento modular. A Figura 4 ilustra a árvore de arquivos do programa.

Figura 4 – Diretórios com a totalidade do código-fonte escrito para este trabalho.



Fonte: Autoria própria (2023).

A Figura 5 é o arquivo *main.cpp*, e é o arquivo principal que coordena a execução do código. Todas as camadas do sistema estão abstraídas, então o papel deste arquivo é apenas declarar uma instância da classe *application* com os parâmetros de tamanho de janela, e executar o método que dá início a simulação.

Figura 5 – Código main.cpp.

```
#include "application.h"

int main(int argc, char *argv[])
{
    Application *app = new Application(800U, 600U);

    app->Run();
}
```

Fonte: Autoria própria (2023).

A classe implementação da *Application* encontra-se no Anexo — E para consulta. Esta tem como papel a definição do laço de execução principal do programa, assim como a inicialização e configuração de todos os atributos gráficos, como janela, controle de contexto de execução, atribuição de funções de *callback* para as teclas de teclado designadas para controle do programa, controle do tempo de execução da simulação e gerenciamento de

recurso de memória (quais instâncias deve iniciar e destruir, com o auxílio de outra classe chamada *ResourceManager*).

O código na pasta *core*, ver Figura 4, é responsável pela execução em mais alto nível do sistema. Enquanto a classe *Application* conduz o laço de execução, as demais classes deste diretório implementam funcionalidades mais especializadas. A classe *GameObject* descreve um objeto que possui métodos e variáveis membro de um ator em um jogo, e para esta simulação, este ator é o drone. Os métodos descrevem como o objeto se movimenta, seus atributos de textura, tamanho, posição na tela, *shader* utilizado e modelo físico utilizado para modelar seu movimento. A classe *GameScreen* implementa os métodos do menu da simulação. A classe *Game* implementa a interação de objetos pertencentes a classe *GameObject* e ao modelo de simulação. Também implementa a máquina de estados que define em qual estado de execução a simulação está: menu principal, controle manual e controle automático. As classes *Input* e *ResourceManager* implementam a interface para implementação dos métodos de entrada do usuário (teclado e mouse) e o gerenciamento de texturas, respectivamente.

O código nos diretórios *shaderers* e *shading* implementam os materiais utilizados para renderizar os polígonos dos objetos (planos 2D desenhados utilizando formas primitivas em *OpenGL*) em cena.

3.1.1 Sistema de render de sprites

A escolha tecnológica da API *OpenGL* foi deliberada, e a intenção era explorar a capacidade de desempenho de *software* de computação gráfica escrito com uma API de baixo nível. Apesar de eficiente, as ferramentas oferecidas pela *OpenGL* são menos flexíveis do que outras APIs, como a *Vulkan*. Assim, desenvolver sistemas de renderizador 2D requer algumas concessões de projeto.

OpenGL lida com polígonos, vértices e coordenadas geométricas em espaço 3D. Portanto, um renderizador 2D nada mais é do que um renderizador 3D com uma câmera estática, com projeção ortogonal.

Um dos primeiros passos da classe que implementa o processamento de polígonos de *sprites* é criar e configurar um *vertex buffer object* (VBO) para um polígono 2D. Um VBO é um tipo de memória temporária de memória que armazena dados de vértice, como posições, normais e coordenadas de textura; permite que esses dados sejam enviados para a *GPU* eficientemente, ao invés de serem transferidos a cada quadro, economizando recursos do sistema. Um VBO pode ser criado, preenchido com dados e vinculado a um *pipeline* de renderização para que os dados possam ser utilizados para desenhar primitivas na tela.

Em seguida, um *vertex array object (VAO)* é definido, e nele são armazenados os atributos e dados de configuração do VBO. O trecho de código da Figura 6 documenta

como os dados de vértices são processados na aplicação.

Figura 6 – Trecho de código da aplicação gráfica para *sprites* estáticas.

```
void SpriteSheetRenderer::initRenderData()
{
    unsigned int VBO;
    GLfloat vertices[] = {
    /*  pos         tex       */
        0.0f, 0.2f, 0.0f, 0.16667f,
        1.0f, 0.0f, 1.0f, 0.0f,
        0.0f, 0.0f, 0.0f, 0.0f,

        0.0f, 0.2f, 0.0f, 0.16667f,
        1.0f, 0.2f, 1.0f, 0.16667f,
        1.0f, 0.0f, 1.0f, 0.0f
    };

    glGenVertexArrays(1, &this->m_quad_VAO);
    glGenBuffers(1, &VBO);

    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

    glBindVertexArray(this->m_quad_VAO);
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 4 * sizeof(GLfloat), (void*)0);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    glBindVertexArray(0);
}
```

Fonte: Autoria própria (2023).

Como pode ser observado, uma estrutura de *array* é inicada com dados que representam as posições dos vértices e coordenadas de textura da *sprite* 2D. O VBO é vinculado (torna-se ativo) e seus dados passam a refletir a estrutura de dados descrita pelo *array* de vértices.

O trecho de código da Figura 7 mostra a estratégia utilizada para animar *sprites*. Uma vez que a escolha técnica estabeleceu restrições sobre as tecnologias utilizadas, nenhum recurso semelhante à motores de jogos foram utilizados, então abstrações de funções gráficas básicas, como animações de textura, foram desenvolvidas apenas com as ferramentas de *OpenGL*. Neste caso, a taxa de quadro da simulação fazia com que o sistema de renderização iterasse por uma *sprite sheet* (um conjunto de texturas concatenadas em apenas uma textura maior), escolhendo os quadros animados corretos.

Figura 7 – Trecho de código da aplicação gráfica para *sprites* animadas.

```
void SpriteSheetRenderer::UpdateRenderData(GLuint param_frame)
{
    unsigned int VBO;
    GLfloat vertices[] = {
    /*  pos         tex       */
        0.0f, 0.2f, 0.0f, (0.16667f + (GLfloat)(0.16667 * (param_frame - 1.0f))),
        1.0f, 0.0f, 1.0f, (0.0f     + (GLfloat)(0.16667 * (param_frame - 1.0f))),
        0.0f, 0.0f, 0.0f, (0.0f     + (GLfloat)(0.16667 * (param_frame - 1.0f))),
        0.0f, 0.2f, 0.0f, (0.16667f + (GLfloat)(0.16667 * (param_frame - 1.0f))),
        1.0f, 0.2f, 1.0f, (0.16667f + (GLfloat)(0.16667 * (param_frame - 1.0f))),
        1.0f, 0.0f, 1.0f, (0.0f     + (GLfloat)(0.16667 * (param_frame - 1.0f)))
    };
```

Fonte: Autoria própria (2023).

3.1.2 Simulador de um sistema físico com métodos numéricos

Devido à extensão do código da implementação dos métodos matemáticos, os trechos relevantes para compreensão da simulação foram incluídos nos anexos. Figuras com fragmentos acompanham as descrições presentes neste capítulo.

O método da classe responsável por processar o método RK4 recebe alguns argumentos:

- *param_current_time*: um vetor contendo o tempo atual da simulação
- *param_integration_step*: um vetor contendo o tamanho do passo para a integração
- *param_current_state_vector*: um vetor contendo o estado atual do sistema
- *param_current_input_cmd*: um vetor contendo quaisquer comandos de entrada para o sistema
- *param_model*: um ponteiro para uma classe *Model*, que se assume ter um método chamado *StateVector* que calcula a derivada do vetor de estado

O código, então, procede a realizar o algoritmo de Runge-Kutta em várias etapas — etapas que podem ser compreendidas pelas Equações (2.14 - 2.18).

