Fundação Educacional do Município de Assis
Instituto Municipal de Ensino Superior de Assis
Campus "José Santilli Sobrinho"

GUSTAVO ARAUJO PINTARI

WEBASSEMBLY E RUST: POTENCIAIS PARA O DESENVOLVIMENTO DE APLICAÇÕES WEB

Assis/SP
2023

Fundação Educacional do Município de Assis
Instituto Municipal de Ensino Superior de Assis
Campus "José Santilli Sobrinho"

GUSTAVO ARAUJO PINTARI

WEBASSEMBLY E RUST: POTENCIAIS PARA O DESENVOLVIMENTO DE APLICAÇÕES WEB

Trabalho de Conclusão de Curso apresentado ao Curso de Bacharelado em Ciência da Computação do Instituto Municipal do Ensino Superior de Assis – IMESA e Fundação Educacional do Município de Assis – FEMA, como requisito para a obtenção do Certificado de Conclusão.

Orientando: Gustavo Araujo Pintari
Orientador: Dr. Almir Rogério Camolesi

Assis/SP
2023

FICHA CATALOGRÁFICA

PINTARI, Gustavo Araujo.
WebAssembly e Rust: Potenciais para o desenvolvimento de aplicações web / Gustavo Araujo Pintari. Fundação Educacional do Município de Assis – FEMA – Assis, 2023.
65 f.
Orientador: Dr. Almir Rogério Camolesi
Trabalho de Conclusão de Curso – Fundação Educacional do Município de Assis – Fema, curso de Ciência da Computação, Assis, 2023.

1. WebAssembly. 2. Rust. 3. Jogo 4. Tecnologia 5. Web

CDD:
Biblioteca da FEMA

WEBASSEMBLY E RUST: POTENCIAIS PARA O DESENVOLVIMENTO DE APLICAÇÕES WEB

GUSTAVO ARAUJO PINTARI

Trabalho de Conclusão de Curso apresentado ao Instituto Municipal de Ensino Superior de Assis, como requisito do Curso de Graduação, avaliado pela seguinte comissão examinadora:

Orientador: Dr. Almir Rogério Camolesi

Examinador: Me. Fábio Eder Cardoso

Assis/SP
2023

“Nada na vida deve ser temido, apenas compreendido. Agora é hora de compreender mais, para temer menos.” - Marie Curie

RESUMO

Com a crescente aspiração das empresas em portar seus serviços diretamente para a internet, a web está passando por uma grande transformação tecnológica. Para atender à demanda de aplicações cada vez mais robustas em termos de processamento que possam funcionar diretamente no navegador, surgiu a tecnologia WebAssembly (WASM). Essa tecnologia funciona como um alvo de compilação, permitindo que qualquer linguagem de programação utilize seu compilador e seja interpretada diretamente no navegador. Este estudo tem como objetivo explorar a tecnologia WASM e demonstrar suas características através do desenvolvimento de um jogo interativo utilizando a linguagem de programação Rust.

Palavras-chave: WebAssembly; Rust; Jogo; Tecnologia; Web

ABSTRACT

With companies increasingly aspiring to directly port their services to the internet, the web is undergoing a major technological transformation. To meet the demand for applications that are increasingly robust in terms of processing power and can run directly in the browser, the Web Assembly (WASM) technology has emerged. This technology acts as a target compiler, allowing any programming language to use its compiler and be interpreted directly in the web browser. This study aims to explore WASM technology and demonstrate its characteristics through the development of an interactive game using the Rust programming language.

Keywords: WebAssembly; Rust; Game; Technology; Web

LISTA DE ILUSTRAÇÕES

LISTA DE TRECHOS DE CÓDIGO

SUMÁRIO

1. INTRODUÇÃO

Não sendo diferente de outras áreas da tecnologia, a Web também evoluiu muito com o passar dos anos. Desde a criação do WWW (*World Wide Web*) por Tim Berners-Lee em 1989 e do aparecimento das primeiras páginas de internet, a Web precisou passar por diversas transformações para lidar com o crescente aumento em número de usuários. Segundo consta o CETIC (2021), “O Brasil tem 152 milhões de usuários de Internet, o que corresponde a 81% da população do país com 10 anos ou mais”. Em um cenário onde praticamente tudo é realizado através da internet, as empresas vêm enxergando na Web uma poderosa via para migrar seus serviços.

Um aplicativo web pode ser definido como um recurso que é executado diretamente no navegador de internet do usuário, não sendo preciso realizar nenhuma instalação prévia. Os Web Apps estão sendo cada vez mais agregados pelas empresas em função de sua dinamicidade e praticidade em distribuir serviços. Essa abordagem promove a eficiência, acessibilidade e maior desempenho dos negócios (KACHAN, 2020).

Quando analisamos a atual conjuntura do desenvolvimento de aplicações web, é impossível negar a importância e predominância da linguagem de programação JavaScript (JS) e de todo o seu ecossistema de *plugins* e *frameworks*. O JS foi criado por Brendan Eich em 1995 e atualmente é nativamente interpretado pela maioria dos navegadores modernos, graças ao motor V8 escrito e desenvolvido pela Google. Segundo DiPierro (2018), o motor V8 é o que possibilita o JavaScript a ser uma linguagem de extrema rapidez, sendo em alguns casos comparada até mesmo com linguagens como C e C++. Por esse motivo, com o passar dos anos, grande parte dos navegadores de internet disponíveis no mercado adotaram o JS como o padrão para o desenvolvimento de sistemas web.

Contudo, a grande demanda de Web Apps cada vez mais robustos em termos de processamento estão exigindo mais do que apenas o JavaScript sozinho pode executar. Para sanar esses desafios, os desenvolvedores são muitas vezes obrigados a combinar funcionalidades de diversas outras linguagens de programação, assim aumentando de forma desnecessária a complexidade do sistema como um todo (ŠIPEK et al, 2021, p 01).

Neste contexto, os principais fornecedores de navegadores se uniram com a finalidade de colaborar na criação de uma nova plataforma de distribuição de aplicativos web. O WebAssembly (WASM) nasce com a proposta de, ao lado do JavaScript, atuar como um

compilador-alvo, possibilitando que qualquer linguagem de programação utilize o seu compilador e seja assim interpretada no navegador de internet. De acordo com Peham (2015) “Isso definitivamente significa melhorias de desempenho no navegador. E nos dá acesso a um conjunto de blocos de construção de baixo nível, como uma gama de tipos e operações.“

O WASM viabiliza o desenvolvimento de aplicativos web através de linguagens de programação tradicionalmente restritas ao *desktop* como C, Rust e até Cobol em uma velocidade de compilação quase nativa. Isto indica que a partir desta tecnologia, cada vez mais aplicações serão portadas para este novo paradigma. Vale ressaltar que, mesmo essa sendo uma tecnologia relativamente nova, o WebAssembly já é suportado nativamente pela grande parte dos navegadores de internet disponíveis no mercado.

O objetivo deste trabalho é analisar as características da tecnologia WebAssembly quando associada a uma linguagem de programação baixo nível. Tais noções serão apresentadas através do desenvolvimento de um jogo Snake. O jogo será escrito utilizando as linguagens Rust e Javascript, sendo compiladas e interpretadas pelo WASM.

1.1 OBJETIVOS

Este trabalho tem por objetivo realizar uma revisão bibliográfica sobre a tecnologia WebAssembly e a linguagem de programação Rust, especificando seus atributos e funcionalidades. Para exemplificar a utilização e associação dessas tecnologias de forma prática, será desenvolvido um *webgame* seguindo o modelo do jogo de gráficos bidimensionais Snake.