O tempo de simulação, ou seja, o intervalo entre cada passo, é definido pelo *delta time* calculado na classe de aplicação durante o fluxo de controle da taxa de quadros por segundo do sistema. O *delta time* é um conceito comumente usado em jogos para medir o tempo entre dois quadros (*frames*) de animação. Ele é calculado como a diferença de tempo entre o quadro renderizado atual e o quadro anterior.

O algoritmo tem início ao calcular tempo para o primeiro passo intermediário, adicionando metade do passo de integração ao tempo atual. Em seguida, calcula o primeiro vetor de estado intermediário, chamando o método *StateVector* da classe *Model* com o tempo atual, vetor de estado atual e comandos de entrada atuais. As etapas decorrentes seguem o modelo RK4 proposto, realizando operações com o *StateVector* e os coeficientes de Runge-Kutta. Resulta em um vetor de estado final, produto da soma ponderada dos vetores de estado intermediários. A implementação do método encontra-se no Anexo — B.

Este código implementa um controlador de posição Proporcional-Derivativo para o drone, cujo movimento está restrito ao plano vertical local. De início, são estabelecidos os parâmetros de simulação do drone, tais como massa, aceleração da gravidade, comprimento da asa, constante de força, momento de inércia, tau, passo de integração e escala. Em seguida, os parâmetros são utilizados para calcular as forças e torque que atuam no drone, bem como a matriz de rotação e as derivadas das velocidades e posições. Isso é feito para calcular o próximo estado do drone a partir do estado atual e dos comandos de entrada.s

A classe *Model* também tem um método chamado *StateVector*, responsável por calcular a derivada do vetor de estado do sistema, ou seja, as taxas de mudança dos estados do drone. Esse método usa informações sobre os estados atuais do drone, comandos de entrada e parâmetros do drone para calcular as derivadas. A implementação do modelo do drone encontra-se no Anexo — C, enquanto o sistema de controle encontra-se no Anexo — D.

3.1.3 Recursos visuais

Todas as texturas foram criadas no *software Gimp*, e estão inclusas no repositório.

Ao executar o processo de compilação do código C++ para *bytecode*, antes da conversão para WASM, o *CMake* carrega as texturas para o diretório que será servido no navegador.

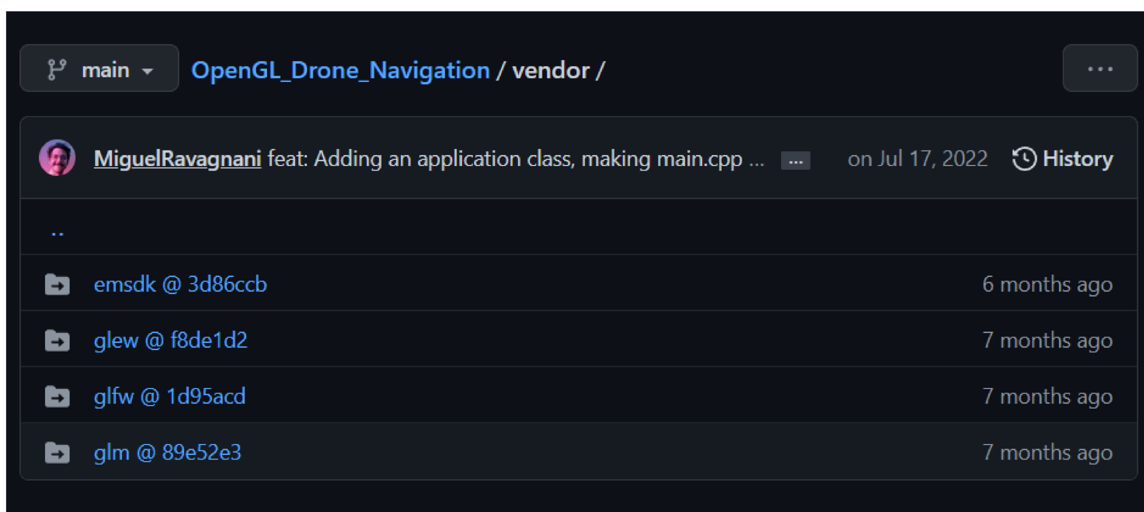
3.2 Configuração para desenvolvimento em WebAssembly

O decorrer do desenvolvimento da aplicação revelou uma necessidade de estrito controle de dependências de todas as tecnologias envolvidas. Uma vez que o programa utilizava três repositórios terceiros (as bibliotecas gráficas GLAD, GLEW e GLFW) como requisitos — configurados como submódulos em um repositório proprietário hospedado no site GitHub — foi preciso definir suas respectivas versões.

O próximo passo foi encontrar compatibilidade entre a versão da *toolchain Emscipten* utilizada, a versão do compilador GCC, a versão da ferramenta *CMake*, e a versão do *Python* utilizado pelo sistema em que estava sendo feito o desenvolvimento. Utilizar um

sistema de controle de versão como GIT possibilitou a organização de certas dependências em forma de submódulos, que automatizam a clonagem de repositórios que possuam dependências de outros repositórios. A Figura 8 ilustra os submódulos mencionados.

Figura 8 – Organização de dependências externas em repositório GIT.



Fonte: Autoria própria (2023).

Porém, o ambiente de desenvolvimento tomava um aspecto inflexível, e casos de incompatibilidade entre bibliotecas nativas do sistema hospedeiro (Linux Mint 20.3) e as versões das dependências intrínsecas de cada uma das bibliotecas impediam o avanço do estudo e uma estratégia de isolamento de recursos teve de ser traçada.

Utilizando Docker, um arquivo *Dockerfile* foi elaborado para atender ao requisito de enclausuramento das diversas dependências em um ambiente de desenvolvimento controlado. A imagem continha instruções que determinavam desde as versões de imagem base do sistema (especificamente, uma imagem Docker Ubuntu), até as instruções de compilação do código-fonte final. O arquivo *Dockerfile* completo, que compreende todas as instruções de instalação automatizadas, está descrito no Anexo — A.

As dependências do projeto foram:

- Sistema operacional Ubuntu, versão 20.04;
- Linguagem de programação Python, versão 3.8;
- *Toolchains* gcc, g++ e gofrtran, de versões gcc-7, g++-7 e gfortran-7, respectivamente;
- CMake, de versão 3.23.2;
- Emscripten SDK, de versão 3.1.12;
- Biblioteca GLEW, versão 2.1.0;

- Biblioteca GLFW, versão 3.2.1;
- Biblioteca GLM, versão 0.9.8.6;

Pacotes que não causaram problemas de incompatibilidade de versionamento não foram listadas.

Uma vez que a imagem *Docker* está endo executada e estabelece o contexto de ambiente restrito, é possível acessá-la através do terminal e interagir dentro desta como se fosse uma instância de máquina virtual. Para facilitar o desenvolvimento, é possível utilizar uma *flag* de execução que define um volume do contêiner, ou seja, um diretório específico que será vinculado a um diretório do sistema anfitrião. Assim, o código-fonte pode ser alterado por um editor de texto externo à imagem *Docker*, como se estivesse no próprio sistema do desenvolvedor. Ao fim do desenvolvimento, o *software* é servido por um script *Ptython* em uma determinada porta, no contexto do contêiner. Para que o conteúdo servido seja acessível na máquina *host*, uma outra *flag* de execução que tem como papel vincular uma porta do sistema convidado ao sistema anfitrião, e permitir que código servido no contexto de uma imagem *Docker* seja acessado de fora (BHAT; BHAT, 2022).