A escolha de utilizar um jogo como prova de conceito decorre do fato de que, ao simular de forma simplificada os aspectos de uma aplicação Web no lado do cliente, um *webgame* possibilita explorar mais profundamente o uso do WebAssembly em conjunto com a linguagem Rust. Além disso, por meio de um *webgame*, é possível explorar recursos como gráficos, animações e interações do usuário de maneira mais adequada.

Ao final deste estudo, espera-se apresentar o WebAssembly e a linguagem Rust como ferramentas poderosas e complementares, capazes de enriquecer a gama de recursos disponíveis para desenvolvedores de aplicações web.

1.2 JUSTIFICATIVAS

A Web 3.0, ou Web Semântica, é caracterizada por uma internet mais democrática, inteligente e organizada, que possibilita a criação de uma rede de dados interconectados e o processamento inteligente de informações. Diferentemente da Web 2.0, em que o conteúdo era gerado apenas pelos usuários e as informações eram descentralizadas, a Web 3.0 introduz uma dinâmica em que os usuários estão constantemente movimentando dados e interagindo com aplicações. De acordo com Oliveira, Maziero e Araújo (2018, p 63) “O avanço tecnológico exige que o cidadão esteja em contato com as tecnologias de rede para ser inserido nesta nova sociedade virtual.”

No entanto, à medida que cada vez mais pessoas acessam uma internet orientada a dados e aplicações, surgem novas necessidades, dentre as quais, a demanda por aplicações seguras e performáticas. Neste contexto, o presente trabalho se justifica em razão da carência de tecnologias disponíveis no mercado para o desenvolvimento de aplicações web que demandam alto desempenho tanto em nível de cliente quanto em nível de servidor. Tecnologias como o WebAssembly e linguagens como o Rust prometem mudar esse cenário, já que adotam o propósito de fornecer alta performance e segurança.

1.3 MOTIVAÇÕES

De acordo com Sletten (2021), o WebAssembly apresenta uma combinação inovadora de alto desempenho e segurança, tornando-se uma tecnologia promissora para desenvolvedores de diferentes áreas. Por essa razão, essa tecnologia tem sido utilizada em diversas camadas de aplicações, com usos que vão desde a Web, desktops e servidores. Seu uso também tem se expandido para ecossistemas de internet das coisas (IOT), *blockchains*, jogos e plataformas de *streaming*, que demandam um alto processamento computacional do navegador. O WASM tem se mostrado uma solução eficiente para suprir essa demanda de processamento, oferecendo pela primeira vez uma alternativa segura e de alto desempenho.

Embora seja uma tecnologia relativamente recente, o WebAssembly tem despertado interesse de grandes empresas que vêm trabalhando para que seus aplicativos aproveitem dos benefícios agregados por essa tecnologia. Portanto, espera-se um crescimento significativo no número de sistemas que utilizem e aproveitem as vantagens proporcionadas pelo WASM.

Já a linguagem Rust tem se destacado por oferecer um controle direto sobre o hardware, mantendo uma preocupação especial com a segurança, especialmente no que se refere a acessos de ponteiros na memória. Devido a essas características, o Rust é frequentemente comparado com linguagens como C e C++, já que as três são utilizadas em aplicações de baixo nível e não possuem um coletor de lixo (*garbage collector*) embutido.

Conforme a Stack Overflow Developer Survey 2022, pesquisa anual realizada desde 2011 pela plataforma de perguntas e respostas Stack Overflow, a linguagem Rust foi eleita pela sétima vez consecutiva como a linguagem mais amada pelos desenvolvedores (STACK OVERFLOW, 2022). A popularidade do Rust pode ser explicada por vários motivos, como a sua capacidade de proporcionar segurança, desempenho e confiabilidade, além de possuir uma comunidade ativa e um ecossistema de ferramentas e bibliotecas em constante evolução. Além disso, o Rust recentemente foi incorporado na versão 6.1 do kernel Linux, com o objetivo de substituir gradualmente módulos que antes eram escritos em C (VAUGHAN-NICHOLS, 2022).

1.4 PERSPECTIVAS DE CONTRIBUIÇÃO

Este trabalho visa contribuir para a disseminação da tecnologia WebAssembly e da linguagem Rust, explorando suas variadas aplicações no contexto atual de desenvolvimento web. Isso é especialmente relevante, uma vez que ambas são tecnologias recentes e ainda não tão familiares para os profissionais da área.

Com base na revisão bibliográfica e na criação do *webgame*, espera-se que este trabalho se torne uma fonte abrangente e confiável a respeito das tecnologias que aqui forem tratadas. Assim, visa-se fornecer informações valiosas para estudantes e desenvolvedores que buscam empregar tais tecnologias em seus projetos futuros.

1.5 METODOLOGIA DE PESQUISA

Este trabalho será fundamentado em pesquisas bibliográficas que abrangem artigos acadêmicos, livros e sites relevantes. A revisão bibliográfica terá como foco a caracterização da tecnologia WebAssembly e da linguagem de programação Rust, além de outras tecnologias relevantes como o Javascript, para contextualizar os temas centrais deste trabalho.

Neste contexto, a metodologia adotada será a de pesquisa exploratória, com um estudo sistemático das tecnologias Rust e WebAssembly de forma individual. Após a caracterização dessas tecnologias, será proposto um estudo de caso para demonstrar a associação entre elas, por meio do desenvolvimento de um jogo Snake, com o intuito de ilustrar os conceitos abstraídos.

O jogo será programado integralmente utilizando as linguagens Rust e Javascript. No entanto, passará pelo processo de compilação para o formato binário do WebAssembly, permitindo sua execução na máquina virtual WASM VM, a qual é incorporada dentro de um navegador web.

1.6 ESTRUTURA DO TRABALHO

O presente trabalho está estruturado em cinco capítulos. O primeiro capítulo oferece uma introdução ao estudo, onde são delineados os objetivos, justificativas, motivações, perspectivas de contribuição e a metodologia adotada. O capítulo seguinte, intitulado “A Web antes do WebAssembly”, explora os desdobramentos e obstáculos da web que culminaram na concepção do WebAssembly, traçando uma trajetória pelas tecnologias que o antecederam. O terceiro capítulo aborda algumas das características distintivas da linguagem Rust. No quarto capítulo, o processo de desenvolvimento do jogo Snake é detalhado. Por fim, o capítulo cinco engloba as conclusões e sugestões para trabalhos futuros, seguidas pelas referências bibliográficas.

2. A WEB ANTES DO WEBASSEMBLY

Em 1993, a necessidade de um navegador que permitisse um acesso mais intuitivo e fácil à internet foi atendida com o lançamento do NCSA Mosaic. Criado por uma equipe de desenvolvedores liderada por Marc Andreessen e Eric Bina, o Mosaic foi o primeiro navegador web a trazer consigo uma interface gráfica de usuário (GUI), oferecendo botões de navegação e suporte nativo a imagens e hyperlinks.

Com o crescimento contínuo da popularização da internet, várias empresas investiram na criação de navegadores e novas tecnologias para atrair usuários, dentre elas estava o JavaScript. Em 1994, Marc Andreessen deixou a equipe do NCSA Mosaic para fundar a Netscape Communications e lançar seu próprio navegador, o Netscape. No ano seguinte, a nova equipe de Andreessen anuncia o JavaScript, uma linguagem de script que permitia ser facilmente integrada ao HTML e trazer funcionalidades como manipulação de elementos. O JS significou que páginas outrora estáticas, agora poderiam ter funcionalidades dinâmicas, como validação de formulários e processamento de dados (GRILLO e FORTES, 2008, p 04).

À medida que os navegadores surgiam e evoluíam, hardware e conexões de rede acompanhavam esse progresso. A web trouxe uma mudança de paradigma significativa em relação à construção de aplicações. De forma a promover questões como escalabilidade, acessibilidade, interoperabilidade e segurança, se tornava necessário estabelecer distinções de contextos entre cada camada de uma aplicação. Nesse momento, modelos de computação distribuída, como o modelo cliente-servidor, se tornaram os mais populares para a arquitetura dessas aplicações.

Conforme Tanenbaum e Steen (2007), o modelo cliente-servidor divide os contextos de uma aplicação em duas camadas distintas: a camada cliente, que é responsável por realizar as requisições, e a camada servidor, que é responsável por fornecer as respostas a essas requisições. Essa arquitetura, além de tornar o desenvolvimento e a manutenção mais eficientes, permitiu uma melhor divisão de responsabilidades entre as camadas. Nessa situação, geralmente, a camada de cliente fica responsável por implementar a interface de usuário, enquanto que a camada de servidor fica centrada na lógica da aplicação.

Notou-se também que uma aplicação executada a partir de um navegador poderia trazer muito mais mobilidade e praticidade tanto para o desenvolvedor quanto para o usuário

final. Dessa forma, sistemas que antes eram executados diretamente no hardware do usuário, agora estavam sendo migrados para funcionar dentro do navegador. Como resultado, a Web se tornou uma plataforma essencial para o desenvolvimento e distribuição de aplicações em larga escala.

Nesse enquadramento, o JavaScript consolidou-se no mercado e ganhou o suporte de diversos navegadores. Além disso, a linguagem continuou em constante evolução, com lançamentos acompanhando a especificação ECMA-262 (também conhecida como ECMAScript). No entanto, apesar das diversas inovações e mudanças de paradigma, o ambiente Web ainda apresentava limitações em termos de recursos e capacidade. Considerando essa situação, as interações disponibilizadas pelo JavaScript ainda estavam restritas ao âmbito do cliente, limitando-se a interações básicas de ação e reação.

2.1 MECANISMOS BASEADOS EM PLUGIN

A chegada do Ajax (*Asynchronous JavaScript And XML*) marcou o início de uma mudança nesse cenário, uma vez que permitia que o JavaScript realizasse solicitações assíncronas para o servidor, melhorando significativamente a experiência do usuário. No entanto, para aplicações web necessitadas de maiores recursos computacionais era comum recorrer a tecnologias como o Adobe Flash Player, Java Applet e Microsoft Silverlight (WIRFS-BROCK e EICH, p 59).

Essas tecnologias compartilhavam um mesmo princípio de funcionamento: eram instaladas previamente no computador do usuário e se comunicavam com o navegador através de mecanismos de *plugin*, permitindo a utilização de recursos diretos do hardware para execução de tarefas. No entanto, ao longo do tempo, ficou claro que apesar da dinamicidade que traziam aos navegadores, essas tecnologias sofriam graves falhas de segurança. Além disso, os mecanismos de *plugin* geravam diversos problemas de compatibilidade entre sistemas operacionais e navegadores.

O antigo modelo baseado no uso de *plugins* caiu em desuso, e as tecnologias correspondentes foram descontinuadas em favor de abordagens mais seguras. Segundo Subramanian (2019), tais abordagens se tornaram viáveis graças ao lançamento do interpretador V8 da Google e às mudanças de paradigma da web, incluindo a ampla adoção do HTML5.

2.2 INTERPRETADOR V8 E COMPILAÇÃO JUST-IN-TIME.

Em 2008, a Google lançou o interpretador V8, que se tornou um marco importante na história do JavaScript e no desenvolvimento web como um todo. O principal objetivo desse interpretador era aprimorar a velocidade de execução do JavaScript no navegador Chrome, utilizando o paradigma de compilação *just-in-time* (JIT).

Antes do V8, o JavaScript era executado de forma puramente interpretada nos navegadores disponíveis no mercado. Nessa abordagem, o código era traduzido e executado linha a linha, o que tornava o processo de execução mais lento na maioria dos cenários. Em contrapartida, no processo de compilação, o código antes de ser executado, é avaliado como um todo, permitindo que o compilador realize otimizações e adote caminhos mais eficientes para traduzir as instruções (WAGNER, 2023).

Percebeu-se que para otimizar a velocidade do JavaScript, era essencial adotar uma abordagem que fosse adequada ao hardware da época, mas que também se aproximasse da velocidade de linguagens compiladas. Esse objetivo foi alcançado por meio do paradigma *just-in-time*.

Conforme Lyamkin (2020), o JIT combina o melhor da interpretação e da compilação, pois identifica os trechos de código mais frequentemente executados pelo interpretador e os traduz para código de máquina otimizado. Essa abordagem híbrida permite ao JavaScript alcançar um desempenho mais eficiente, tirando proveito das vantagens de ambas as estratégias.

O JIT só entra em ação após o código ter sido interpretado pelo menos uma vez, como afirmado por Frota (2013) “Quando se executa a aplicação pela primeira vez, ela é executada em modo interpretado, ou seja, não é feita nenhuma otimização inicial. Durante a utilização da aplicação, são feitas estatísticas de runtime.” É com base nessas estatísticas que o JIT identifica as partes e trechos de código que podem ser otimizados e os agrupa nos chamados *hotspots*.

Com os *hotspots* agrupados, o JIT compilará cada trecho em código de máquina otimizado. Cada trecho de *hotspot*, agora otimizado, será incorporado ao código original, substituindo as partes interpretadas correspondentes, o que resultará em um ganho significativo de desempenho.

A implementação da compilação JIT no V8 trouxe um notável aumento de desempenho ao JavaScript. Essa melhoria possibilitou expandir as capacidades da linguagem para além do ambiente tradicional do navegador, permitindo a criação de *frameworks* e servidores de aplicação por meio do ecossistema Node.js. Essa versatilidade e aprimoramento do desempenho proporcionados pela V8 contribuíram significativamente para a popularização e disseminação do JavaScript como uma das principais linguagens no cenário do desenvolvimento web moderno.

2.3 COMPILADOR EMSCRIPTEN E SUBCONJUNTO ASM.JS

Mesmo com o aumento de performance do JavaScript a partir do interpretador V8, a busca por velocidade em tempo de compilação na web persistiu. Em 2011, surgiu o projeto Emscripten, representando um marco importante ao oferecer uma forma inédita de compilar as linguagens C e C++ para JavaScript.

Essa inovação foi um passo significativo para a web, pois possibilitou o uso das bibliotecas de C e C++ em um contexto web, abrindo caminho para a portabilidade de diversas aplicações para a plataforma online. Contudo, é importante notar que, apesar desse avanço, o produto final continuava sendo JavaScript, o que não resultava em grande progresso no quesito de velocidade de execução.

Em 2013, a Mozilla lançou o projeto Asm.js como uma tentativa de aprimorar o desempenho do Emscripten. Como o C e C++ eram compilados diretamente para JavaScript, a solução adotada foi criar um subconjunto limitado de instruções JavaScript, já otimizadas para essas linguagens (Resig, 2013).

A ideia de restringir um subconjunto limitado de instruções era tornar o escopo mais previsível, reduzindo as possibilidades e, assim, aumentando a taxa de otimização proporcionada pela compilação JIT. No entanto, os esforços em torno do Asm.js não conduziram a um aumento significativo de desempenho, pois sua performance ainda era bastante similar ao que o JavaScript padrão proporcionava (MDN Web Docs, 2023). Ademais, o suporte integral ao Asm.js foi disponibilizado apenas pelo Mozilla Firefox, o navegador onde o projeto teve origem.

Apesar das expectativas iniciais em relação ao Asm.js, sua adoção ficou restrita a apenas um navegador, o que limitou seu impacto na melhoria geral do desempenho do JavaScript em diferentes ambientes de navegação. Diante dessa situação, continuou-se a explorar

outras alternativas e abordagens para aprimorar a velocidade e a eficiência do JavaScript, buscando atender melhor às demandas do desenvolvimento web.