3.3 Conversão entre C++/OpenGL e WebAssembly/WebGL

A conversão de código C++ e *OpenGL* para *WebAssembly* e *WebGL* envolve várias etapas:

- Compilação: O código C++ é compilado para um arquivo binário, geralmente usando um compilador como o GCC ou Clang. Esse arquivo binário contém código de máquina que pode ser executado diretamente pelo processador;
- Conversão para *WebAssembly*: O arquivo binário gerado na etapa anterior é convertido para *WebAssembly*. Neste projeto, a ferramenta responsável pela conversão é a *toolchain Emscripten*. Segundo Zakai (2011) *Emscripten* compila o código e realiza a tradução, as chamadas de função OpenGL são substituídas com chamadas equivalentes em WebGL. Nesta etapa, o *Emscripten* é capaz de executar técnicas de otimização de código e remoção de código morto;
- Adequação das chamadas *OpenGL*: O código *C++/OpenGL* original pode fazer chamadas para funções e recursos específicos do *OpenGL*, que precisam ser adaptadas para o *WebGL*, que é a versão da biblioteca de gráficos 3D para a *Web*. Esta etapa é realizada após a compilação, e por mais que seja comum, não foi necessária no desenvolvimento deste projeto.

3.3.1 Laços de execução Emscripten

A classe de aplicação (encontrada no Anexo — E) utiliza a biblioteca GLFW para criar uma janela de aplicativo, gerenciar o contexto da aplicação e processar eventos de entrada. As funções da API OpenGL são carregadas através da biblioteca GLAD. Quando esse código é compilado com *Emscripten*, a biblioteca GLFW é substituída por sua versão JavaScript equivalente e as chamadas OpenGL são traduzidas para as chamadas WebGL correspondentes. Além disso, a biblioteca de tempo de execução JavaScript fornecida por *Emscripten* é usada para fornecer as chamadas de sistema e as funções de biblioteca necessárias para que o código possa ser executado no navegador. A Figura 9 ilustra a forma com que a ferramenta *Emscripten* é utilizada para identificar o laço de execução principal da aplicação. No corpo de execução do método principal da classe, a função *registered_loop* recebe uma função lamba: função com o papel de alterar, de forma dinâmica, o comportamento de uma função definida de forma anônima. Assim, o código produzido pela compilação WASM precisará apenas executar o laço principal do código.

Figura 9 – Laço de execução Emscripten.

```
#include "application.h"

std::function<void()> registered_loop;
void loop_iteration() {
    registered_loop();
}
```

Fonte: Autoria própria (2023).

Além de atualizar as entradas e saídas das funções da API gráfica, o laço principal executa os passos de simulação.

3.4 Execução

A última etapa do desenvolvimento foi estabelecer uma forma de servir a aplicação na *Web*. Isso foi realizado através de um simples servidor HTTP desenvolvido em *Python*, que serve o diretório com o código WASM compilado no navegador. O programa define a porta do servidor como 8888, e permite que o diretório servido seja acessado pelo endereço “http://localhost:8888”. O servidor continuará funcionando até que seja interrompido pelo usuário.

O comando de execução do servidor HTTP, bem como a definição do diretório que será servido são definidos no arquivo *Dockerfile*, e quando a imagem do contêiner é executada, o servidor inicia a aplicação com o auxílio de um arquivo *docker-compose*, cujo

Figura 10 – Servidor HTTP.

```python
#! /usr/bin/python3
from http.server import HTTPServer, SimpleHTTPRequestHandler
import sys
import webbrowser

PORT = 8888

class CustomHTTPRequestHandler(SimpleHTTPRequestHandler):
    def end_headers(self):
        self.send_header("cross-origin-embedder-policy", "require-corp")
        self.send_header("cross-origin-opener-policy", "same-origin")
        SimpleHTTPRequestHandler.end_headers(self)

with HTTPServer(("", PORT), CustomHTTPRequestHandler) as httpd:
    try:
        webbrowser.open("http://localhost:%s" % PORT)
        print("serving at port", PORT)
        httpd.serve_forever()
    finally:
        sys.exit(0)
```

Fonte: Autoria própria (2023).

papel é orquestrar a execução de múltiplos contêineres *Docker*. Este arquvio permite que sejam definidas as regras para gerenciamento de todos os recusos *Docker*, como volumes de dados, redes e configurações, versão, variáveis de ambiente, comandos de execução e também é capaz de buscar imagens no repositório oficial do *Docker*. Isso facilita a criação de ambientes de desenvolvimento e produção consistentes e escaláveis. Tal controle ocorre a partir do arquivo YAML (acrônico recursivo para *YAML Ain't Markup Language*), um arquivo de marcação de dados. A Figura 11 ilustra a configuração deste arquivo (BHAT; BHAT, 2022).

Figura 11 – Arquivo docker-compose.yml.

```yaml
version: '3.3'

services:
    web:
        build:
            dockerfile: Dockerfile
            context: .
        image: webassembly:latest
        container_name: webassembly

        volumes:
            - ./:/home/transfer

        network_mode: "host"

        restart: always
```

Fonte: Autoria própria (2023).

4 Resultados

Os resultados validaram as propostas do projeto incrementalmente. Na primeira etapa de desenvolvimento, a aplicação funcionava em um ambiente nativo, executando a simulação em *bytecode* produzido pela compilação de código C++ para um sistema operacional Linux.

Na próxima etapa de desenvolvimento, o processo de compilação e gerenciamento de dependências foi encapsulado em um contêiner Docker, e a execução passou a ser realizada de forma automatizada e independente do ambiente de desenvolvimento.

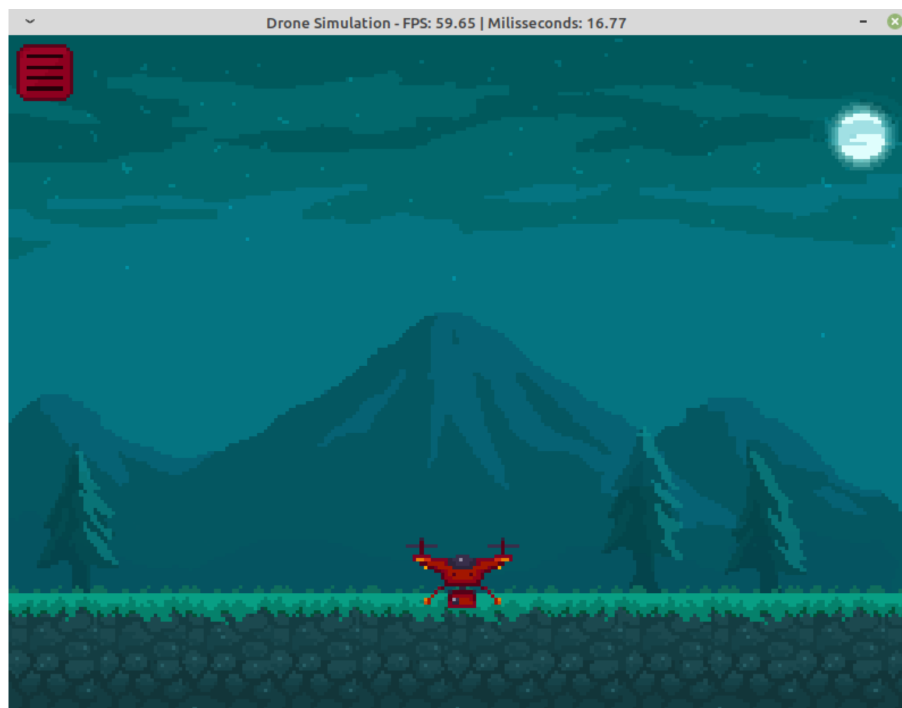
Por fim, a *toolchain Emscripten* foi integrada ao arquivo *Dockerfile* responsável por compilar o código, e passou a produzir *bytecode* executável em navegadores.

A Figura 12 ilustra a primeira tela da simulação. O programa conta com dois métodos de interação: controle manual e automático. O controle manual utiliza as teclas A e D para controlar, respectivamente, os rotores esquerdo e direito. A Figura 13 é uma captura da simulação em modo manual, com o drone em repouso. O controle automático, por sua vez, define a trajetória do drone através de *waipoints*, ou seja, de posições desejadas. Quando o usuário seleciona algum ponto da tela com o mouse, o sistema de controle calcula o erro de posição e rotação, e atua sobre os rotores para que o drone alcance a posição marcada na tela, como ilustra a Figura 14.

Figura 12 – Aplicação: Menu.

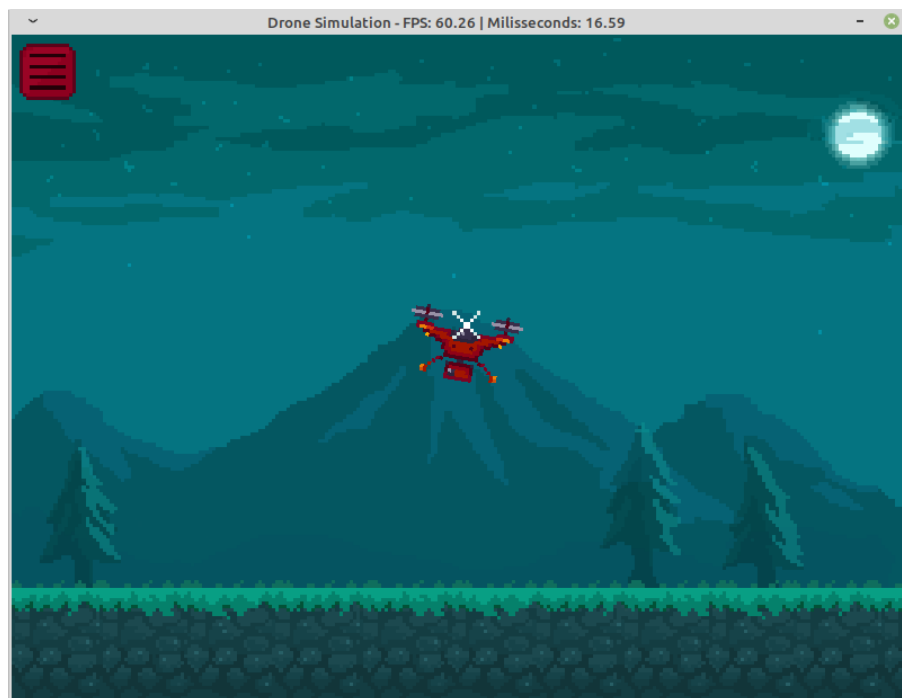
Fonte: Autoria própria (2022).

Figura 13 – Aplicação: Controle manual.



Fonte: Autoria própria (2022).

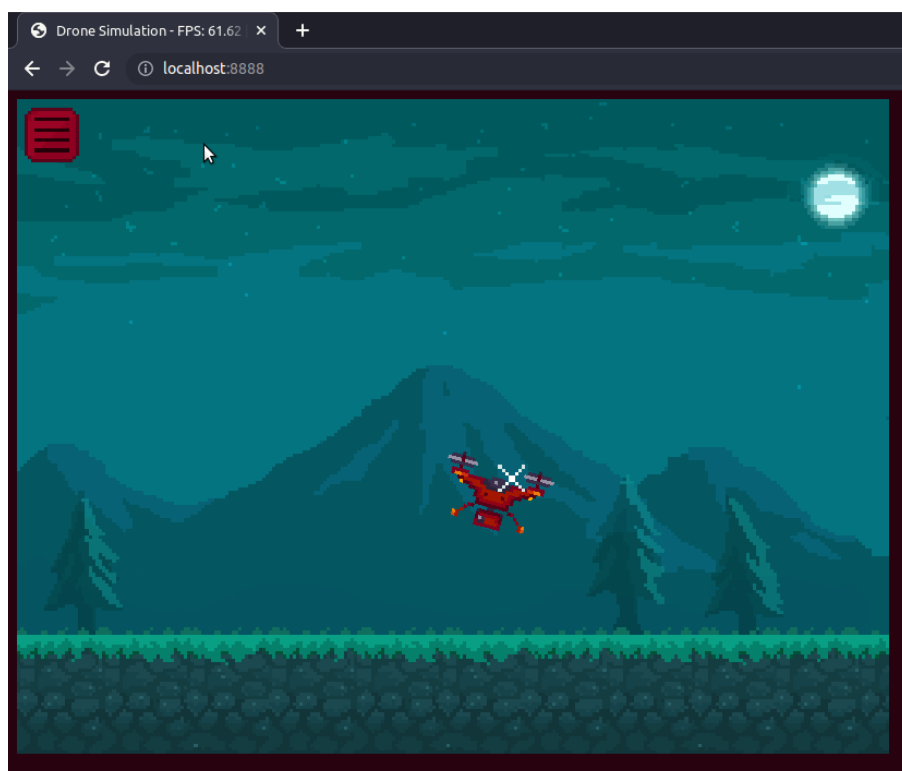
Figura 14 – Aplicação: Controle automático.



Fonte: Autoria própria (2022).

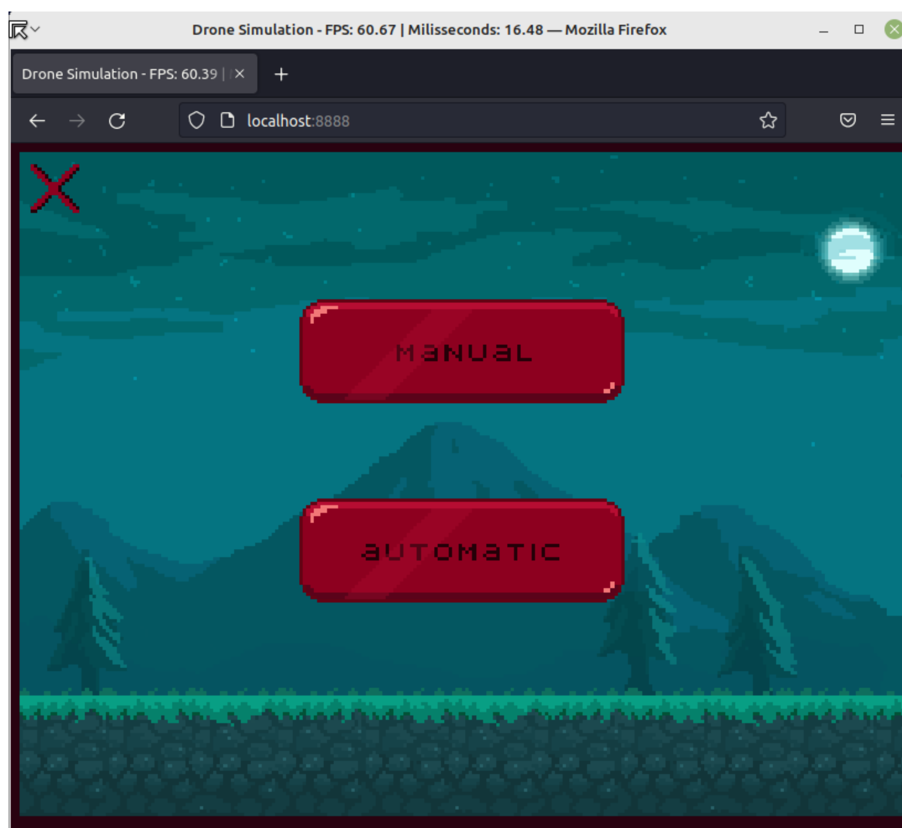
O código foi testado em dois navegadores: Google Chrome e Mozilla Firefox, e as Figuras 15 e 16 ilustram a execução da aplicação em ambos os navegadores, respectivamente.