A figura 1 ilustra uma linha do tempo com as tecnologias citadas neste capítulo que antecederam a chegada do WebAssembly.

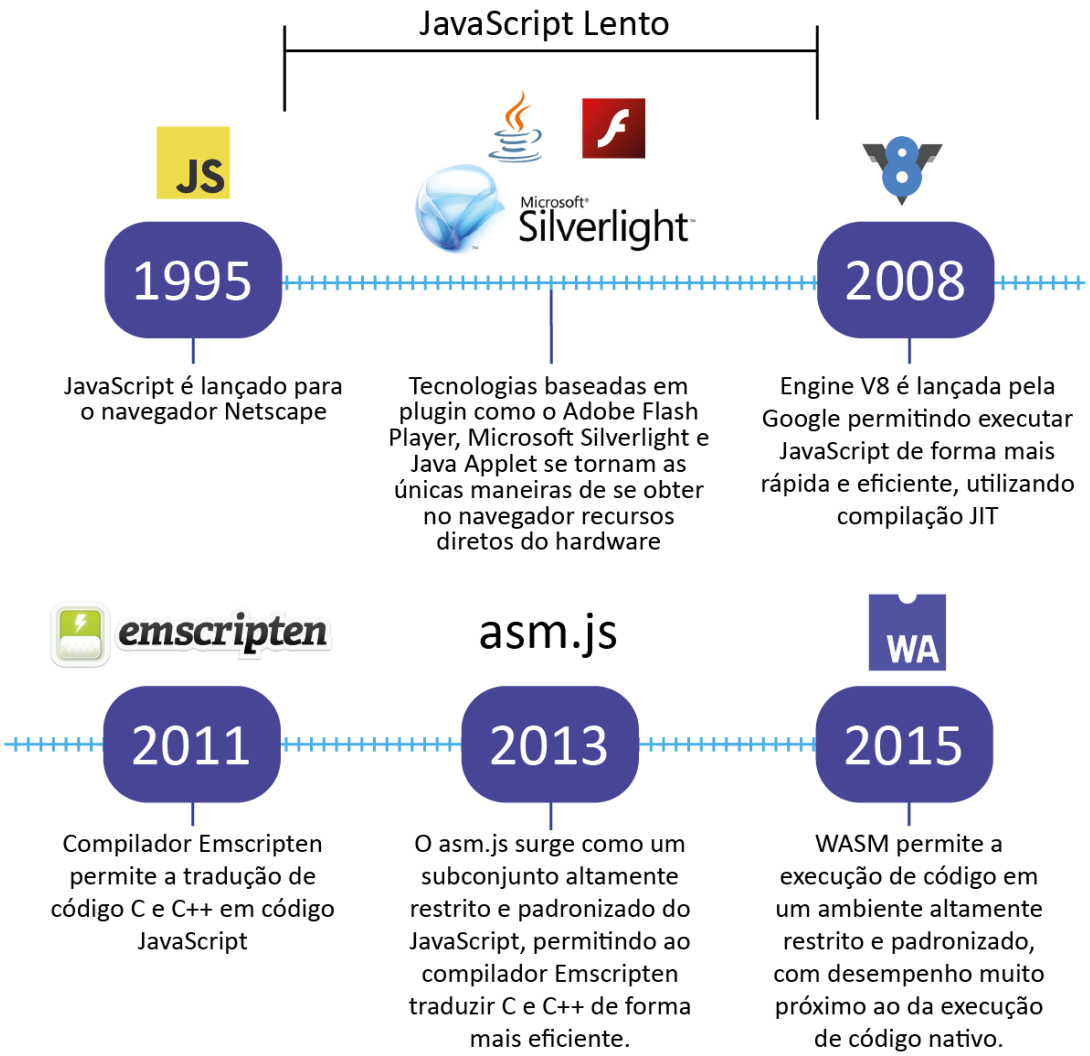

Figura 1: *Cronologia de tecnologias que antecederam o WebAssembly*

2.4 WEBASSEMBLY

Embora o JavaScript tenha passado por diversas melhorias em termos de desempenho graças às *engines* como V8 e SpiderMonkey, elas não foram suficientes para atender às demandas por aplicações web mais potentes. Como alternativa, surgiram projetos como o Emscripten e o Asm.JS, que visavam aproveitar os benefícios das linguagens C e C++ no

desenvolvimento de aplicações web. No entanto, como discutido anteriormente, a abordagem desses projetos se resumia a uma tradução otimizada das instruções dessas linguagens para o JavaScript, o que proporcionava portabilidade mas não melhorias significativas em termos de desempenho.

Com base nas necessidades do mercado, tornou-se evidente a necessidade de um modelo que permitisse a utilização de código nativo, sem perda de desempenho, segurança e sem que houvesse a necessidade de tradução prévia para o JavaScript. Em 2015, como solução para essa questão, empresas fornecedoras de navegadores, tais como Google, Mozilla, Apple e Microsoft, se uniram para criar o WebAssembly, visando um modelo universal e aberto para ser desenvolvido de forma colaborativa (Mozilla, 2023).

Logo, o World Wide Web Consortium (W3C), responsável por definir os padrões de desenvolvimento da World Wide Web, agregou o WebAssembly em sua lista de tecnologias e criou o WebAssembly Working Group dedicado a desenvolver e padronizar a tecnologia (W3C, 2022). Além do W3C, a Bytecode Alliance é uma organização importante no cenário do WebAssembly, formada por empresas como a Amazon, Mozilla, Intel e Red Hat. Essa fundação tem como objetivo difundir e popularizar o WASM entre os desenvolvedores, além de desenvolver tecnologias complementares (BYTECODE ALLIANCE, 2021).

2.4.1 Definição

A página oficial do WebAssembly define a tecnologia como uma especificação de formato de código binário para uma máquina virtual baseada em pilha (WebAssembly, 2023). No entanto, como observado por Rodrigues (2022), embora a nomenclatura sugira que o WASM se trate de uma linguagem Assembly, é incorreto defini-lo dessa forma.

As Linguagens Assembly são linguagens de baixo nível projetadas para corresponderem de forma direta às instruções executadas pelo processador. Isso reflete a existencia de uma linguagem Assembly única para cada arquitetura de processador. Como consequencia, instruções escritas em Assembly são altamente específicas e pouco portáveis.

Sendo o WebAssembly um formato de instrução binária e o Assembly uma representação simbólica da linguagem de máquina, a diferenciação das duas também ocorre quando

comparamos o ambiente de execução das duas tecnologias.O WASM foi projetado para ser executado por uma máquina virtual, a WebAssembly virtual machine (WASM VM), ao contrário do Assembly, que é destinado a uma arquitetura específica de processadores. Dessa maneira, a WASM VM possibilita que o código WebAssembly seja portátil, podendo ser executado em uma ampla variedade de plataformas e sistemas, sem que haja necessidade re recompilar ou modificar o código-fonte original. Essa funcionalidade é similar ao que acontece nas plataformas Java e .NET, que possuem suas próprias máquinas virtuais para possibilitar a portabilidade de código (HOFFMAN, 2019, p 4).

Neste contexto, para executar um programa em um ambiente WebAssembly, o código-fonte original precisa ser compilado para o formado de bytecode do WebAssembly. Conforme destacado por Smith (2022), ao contrário de outras abordagens onde as linguagens eram traduzidas para JavaScript, o código em WebAssembly é executado diretamente como código binário, o que oferece um desempenho superior.

Assim, é possível que linguagens que ofereçam suporte ao WASM utilizem seu formato de código binário como alvo de compilação. Para isso, é necessário utilizar ferramentas apropriadas que transformem o código-fonte original em bytecode WebAssembly (formato .wasm). Por exemplo, para linguagens como C e C++, é comum utilizar o Emscripten, enquanto para Rust, uma ferramenta popular é o wasm-pack, descrita na subseção 4.2.2.