Figura 15 – Aplicação: Navegador Google Chrome.



Fonte: Autoria própria (2023).

Figura 16 – Aplicação: Navegador Mozilla Firefox.



Fonte: Autoria própria (2023).

5 Conclusão

O objetivo deste trabalho era estabelecer um exemplo de portabilidade de código compilado, executado em um sistema operacional nativamente, para um ambiente *Web*. A aplicação produzida simula um sistema simples de controle de um drone em um espaço 2D, desenvolvido em uma linguagem destinada à produção de código que realiza interface em baixo nível com os sistemas anfitriões.

As dificuldades enfrentadas em cada parte do desenvolvimento condicionou a elaboração de um documentar com todas as etapas necessárias para estabelecer um ambiente de desenvolvimento para tal tipo de *software*, bem como propor uma forma de controle de dependências estrito.

A bibliografia disponível para esse tipo de desenvolvimento — uma aplicação que utiliza bibliotecas gráficas de baixo nível sendo compilada em WASM — ainda é escassa, mas, por meio de pesquisa extensa bibliográfica e diversos testes de implementação, foi possível tecer uma linha do tempo que narra pontos relevante da história do desenvolvimento web, estabelecer uma justificativa para o emprego de uma tecnologia como o *WebAssembly* e estratégia de implementação.

Ademais, para trabalho futuros, sugere-se um desenvolvimento nas estratégias de desenvolvimento *WebAssembly*, com elaboração de tutoriais, protótipos e abstrações de códigos em bibliotecas, com fim de difundir a tecnologia, e habilitar a portabilidade de aplicações tradicionalmente executadas em ambientes nativos, mas que poderiam se beneficiar da flexibilidade que a *Web* pode oferecer.

Referências Bibliográficas

ANDERSON, C. Docker [software engineering]. *IEEE Software*, v. 32, n. 3, p. 102–c3, 2015. Disponível em: <https://doi.org/10.1109/MS.2015.62>. Citado na página 24.

BAST, R.; REMIGIO, R. D. *CMake Cookbook: Building, testing, and packaging modular software with modern CMake*. [S.l.]: Packt Publishing, 2018. ISBN 9781788472340. Citado na página 23.

BHAT, S.; BHAT, S. Understanding docker compose. *Practical Docker with Python: Build, Release, and Distribute Your Python App with Docker*, Springer, p. 165–198, 2022. Disponível em: <https://doi.org/10.1007/978-1-4842-7815-4_7>. Citado 2 vezes nas páginas 37 e 39.

CHEN, B.; CHENG, H. H. Interpretive OpenGL for computer graphics. *Computers & Graphics*, Elsevier BV, v. 29, n. 3, p. 331–339, 2005. Disponível em: <https://doi.org/10.1016/j.cag.2005.03.002>. Citado na página 20.

COMPUTER and World Wide Web Accessibility by Visually Disabled Patients: Problems and Solutions. *Survey of Ophthalmology*, v. 50, n. 4, p. 394–405, 2005. Disponível em: <https://doi.org/10.1016/j.survophthal.2005.04.004>. Citado na página 15.

EMSCRIPTEN. *Emscripten*. 2015. Accessed on: October 5, 2022. Disponível em: <https://emscripten.org/>. Citado na página 23.

GAY, W. Advanced raspberry pi: Raspbian linux and gpio integration. Apress, Berkeley, CA, p. 357–387, 2018. Disponível em: <https://doi.org/10.1007/978-1-4842-3948-3_20>. Citado na página 22.

GOURLEY, D. et al. *HTTP: The Definitive Guide*. [S.l.]: O'Reilly Media, Incorporated, 2002. (Definitive Guides). Citado na página 14.

HAAS, A. et al. Bringing the web up to speed with webassembly. In: . [s.n.], 2017. p. 185–200. Disponível em: <https://doi.org/10.1145/3062341.3062363>. Citado 3 vezes nas páginas 12, 17 e 18.

HILBIG, A.; LEHMANN, D.; PRADEL, M. An empirical study of real-world webassembly binaries: Security, languages, use cases. In: *Proceedings of the Web Conference 2021*. Association for Computing Machinery, 2021. (WWW '21), p. 2696–2708. Disponível em: <https://doi.org/10.1145/3442381.3450138>. Citado na página 19.

KEITH, J. Dom scripting: Web design with javascript and the document object model. Apress, p. 3–10, 2005. Disponível em: <https://doi.org/10.1007/978-1-4302-0062-8_1>. Citado 2 vezes nas páginas 15 e 16.

LATTNER, C.; ADVE, V. *The LLVM instruction set and compilation strategy*. [S.l.: s.n.], 2002. Citado na página 16.

LEHMANN, D.; KINDER, J.; PRADEL, M. Everything old is new again: Binary security of webassembly. In: . USENIX Association, 2020. p. 217–234. Disponível em: <https://doi.org/10.5555/3489212.3489225>. Citado na página 17.

LERNER, J.; TIROLE, J. The open source movement: Key research questions. *European Economic Review*, v. 45, n. 4, p. 819–826, 2001. Disponível em: <https://doi.org/10.1016/S0014-2921(01)00124-6>. Citado na página 23.

MATSUDA, K.; LEA, R. *WebGL programming guide: interactive 3D graphics programming with WebGL*. [S.l.]: Addison-Wesley, 2013. Citado na página 21.

MDN. Webassembly.module - webassembly. 2022. Accessed on: October 5, 2022. Disponível em: <https://developer.mozilla.org/en-US/docs/WebAssembly/JavaScript_interface/Module>. Citado na página 18.

MOURA Éder Alves de. *SEII - Trabalho prático 01*. 2022. Citado 2 vezes nas páginas 25 e 27.

NOAH et al. Implementing a performant scheme interpreter for the web in asm.js. *Computer Languages, Systems & Structures*, v. 49, p. 62–81, 2017. ISSN 1477-8424. Disponível em: <https://doi.org/10.1016/j.cl.2017.02.002>. Citado na página 16.

OLIVEIRA, G. S.; SILVA, A. F. da. Compilação just-in-time: Histórico, arquitetura, princípios e sistemas. v. 20, n. 2, p. 174–213, May 2013. Disponível em: <https://doi.org/10.22456/2175-2745.25803>. Citado na página 18.

PEREIRA, R. et al. Energy efficiency across programming languages: How do energy, time, and memory relate? Association for Computing Machinery, p. 256–267, 2017. Disponível em: <https://doi.org/10.1145/3136014.3136031>. Citado na página 15.

PERNA, G. et al. A first look at http/3 adoption and performance. *Computer Communications*, Elsevier, v. 187, p. 115–124, 2022. Citado na página 15.