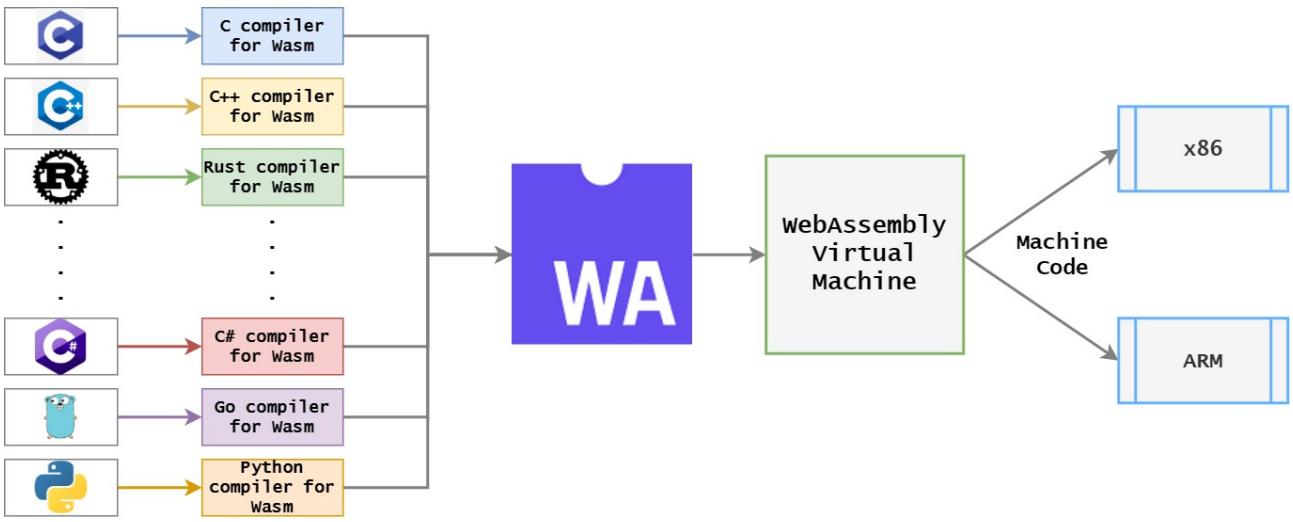

Figura 2: *WebAssembly como alvo de compilação (In: ENDEL A, 2020).*

De acordo com Rodrigues (2022, p 8), “Estas conversões para código WASM só são possíveis porque esta foi projetada para ser independente de qualquer linguagem em específico, não privilegiando nenhuma linguagem ou modelo de programação em específico.” A figura 2 ilustra como várias linguagens de programação podem ser compiladas para o formato bytecode do WebAssembly e, em seguida, serem executadas pela máquina virtual WASM compatível com as arquiteturas x86 e ARM.

2.4.2 WebAssembly Virtual Machine

A WebAssembly virtual machine, como mencionado anteriormente, é uma máquina virtual baseada em pilha. Essa estrutura de dados é uma parte fundamental da execução do código WebAssembly, pois os operandos são empilhados uns sobre os outros seguindo o padrão LIFO (Last In, First Out ou Último a Chegar, Primeiro a Sair). Esse modelo de computação implica que os dados só poderão ser manipulados estanto no topo da pilha, tornando impossível o acesso destes de forma dinâmica como em um *array* de endereços.

A arquitetura baseada em pilha traz algumas vantagens para a execução eficiente do código. Por exemplo, ela simplifica a implementação da máquina virtual, tornando-a mais compacta e rápida. Como as operações são realizadas apenas no topo da pilha, é possível utilizar uma quantidade menor de instruções para manipular os dados, o que contribui para um menor consumo de memória e uma execução mais ágil do código WebAssembly.

De maneira geral, as máquinas de pilhas podem ser manipuladas por meio de duas instruções básicas: o ‘PUSH’, que adiciona um valor ao topo da pilha, e o ‘POP’, que remove o valor do topo da pilha. No entanto, à medida que as linguagens Assembly evoluíram, a complexidade das operações exigiu a implementação de instruções mais específicas para lidar com diferentes tipos de dados e operações. Isso resulta em uma variedade de instruções personalizadas que podem ser otimizadas para atender a requisitos particulares de programação (BATTAGLINE, 2021, p 8).

No caso da WebAssembly VM, além das instruções de manipulação da pilha, há outras instruções para lidar com operações lógicas (como *add*, *sub*, *mul*, *div*, *min* e *max*) e aritméticas (como *select, if* e *else*). A máquina também possibilita a criação de variáveis locais, globais e constantes, com instruções específicas para lidar com cada um desses

tipos de dados. Essas instruções simplificam o processo de armazenamento, acesso e manipulação de informações, contribuindo para a eficiência e a segurança dos programas desenvolvidos utilizando a tecnologia WebAssembly.

De acordo com a WebAssembly Specification (2023), algumas das operações disponíveis incluem:

- *drop*: descarta o valor do topo da pilha, semelhante a instrução ‘PUSH’;
- *dup*: duplica o valor do topo da pilha;
- *get_local*: empilha o valor de uma variável local na pilha;
- *get_global*: empilha o valor de uma variável global na pilha;
- *set_local*: desempilha um valor e o armazena em uma variável local;
- *set_global*: desempilha um valor e o armazena em uma variável global;
- *tee_local*: desempilha um valor, o armazena em uma variável local e empilha o mesmo valor novamente.

Segundo Brito (2019), a adição dos números inteiros em um código WebAssembly requer a retirada dos valores do topo da pilha por meio da instrução *get_local*. Após a retirada dos valores, a instrução *add* deve ser utilizada para somá-los e, em seguida, o resultado deve ser retornado ao topo da pilha.

```
( func $soma (param $a i32) (param $b i32) (result i32)
        get_local $a
        get_local $b
        i32.add
)
```

Código 1: Função soma na notação WAT

Apesar de serem denominadas com outros nomes, essas instruções em WebAssembly são, na verdade, operações de ‘POP’ e ‘PUSH’. No trecho de código 1, é possível observar um exemplo da representação textual do WebAssembly, também conhecida como WAT (*WebAssembly Text Format*). Nesse exemplo, uma função é definida e tem como retorno a soma de dois números inteiros de 32 bits.

2.4.3 ISA

O formato de instrução binária do WebAssembly é definido graças à sua ISA (*Instruction Set Architecture*), que atua como uma interface de comunicação entre o software e o hardware, desempenhando funções diversas que vão desde a definição de formatos de instrução até o controle do uso de registradores e memória.

No caso específico do WebAssembly, o conjunto de instruções foi projetado para ser simples, com poucas instruções básicas, o que torna o formato binário compacto e fácil de interpretar. Entretanto, é importante ressaltar que uma ISA geralmente é projetada para viabilizar a execução em uma máquina física. Dessa forma, o WebAssembly é uma das primeiras tecnologias a utilizar essas instruções dentro de um ambiente virtualizado (Battagline, 2021, p 2).

Conforme mencionado por Zhou (2020), há diversas ISAs disponíveis no mercado, cada uma delas normalmente projetada para atender a uma arquitetura específica de processador. No entanto, o WebAssembly introduz uma ISA focada no uso em navegadores, tendo como o maior alvo suportar compilação JIT (*Just in Time*), enquanto atua como uma máquina virtual com contexto próprio de memória.

A ISA do WebAssembly é responsável pela definição da estrutura de endereçamento de memória virtual. No WASM, a memória é representada por uma grande matriz de bytes, que por padrão é protegida, impedindo acesso direto e não autorizado.

A memória no WebAssembly é acessível somente por meio de referências, mas o próprio WebAssembly não gerencia a memória automaticamente. O gerenciamento de memória é responsabilidade do ambiente em que o WASM é executado. Isso significa que os navegadores que desejam interagir com o WebAssembly devem atender às suas condições estritamente limitadas e controladas de acesso à memória, garantindo que nenhuma informação sensível continue protegida.

3. LINGUAGEM RUST

As linguagens de programação desempenham um papel crucial no desenvolvimento de software, capacitando os programadores a criar programas capazes de automatizar tarefas, resolver desafios e conceber novas aplicações. Na atualidade, o mercado oferece uma ampla gama de linguagens de programação, cada uma orientada para atender a um domínio específico. Algumas se destacam pela velocidade, outras pela sua adequação à web, enquanto outras focam na portabilidade.

Muitas linguagens de programação surgiram em resposta a desafios específicos de suas épocas. Aquelas que foram capazes de evoluir continuamente, se adaptar a novos problemas e se aplicar em diferentes contextos ao longo do tempo alcançaram popularidade duradoura e são mantidas até hoje. No entanto, a cada dia surgem desafios inéditos que muitas vezes as linguagens existentes não conseguem abordar de maneira ideal, dando continuidade ao surgimento de novas linguagens ou ao aprimoramento das já existentes.

Nesse contexto, surge em 2006 a linguagem Rust. Graydon Hoare, um desenvolvedor da Mozilla, deu início à sua criação como um projeto pessoal. Hoare tinha a ambição de criar uma linguagem tão sólida e eficiente quanto o C++, porém com um enfoque maior na segurança e na facilidade de uso. Ao longo do desenvolvimento, o Rust adquiriu recursos que Hoare ansiava há tempos por ver em outras linguagens com as quais havia trabalhado (BOEIRA, 2018, p 04).

Embora não contasse com expectativas elevadas de que seu projeto conquistasse grande sucesso comercial, Graydon Hoare entendia que introduzir uma nova linguagem no meio de tantas outras já estabelecidas não seria uma tarefa simples. Determinado a enfrentar esse obstáculo, Hoare elaborou um conjunto de ideias altamente originais, centradas principalmente em otimizar a velocidade da linguagem e reforçar a segurança no acesso à memória.

Essas ideias culminam nos conceitos de “lifetimes” (duração de referências), “borrowing” (empréstimo) e “ownership” (propriedade), os quais serão abordados nas seções subsequentes deste capítulo. Em conjunto, estes agregam ao Rust um ambiente de extrema segurança, que elimina a necessidade da linguagem utilizar um coletor de lixo (garbage collector). Mesmo para aqueles menos familiarizados com a manipulação de ponteiros e variáveis, os mecanismos integrados na linguagem gerenciam o acesso de

forma a preservar a integridade e a segurança, especialmente em comparação com sua linguagem inspiradora, o C++.

Até 2009, Graydon Hoare trabalhou de forma solitária no desenvolvimento do Rust. Nesse ano, a Mozilla, em busca de uma alternativa ao C++, descobriu o projeto e começou a patrocinar seu progresso. O Rust foi anunciado em 2010 e, embora a primeira versão estável tenha sido lançada somente em 2015 (CASTELLANI, 2018, p. 08), rapidamente conquistou uma vasta comunidade de usuários, chamando a atenção em diversas esferas. Sua abordagem segura demonstrou versatilidade, encontrando aplicação tanto em projetos de baixo nível, quanto em aplicações web.

3.1 LIFETIME

O sistema de *lifetime* (tempo de vida) é uma das características únicas da linguagem Rust. Esse mecanismo tem por função assegurar um escopo para o qual uma referência é válida, assim prevenindo erros de gerenciamento de memória como referências inválidas e problemas relacionados à concorrência. Ao associar cada referência a um tempo de vida específico, o Rust é capaz de rastrear a validade das variáveis e determinar quando seu uso se torna inseguro.

As *lifetimes* abrangem não somente variáveis, mas também todos os atributos essenciais da linguagem Rust, incluindo referências à *structs*, *enums*, parâmetros e retornos de funções. Conforme ressaltado por Boeira (2018, p 14), essa característica inerente à linguagem impede a ocorrência de erros inesperados durante a execução do programa, decorrentes do acesso a ponteiros nulos.

```
{
        let r;
        {
                let x = 5;
                r = &x;
        }
        println!("r: {}", r);
}
```

Código 2: Ao criar uma referência a uma variável de escopo interno e tentar utilizá-la fora desse escopo, o Rust emitirá um erro de compilação.

O trecho de código 2 foi apresentado pelos autores Klabnik e Nichols (2018, p. 193) com o intuito de ilustrar uma situação que destaca o uso de *lifetimes*. Inicialmente, uma variável ‘r’ é declarada sem nenhum valor específico. Em seguida, um novo escopo é iniciado, onde uma nova variável ‘x’ é criada e inicializada com o valor 5. Na próxima linha, a variável ‘r’ é atribuída como uma referência ‘&x’, ou seja, ela passa a apontar para o valor de ‘x’. O problema surge ao utilizar a referência ‘r’ fora do escopo onde ela foi originalmente definida.

Ao tentar imprimir o valor de ‘r’ fora do escopo onde ocorreu a referência para ‘x’ ocorre um erro de compilação referente ao tempo de vida inválido. Isso acontece porque a referência ‘r’ ainda está tentando apontar para a memória onde ‘x’ estava armazenada, mas essa memória já foi liberada após o término do escopo interno. Em decorrência, o compilador rustc detecta essa tentativa de acesso inválido e previne a compilação do código, sinalizando que o ciclo de vida da variável ‘x’ não se estendeu o bastante para abranger o escopo ao qual foi atribuída, como exemplificado na figura 3.

```
❯ cargo build
   Compiling rust_examples v0.1.0 (/home/pintari/Documents/Projetos/Rust/rust_examples)
error[E0597]: `x` does not live long enough
```

Figura 3: *Erro de compilação gerado pelo mecanismo de lifetimes*

É preciso destacar que, em linguagens como C e C++, que não têm um coletor de lixo, cabe ao programador cuidar da memória e dos ciclos de vida de ponteiros e referências. Portanto, se o trecho de código 2 fosse escrito e compilado em uma dessas linguagens, não haveria problemas de compilação.

3.2 BORROWING

O conceito de *borrowing* (empréstimo) é intrínseco ao mecanismo de *lifetimes* do Rust e é uma das características essenciais dessa linguagem para prevenir vazamentos de memória. Esse sistema estabelece diretrizes para o uso de referências por meio de ponteiros, levando em consideração as distinções entre variáveis mutáveis e imutáveis. O propósito é garantir a segurança e a integridade dos dados ao evitar acessos concorrentes e inseguros.

De acordo com Castellani (2018, p. 44), quando uma variável é criada utilizando a instrução ‘let’, ela é automaticamente definida como imutável. Isso significa que qualquer tentativa de alterar o valor dessa variável resultará em um erro de compilação. No entanto, se for necessário alterar o valor da variável durante a execução do código, podemos usar a palavra-chave ‘mut’ na declaração.

A imutabilidade é uma característica que de início pode parecer um pouco restritiva, mas traz benefícios significativos. Ela ajuda a promover o compartilhamento seguro de informações entre diferentes partes do código, o que é especialmente útil em cenários de concorrência, assegurando que múltiplas *threads* possam acessar os mesmos dados com um nível mais apurado de controle sobre o tipo de informação que está sendo modificada (BOEIRA, 2018, p 11).