ROSSBERG, A. *WebAssembly Specification*. WebAssembly Community Group, 2022. Accessed on: October 5, 2022. Disponível em: <https://webassembly.org>. Citado na página 18.

SHI, Y. et al. Virtual machine showdown: Stack versus registers. Association for Computing Machinery, v. 4, n. 4, jan 2008. Disponível em: <https://doi.org/10.1145/1328195.1328197>. Citado na página 17.

SLETTEN, B. *WebAssembly: the Definitive Guide: Safe, Fast, and Portable Code*. [S.l.]: O'Reilly Media, Incorporated, 2022. Citado 3 vezes nas páginas 14, 16 e 19.

WOO, M. et al. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. 3rd. ed. USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN 0201604582. Citado na página 20.

ZAKAI, A. Emscripten: An llvm-to-javascript compiler. In: . Association for Computing Machinery, 2011. (OOPSLA '11), p. 301–312. Disponível em: <https://doi.org/10.1145/2048147.2048224>. Citado na página 37.

ZAKAI, A. Fast physics on the web using c++, javascript, and emscripten. *Computing in Science & Engineering*, v. 20, n. 1, p. 11–19, 2018. Accessed on: October 5, 2022. Disponível em: <https://doi.org/10.1109/MCSE.2018.110150345>. Citado na página 23.

ANEXO A – Dockerfile

```
FROM ubuntu:20.04

RUN mkdir /home/opengl
WORKDIR /home/opengl


RUN ln -snf /bin/bash /bin/sh

RUN : \
    && apt-get update \
    && DEBIAN_FRONTEND=noninteractive apt-get install -y --no
       -install-recommends \
        software-properties-common \
    && add-apt-repository -y ppa:deadsnakes \
    && DEBIAN_FRONTEND=noninteractive apt-get install -y --no
       -install-recommends \
        python3.8-venv \
    && apt-get clean \
    && rm -rf /var/lib/apt/lists/* \
    && :

RUN python3.8 -m venv /venv
ENV PATH=/venv/bin:$PATH

RUN apt-get update \
    && apt-get install git -y \
    && apt-get install wget -y \
    && apt-get install build-essential -y \
    && apt-get install libssl-dev -y

RUN apt-get install gcc-7 g++-7 g++-7-multilib gfortran-7 -y
   \
    && update-alternatives --install /usr/bin/gcc gcc /usr/
       bin/gcc-7 60 --slave /usr/bin/g++ g++ /usr/bin/g++-7 \
```

```
    && update-alternatives --config gcc \
    && echo `gcc --version | grep ^gcc | sed 's/^.* //g'`

RUN wget http://www.cmake.org/files/v3.23/cmake-3.23.2.tar.gz
RUN tar xzf cmake-3.23.2.tar.gz
WORKDIR /home/opengl/cmake-3.23.2
RUN ./bootstrap
RUN make
RUN make install

WORKDIR /home/opengl
COPY . .

WORKDIR /home/opengl/vendor/emsdk
RUN python -X utf8 emsdk.py install 3.1.12
RUN python -X utf8 emsdk.py activate 3.1.12
RUN chmod +x emsdk_env.sh
RUN source  /home/opengl/vendor/emsdk/emsdk_env.sh

WORKDIR /home/opengl

RUN apt-get install libxrandr-dev libxinerama-dev libxcursor-
   dev libxi-dev extra-cmake-modules vim -y

RUN source  /home/opengl/vendor/emsdk/emsdk_env.sh \
   && cd /home/opengl \
   && mkdir build_emscripten \
   && cd build_emscripten \
   && CC=emcc CXX=em++ cmake .. \
   && make

ARG OPENGL_UID=1000
ARG OPENGL_GID=1000
RUN groupadd -f -g ${OPENGL_GID} opengl \
    && useradd -d /home/opengl -s /bin/bash -g ${OPENGL_GID}
       -u ${OPENGL_UID} opengl

USER opengl
```

```
CMD [ "python", "/home/opengl/build_emscripten/
  run_webassembly.py" ]
```

ANEXO B – Runge-kutta em C++

```
std::vector<GLfloat> Math::FourthOrder::RungeKutta(
    std::vector<GLfloat> param_current_time,
    std::vector<GLfloat> param_integration_step,
    std::vector<GLfloat> param_current_state_vector,
    std::vector<GLfloat> param_current_input_cmd,
    Model* param_model)
{
    std::vector<GLfloat> k_time;
    k_time.push_back(param_current_time[0] +
       param_integration_step[0] / 2.0f);

    std::vector<GLfloat> k_1 = param_model->StateVector(
        param_current_time,
        &param_current_state_vector,
        param_current_input_cmd);

    std::vector<GLfloat> k_input
    {
        param_current_state_vector[0] + (k_1[0] *
           param_integration_step[0] / 2.0f),
        param_current_state_vector[1] + (k_1[1] *
           param_integration_step[0] / 2.0f),
        param_current_state_vector[2] + (k_1[2] *
           param_integration_step[0] / 2.0f),
        param_current_state_vector[3] + (k_1[3] *
           param_integration_step[0] / 2.0f),
        param_current_state_vector[4] + (k_1[4] *
           param_integration_step[0] / 2.0f),
        param_current_state_vector[5] + (k_1[5] *
           param_integration_step[0] / 2.0f),
        param_current_state_vector[6] + (k_1[6] *
           param_integration_step[0] / 2.0f),
        param_current_state_vector[7] + (k_1[7] *
           param_integration_step[0] / 2.0f)
```

```
};

std::vector<GLfloat> k_2 = param_model->StateVector(
     k_time,
     &k_input,
     param_current_input_cmd);

k_input[0] = param_current_state_vector[0] + (k_2[0] *
   param_integration_step[0] / 2.0f);
k_input[1] = param_current_state_vector[1] + (k_2[1] *
   param_integration_step[0] / 2.0f);
k_input[2] = param_current_state_vector[2] + (k_2[2] *
   param_integration_step[0] / 2.0f);
k_input[3] = param_current_state_vector[3] + (k_2[3] *
   param_integration_step[0] / 2.0f);
k_input[4] = param_current_state_vector[4] + (k_2[4] *
   param_integration_step[0] / 2.0f);
k_input[5] = param_current_state_vector[5] + (k_2[5] *
   param_integration_step[0] / 2.0f);
k_input[6] = param_current_state_vector[6] + (k_2[6] *
   param_integration_step[0] / 2.0f);
k_input[7] = param_current_state_vector[7] + (k_2[7] *
   param_integration_step[0] / 2.0f);

std::vector<GLfloat> k_3 = param_model->StateVector(
     k_time,
     &k_input,
     param_current_input_cmd);

k_time[0] = k_time[0] * 2.0f;

k_input[0] = param_current_state_vector[0] + (k_3[0] *
   param_integration_step[0]);
k_input[1] = param_current_state_vector[1] + (k_3[1] *
   param_integration_step[0]);
k_input[2] = param_current_state_vector[2] + (k_3[2] *
   param_integration_step[0]);
k_input[3] = param_current_state_vector[3] + (k_3[3] *
   param_integration_step[0]);
```

```
k_input[4] = param_current_state_vector[4] + (k_3[4] *
   param_integration_step[0]);
k_input[5] = param_current_state_vector[5] + (k_3[5] *
   param_integration_step[0]);
k_input[6] = param_current_state_vector[6] + (k_3[6] *
   param_integration_step[0]);
k_input[7] = param_current_state_vector[7] + (k_3[7] *
   param_integration_step[0]);

std::vector<GLfloat> k_4 = param_model->StateVector(
    k_time,
    &k_input,
    param_current_input_cmd);

std::vector<GLfloat> k_output
{
    param_current_state_vector[0] + (
       param_integration_step[0] / 6.0f) * (k_1[0] + k_2
       [0] * 2.0f + k_3[0] * 2.0f + k_4[0]),
    param_current_state_vector[1] + (
       param_integration_step[0] / 6.0f) * (k_1[1] + k_2
       [1] * 2.0f + k_3[1] * 2.0f + k_4[1]),
    param_current_state_vector[2] + (
       param_integration_step[0] / 6.0f) * (k_1[2] + k_2
       [2] * 2.0f + k_3[2] * 2.0f + k_4[2]),
    param_current_state_vector[3] + (
       param_integration_step[0] / 6.0f) * (k_1[3] + k_2
       [3] * 2.0f + k_3[3] * 2.0f + k_4[3]),
    param_current_state_vector[4] + (
       param_integration_step[0] / 6.0f) * (k_1[4] + k_2
       [4] * 2.0f + k_3[4] * 2.0f + k_4[4]),
    param_current_state_vector[5] + (
       param_integration_step[0] / 6.0f) * (k_1[5] + k_2
       [5] * 2.0f + k_3[5] * 2.0f + k_4[5]),
    param_current_state_vector[6] + (
       param_integration_step[0] / 6.0f) * (k_1[6] + k_2
       [6] * 2.0f + k_3[6] * 2.0f + k_4[6]),
    param_current_state_vector[7] + (
       param_integration_step[0] / 6.0f) * (k_1[7] + k_2
```

```
            [7] * 2.0f + k_3[7] * 2.0f + k_4[7])
    };

    return k_output;
}
```

ANEXO C – Modelo do drone C++

```
Model::Model(
    GLfloat param_max_motor_speed,
    GLfloat param_mass,
    GLfloat param_gravity_acc,
    GLfloat param_wing_lenght,
    GLfloat param_force_constant,
    GLfloat param_momentum_of_inercia,
    GLfloat param_tau,
    GLfloat param_integration_step,
    GLfloat param_scale) : m_delta_time (0.0f), m_controlled
       (false)
{
    m_drone_parameters.scale = param_scale;
    m_drone_parameters.max_motor_speed =
       param_max_motor_speed;
    m_drone_parameters.mass = param_mass;
    m_drone_parameters.gravity_acc = param_gravity_acc *
       -1.0;
    m_drone_parameters.wing_lenght = param_wing_lenght;
    m_drone_parameters.force_constant = param_force_constant;
    m_drone_parameters.momentum_of_inercia =
       param_momentum_of_inercia;
    m_drone_parameters.tau = param_tau;
    m_drone_parameters.integration_step =
       param_integration_step;
    m_drone_parameters.state_vector = {0.0f,0.0f,0.0f,0.0f
       ,0.0f,0.0f,0.0f,0.0f};
    m_drone_parameters.command = {0.0f,0.0f};
}

std::vector<GLfloat> Model::StateVector(
    std::vector<GLfloat> param_time,
    std::vector<GLfloat>* param_state_vector,
    std::vector<GLfloat> param_input_cmd)
```

```
{
    /* Current states */

    glm::vec2 current_w = glm::vec2(
        (*param_state_vector)[0],
        (*param_state_vector)[1]);

    glm::vec2 current_r = glm::vec2(
        (*param_state_vector)[2],
        (*param_state_vector)[3]);

    glm::vec2 current_v = glm::vec2(
        (*param_state_vector)[4] * -1.0f,
        (*param_state_vector)[5]);

    glm::vec1 current_phi = glm::vec1((*param_state_vector)
       [6]);

    glm::vec1 current_omega = glm::vec1((*param_state_vector)
       [7]);

    /* Auxilay force 1 */
    glm::vec1 force_1 = glm::vec1(m_drone_parameters.
       force_constant * pow(current_w[0], 2));

    /* Auxilay force 2 */
    glm::vec1 force_2 = glm::vec1(m_drone_parameters.
       force_constant * pow(current_w[1], 2));

    /* Torque */
    glm::vec1 torque = glm::vec1((m_drone_parameters.
       wing_lenght * (force_1[0] - force_2[0])));

    /* Control force */
    glm::vec2 control_force = glm::vec2(
        0.0f,
        (force_1[0] * - 1.0f) + (force_2[0] * - 1.0f));

    /*  Rotation matrix */
```

```
    glm::mat2 rotation_matrix = glm::mat2(
        cos(current_phi[0]), -1.0f * sin(current_phi[0]),
        sin(current_phi[0]), cos(current_phi[0]));

    /* Derivatives */

    glm::vec2 w_dot = glm::vec2(
        ((-1.0f * current_w[0]) + param_input_cmd[0]) /
          m_drone_parameters.tau,
        ((-1.0f * current_w[1]) + param_input_cmd[1]) /
          m_drone_parameters.tau);

    glm::vec2 r_dot = current_v;
    glm::vec2 v_dot = rotation_matrix * control_force;
    v_dot.y = (1 / m_drone_parameters.mass) * (v_dot.y +
       (-1.0f * m_drone_parameters.mass * m_drone_parameters.
       gravity_acc));
    v_dot.x = (1 / m_drone_parameters.mass) * v_dot.x;
    glm::vec1 phi_dot = current_omega;
    glm::vec1 omega_dot = glm::vec1(torque/
       m_drone_parameters.momentum_of_inercia);

    m_drone_parameters.state_vector[0] = w_dot[0];
    m_drone_parameters.state_vector[1] = w_dot[1];
    m_drone_parameters.state_vector[2] = r_dot[0] *
       m_drone_parameters.scale;
    m_drone_parameters.state_vector[3] = r_dot[1] *
       m_drone_parameters.scale;
    m_drone_parameters.state_vector[4] = v_dot[0] * 1.0f *
       m_delta_time;
    m_drone_parameters.state_vector[5] = v_dot[1] * 1.0f *
       m_delta_time;
    m_drone_parameters.state_vector[6] = phi_dot[0];
    m_drone_parameters.state_vector[7] = omega_dot[0] * 1.0f
       *  m_delta_time ;

    return m_drone_parameters.state_vector;
}
```