Ao implementar a imutabilidade em conjunto com ponteiros, é necessário estar atento a algumas considerações. No Rust, ao possuir uma referência a um valor, essa referência normalmente é imutável. Entretanto, é possível empregar o mecanismo de *borrowing* para possibilitar mudanças no valor referenciado, desde que se sigam diretrizes específicas estabelecidas pelo Rust.

```
{
        let mut x = 10;
        let y = &mut x;

        *y = 20;

        println!("x: {}", x);
}
```

Código 3: Demonstração de uma ocorrência de *borrowing*

Um exemplo se encontra no trecho de código 3, onde a variável ‘x’ é declarada sendo mutável. Em seguida, a variável ‘y’ cria uma referência a ‘x’, porém, com a especificação de que essa referência é mutável. Dessa maneira, mesmo que a variável ‘y’ seja imutável, ainda é possível atribuir valores a ela que serão refletidos na variável ‘x’ sem problemas de compilação.

Segundo Johnson e Kaihlavirta (2017, p 131), o Rust decreta duas regras fundamentais para que o *borrowing* possa ocorrer. Essas regras, enraizadas na filosofia de segurança e

gerenciamento de memória da linguagem, desempenham um papel crucial na prevenção de erros comuns e na promoção de práticas de programação seguras.

- O empréstimo feito a partir de uma variável não pode ter um tempo de vida maior do que a variável original da qual foi feito o empréstimo. Em outras palavras, a referência emprestada deve permanecer válida apenas enquanto a variável original ainda estiver válida.
- Dois tipos de empréstimos não podem coexistir ao mesmo tempo. Isso significa que não é possível ter ao mesmo tempo uma referência imutável a um recurso e uma referência mutável ao mesmo recurso.

```
{
        let mut x = 42;
        let y = &x;
        let z = &mut x;

        println!("y: {}", y);
        println!("z: {}", z);
}
```

Código 4: Coexistência de empréstimos, causando um erro de compilação por conta do mecanismo de *borrowing*

O trecho de código 4 ilustra uma situação em que a variável ‘x’ é declarada como mutável e recebe o valor 42. Em seguida, esse valor é “emprestado” para a variável ‘y’. No entanto, a variável ‘z’ também obtém uma referência a ‘x’, e essa referência é mutável. Em linguagens de programação que permitem o uso de ponteiros sem restrições, isso não resultaria em um erro de compilação. Entretanto, no contexto do Rust, o código não seria compilado. Isso ocorre porque uma referência mutável e uma referência não mutável para a variável ‘x’ coexistem, violando assim uma das regras fundamentais relacionadas ao *borrowing*.

3.3 OWNERSHIP

Além de gerenciar o tempo de vida, e as referências de cada objeto, o Rust também efetua a administração da propriedade de cada elemento ao longo da execução do

programa. Esse é o mecanismo de *ownership*, cujo propósito é estabelecer diretrizes para a posse dos elementos.

Tal como acontece com as l*ifetimes*, o *ownership* é limitado ao escopo em que o elemento está inserido. Neste contexto, o mecanismo de *ownership* estabelece que cada objeto tenha um proprietário exclusivo. Quando um valor é movido para outra variável ou função, a propriedade desse objeto é transferida para o novo proprietário, que pode escolher devolvê-la ou não.

```
fn main() {
        let x: String =  String::from("Fema");
        print_name(x);
        println!("x: {}", x);
}

fn print_name(x: String) {
        println!("string: {}", x);
}
```

Código 5: Transferência de *ownership*

O trecho de código 5 pode parecer estar livre de erros à primeira vista, porém, devido às regras de propriedade *ownership*, resultaria em um erro de compilação. Isso ocorre porque a propriedade da variável 'x' é transferida ao ser passada como parâmetro para a função 'print_name'.

A tentativa de utilizar 'x' posteriormente no código resultaria em um erro, uma vez que sua propriedade já não mais existe no contexto original. De acordo com Koch (2015, p 5) “O escopo de um objeto é o bloco (delimitado com { }) em que o objeto foi criado. Quanto a execução sai desse bloco, o objeto é destruído automaticamente sem necessidade de um destrutor.” Nesse contexto, a transferência de propriedade exige que a variável original seja invalidada após ser cedida a uma função ou escopo diferente.

4. ESTUDO DE CASO

Neste capítulo, será apresentada uma demonstração prática do funcionamento do WebAssembly e sua capacidade de interoperabilidade com outras linguagens. Dessa forma serão utilizadas as linguagens Rust e Javascript em conjunto na implementação do jogo Snake, no Brasil popularmente conhecido como “Jogo da Cobrinha”. Com essa abordagem, o Rust será responsável por gerar um pacote binário no formato WASM, enquanto o Javascript cuidará de renderizar o conteúdo do jogo em uma página web.

4.1 BREVE HISTÓRICO E FUNCIONAMENTO DO JOGO SNAKE

De acordo com Benedetti (2021), o Snake teve sua origem na empresa Gremlin Interactive no ano de 1976, quando foi lançado como ‘Blockade’ para fliperamas. O jogo possui uma premissa simples, o que lhe rendeu várias adaptações para diferentes plataformas ao longo dos anos. Uma das mais famosas ocorreu através da Nokia, uma empresa finlandesa, que em 1997 comercializava o celular Nokia 6110, o qual vinha com o jogo Snake incluso em sua memória, entre outros atrativos da época.

Essa premissa simples vem do jogo conter apenas dois elementos principais: uma cobra e uma maçã. O jogador tem o objetivo de alimentar a cobra que, a cada maçã consumida aumenta de tamanho. O objetivo é fazer com que o tamanho da cobra cubra toda a extensidade do mapa sem que ela esbarre com o seu próprio corpo.

Em geral, a maioria das implementações do jogo inicia com seus dois elementos, a cobra e a maçã, posicionados aleatoriamente em um mapa bidimensional. A cobra tem a liberdade de movimentar-se tanto no eixo horizontal quanto no eixo vertical. Ao se consumir uma maçã, uma nova deve aparecer instantaneamente em uma posição aleatória do mapa, idealmente sem sobrepor-se ao corpo da cobra.

O mapa é dividido em células, representando cada índice que pode ser ocupado tanto pela cobra quanto por uma maçã. Completar todas as células do mapa com a cobra resulta na vitória do jogo. No entanto, à medida que a cobra ocupa mais células, é preciso maior habilidade por parte do jogador para manipular o corpo da cobra. Caso a cobra esbarre em si mesma durante o movimento, o jogo é considerado como perdido.

4.2 BIBLIOTECAS E FERRAMENTAS UTILIZADAS

Wozniewicz (2019) descreve o uso de bibliotecas como uma analogia interessante: “Uma biblioteca é como ir à sua loja de móveis favorita. Você já tem uma casa, mas precisa de ajuda com os móveis. Não é sua intenção fazer uma mesa do zero.”

4.2.1 Wasm-Bindgen

A biblioteca ‘wasm-bindgen’ é um utilitário que facilita a comunicação entre o Rust e o JavaScript, permitindo que funções sejam disponibilizadas de uma para outra por meio do WebAssembly. No Rust, para tornar uma função disponível para uso no JavaScript, basta utilizar a notação ‘#[wasm_bindgen]’ antes da sua declaração. Da mesma forma, é possível utilizar funções do JavaScript no Rust usando a instrução *extern*. (MDN WEB DOCS, 2023).

O trecho de código 6 exemplifica o uso do Rust em conjunto com a função *alert* do Javascript. Essa função, quando aplicada a uma página web, tem a finalidade de exibir uma caixa de diálogo *pop-up* no navegador do usuário contendo uma mensagem especificada. O termo *extern* é empregado para indicar que a função subsequente será externa, ou seja, não é definida em Rust, mas sim em outro ambiente, neste caso, o JavaScript.

```
{
	#[wasm_bindgen]
	extern {
		pub fn alert(message: &str);
	}

	#[wasm_bindgen]
	pub fn run_alert(message: &str) {
		alert(message);
	}
}
```

Código 6: Função nativa do Javascript sendo utilizada dentro de um programa Rust

A função *run_alert* também é decorada com a notação *‘*#[wasm_bindgen]’, mas desta vez para indicar que esta será acessível a partir do JavaScript. Dessa forma, por meio de um arquivo JS vinculado a uma página HTML, é possível invocar a função *run_alert*, passando uma *string* como parâmetro e, assim, exibindo uma caixa de diálogo com a

mensagem fornecida. É importante destacar que essa interação só é viabilizada ao utilizar o Rust como compilador-alvo para o WebAssembly.