ANEXO D – Sistema de controle em C++

```
std::vector<GLfloat> Model::Control_System(
    std::vector<GLfloat> param_state_vector)
{
    /* Control restrictions */

    float gravity_force = -1.0f * m_drone_parameters.mass *
       m_drone_parameters.gravity_acc;

    float phi_max = 15.0f * M_PI / 180.0f;

    float w_max = 15000.0f;

    float max_control_force = m_drone_parameters.
       force_constant * pow(w_max, 2);

    float max_torque = m_drone_parameters.wing_lenght *
       m_drone_parameters.force_constant * pow(w_max, 2);

    /* Current states */

    glm::vec2 current_w = glm::vec2(
        (param_state_vector)[0],
        (param_state_vector)[1]);

    glm::vec2 current_r = glm::vec2(
        (param_state_vector)[2],
        (param_state_vector)[3]);

    glm::vec2 current_v = glm::vec2(
        (param_state_vector)[4] * -1.0f,
        (param_state_vector)[5]);

    glm::vec1 current_phi = glm::vec1((param_state_vector)
```

```
    [6]);

glm::vec1 current_omega = glm::vec1((param_state_vector)
    [7]);

/* Position control */
glm::vec1 kp_p(0.075f * 1250.1f);
glm::vec1 kd_p(0.25f  * 140.1f);

Control_CalculatePositionError();
glm::vec2 position_error = Control_GetPositionError();

std::cout << "Position error: " << position_error.x << "
    " << position_error.y << std::endl;
std::cout << "Position error: " << position_error.x << "
    " << position_error.y << std::endl;

glm::vec2 position_command(0.0f, 0.0f);

glm::vec2 velocity_error(position_command - current_v);

std::cout << "Velocity error: " << velocity_error.x << "
    " << velocity_error.y << std::endl;
std::cout << "Velocity error: " << velocity_error.x << "
    " << velocity_error.y << std::endl;

float control_action_force_x = kp_p[0] * position_error
    [0] + kd_p[0] * velocity_error[0];
float control_action_force_y = kp_p[0] * position_error
    [1] + kd_p[0] * velocity_error[1] - (gravity_force
    *0.89f);

std::cout << "KP_P: " << kp_p[0]
          << " | Pos Error: " << position_error[1]
          << " | KD_P: " << kd_p[0]
          << " | Vel Error: " << velocity_error[1]
          << " | Gravity: " << gravity_force
          << " | C_A Y: " << control_action_force_y <<
             std::endl;
```

```
control_action_force_y = std::min(-0.001f *
   max_control_force, std::max(control_action_force_y,
   -0.8f * (float)max_control_force));

/* Attitude control */
float phi_control_action = -1.0f * atan2(-1.0f *
   control_action_force_x, 1.0f * control_action_force_y)
   ;

std::cout << "Phi control action: " << phi_control_action
    << " | " << phi_max << std::endl;

std::cout << "X before: " << control_action_force_x <<
   std::endl;
std::cout << "Y before: " << control_action_force_y <<
   std::endl;


if (std::abs(phi_control_action) > phi_max)
{
    float signal = phi_control_action / std::abs(
       phi_control_action);
    phi_control_action = signal * phi_max;
    control_action_force_x =  control_action_force_y *
       tan(phi_control_action);
}

std::cout << "X after: " << control_action_force_x << std
   ::endl;
std::cout << "Y after: " << control_action_force_y << std
   ::endl;

glm::vec2 control_action_force_xy(control_action_force_x,
    control_action_force_y);

float norm = 0.0f;

norm = glm::length(control_action_force_xy);
```

```
std::cout << "norm:" << norm << " x:" <<
   control_action_force_x << " y:" <<
   control_action_force_y << std::endl;

glm::vec2 control_action_force_1_2(norm / 2.0f);

/* Attitude constants */
float kp_a = 0.75f * 10.7f;
float kd_a = 0.05f * 20.5f;

float phi_error = (phi_control_action - current_phi[0]);

float omega_error = (0.0f - current_omega[0]);

float control_action_torque = 0.0f;

control_action_torque = kp_a * phi_error + kd_a *
   omega_error;

std::cout << " | " << kp_a << " | " <<  phi_error << " |
   " <<  kd_a << " | " <<  omega_error << std::endl;

control_action_torque = std::max(-0.2f * (float)
   max_torque, std::min(control_action_torque, 0.2f * (
   float)max_torque));

/* Forces delta */

float delta_1_2 = std::abs(control_action_torque);

std::cout << "Action 1 2: " << control_action_force_1_2
   [0] << " " << control_action_force_1_2[1] << " Delta:
   " << delta_1_2 << std::endl;

std::cout << control_action_torque << std::endl;

if (control_action_torque >= 0.0f)
{
```

```
        (control_action_force_1_2[0] =
           control_action_force_1_2[0] + delta_1_2);
        (control_action_force_1_2[1] =
           control_action_force_1_2[1] - delta_1_2);
    }
    else
    {
        control_action_force_1_2[0] =
           control_action_force_1_2[0] - delta_1_2;
        control_action_force_1_2[1] =
           control_action_force_1_2[1] + delta_1_2;
    }

    float w_1 = sqrt(control_action_force_1_2[0] / (
       m_drone_parameters.force_constant));
    float w_2 = sqrt(control_action_force_1_2[1]/ (
       m_drone_parameters.force_constant));

    float w_1_out = std::max(0.0f, std::min(w_1, w_max));
    float w_2_out = std::max(0.0f, std::min(w_2, w_max));

    // float w_1_out = w_1;
    // float w_2_out = w_2;

    std::vector<GLfloat> w_output_action {w_1_out, w_2_out};

    return w_output_action;
}
```

ANEXO E – Classe de aplicação

```
#include "application.h"

std::function<void()> registered_loop;
void loop_iteration() {
    registered_loop();
}

Application::Application(
        unsigned int param_screen_width,
        unsigned int param_screen_height)
        : m_screen_width (param_screen_width),
          m_screen_height (param_screen_height)
{
    m_drone = new Game(m_screen_width, m_screen_height);
}

Application::~Application() {}

void Application::Run()
{
    Input &callback = Input::GetInstance();
    callback.SetGameInstance(m_drone);
    std::cout << "DEBUG: Game instance set" << std::endl;

    if (!glfwInit()) {
        std::cout << "DEBUG: Could not start Glfw" << std::
            endl;
        throw std::runtime_error("Couldn't init GLFW");
    }
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 0);

#ifdef __APPLE__
    glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
#endif
```

```
glfwWindowHint(GLFW_RESIZABLE, false);

GLFWwindow* window = glfwCreateWindow(m_screen_width,
   m_screen_height, "Drone", nullptr, nullptr);
glfwMakeContextCurrent(window);

if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
{
    std::cout << "DEBUG: Failed to initialize GLAD" <<
       std::endl;
    std::cout << "Failed to initialize GLAD" << std::endl
       ;
}

glfwSetKeyCallback(window, callback.key_callback);
glfwSetMouseButtonCallback(window, callback.
   mouse_button_callback);
glfwSetFramebufferSizeCallback(window, callback.
   framebuffer_size_callback);

glViewport(0, 0, m_screen_width, m_screen_height);
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

m_drone->Init();
std::cout << "DEBUG: Drone Init" << std::endl;

GLfloat delta_time = 0.0f;
GLfloat delta_last_time = 0.0f;

GLfloat last_sprite_frame_time = 0.0f;

registered_loop = [&]()
{
    GLfloat delta_current_time = glfwGetTime();
    GLfloat current_sprite_frame_time = glfwGetTime();

    bool tick = false;
```

```
delta_time = delta_current_time - delta_last_time;
delta_last_time = delta_current_time;

std::stringstream FPS(std::stringstream::in | std::
   stringstream::out);
FPS << std::fixed << std::setprecision(2) << (1.0f /
   delta_time);

std::stringstream milisseconds(std::stringstream::in
   | std::stringstream::out);
milisseconds << std::fixed << std::setprecision(2) <<
    (delta_time * 1000.0f);

std::stringstream FPS_title(std::stringstream::in |
   std::stringstream::out);
FPS_title << "Drone Simulation - FPS: "
          << FPS.str()
          << " | Milisseconds: "
          << milisseconds.str();

glfwSetWindowTitle(window, FPS_title.str().c_str());

if (current_sprite_frame_time -
   last_sprite_frame_time >= 0.075f)
{
    last_sprite_frame_time =
       current_sprite_frame_time;

    tick = true;
}

glfwPollEvents();

m_drone->ProcessInput(delta_time);
m_drone->Update(tick);

glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT);
m_drone->Render();
```

```
        glfwSwapBuffers(window);
    };

    emscripten_set_main_loop(loop_iteration, 0, 1);

    ResourceManager::Clear();

    glfwTerminate();
}
```