4.2.2 Wasm-Pack

O wasm-pack é uma biblioteca que fornece um conjunto de instruções via linha de comando que simplificam a compilação direta do Rust para o formato binário do WebAssembly. Na realidade, o wasm-pack produz como produto um módulo completo que pode ser facilmente importado por uma gama de ambientes.

Ao utilizar o comando ‘wasm-pack build’, o compilador do rustc é direcionado para compilar todo o código Rust em módulo (ou pacote), denominado ‘pkg’. Esse módulo, além do binário .wasm contém também outros artefatos que atuarão como facilitadores na importação e utilização do WebAssembly pelo ambiente alvo (Nellaiyapen, 2021).

Sendo assim, para garantir uma utilização completa do WebAssembly em um ambiente específico, é necessário definir o destino do pacote por meio do parâmetro ‘--target’ disponível no comando de *build* do wasm-pack.

Neste trabalho, o foco é utilizar o módulo WebAssembly em um ambiente web. Portanto, ao executar o comando ‘wasm-pack build --target web’, será gerado um pacote otimizado para navegadores, facilitando sua integração em projetos Node.js que utilizam a biblioteca Webpack ou outras ferramentas de empacotamento.

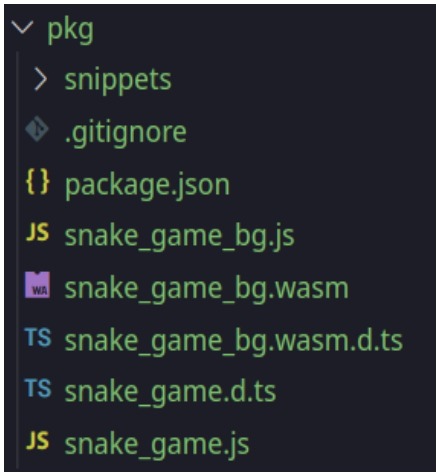

Figura 4: *Organização de um módulo WebAssembly voltado para ambientes web*

A Figura 4 ilustra um exemplo de um módulo WebAssembly direcionado para ambientes web, gerado através da biblioteca wasm-pack. Esse módulo inclui o binário .wasm, bem como arquivos JavaScript e TypeScript que atuarão como facilitadores na integração do WebAssembly ao projeto. O arquivo ‘package.json’ é essencial, pois identifica e gerência o pacote, permitindo a sua publicação no NPM (*Node Package Manager*).

4.2.3 Webpack

O Webpack é uma ferramenta JavaScript construída e distribuída para projetos Node.js, utilizada para empacotar recursos estáticos em projetos web. Além disso, o Webpack também possui um servidor embutido que facilita o desenvolvimento local, permitindo visualizar a aplicação em tempo real durante o desenvolvimento.

O Webpack tem como principal função agrupar recursos estáticos e suas dependências, como módulos, arquivos HTML, CSS e JS/TS, em um único pacote chamado *bundle*, tal qual é ilustrado pela figura 5. Essa abordagem torna o carregamento da aplicação mais eficiente no navegador, pois reduz a quantidade de requisições HTTP necessárias o *download* de cada artefato.

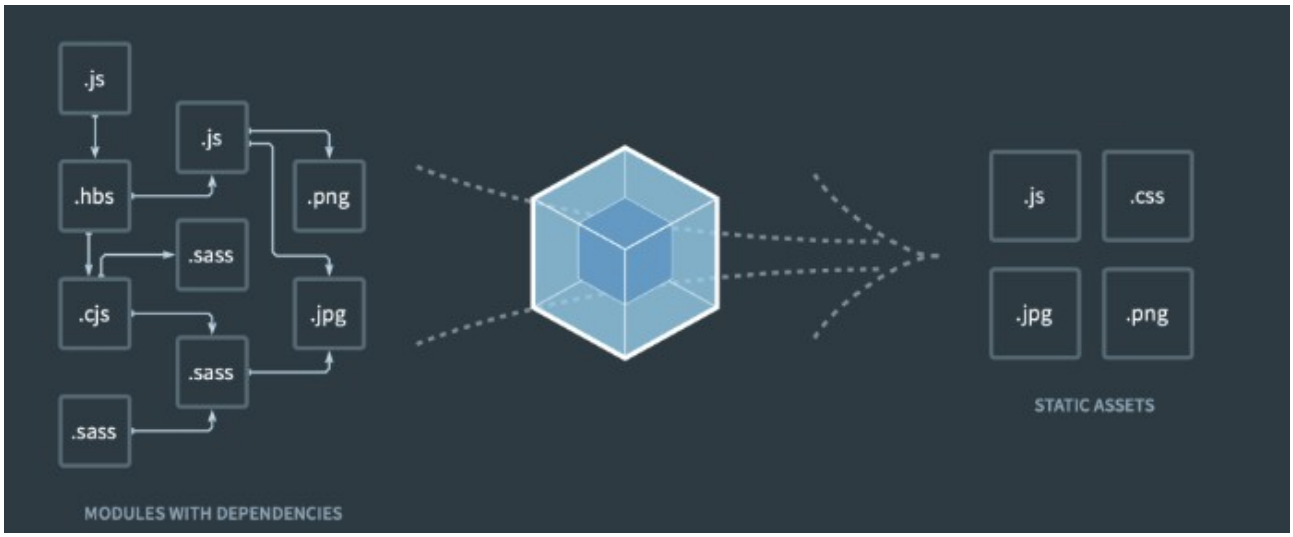

Figura 5: *Demonstração do bundle gerado pela ferramenta Webpack (In: ENDEL B, 2020).*

O *bundle* gerado pelo Webpack não só melhora a organização e manutenção do projeto, mas também oferece um melhor gerenciamento dos módulos utilizados. Esse aspecto é especialmente relevante para este trabalho, uma vez que demandará comunicação constante com o módulo WebAssembly gerado pela ferramenta wasm-pack.

De acordo com Stender (2020) “O Webpack vem para nos ajudar oferecendo soluções para gerenciar seu projeto, lhe permitindo manter o uso de arquivos desacoplados e os empacotar em um único arquivo no final tornando mais fácil a inclusão em sua página HTML.”

4.3 ESTRUTURA BASE DO PROJETO

O desenvolvimento do jogo Snake adota uma estrutura de diretórios dividida de acordo com a lógica (escrita em Rust) e a renderização do jogo (escrita em Javascript), conforme é demonstrado na figura 6.

Figura 6: *Organização de arquivos e diretórios para a implementação do jogo Snake*

O diretório ‘web’ é dedicado ao desenvolvimento web e abrange todos os elementos relacionados a essa parte do projeto. Dentro dele, encontramos a estrutura característica de um projeto Node.js, com destaque para dois arquivos principais: o ‘index.html’, que define a estrutura dos elementos na página, e o ‘index.js’, responsável por toda a lógica de renderização do mapa e dos elementos do jogo. O diretório ‘utils’ será utilizado para armazenar funções utilitárias em Javascript.

Dentro do diretório ‘src’, está concentrada a lógica do jogo relacionada ao *backend*. O arquivo ‘lib.rs’ contém as structs e funções que englobam toda a lógica do jogo, desde o movimento e evolução da cobra até a geração das maçãs e o controle do tamanho do mapa.

Outro diretório relevante é o ‘pkg’. Esse diretório, como citado na subseção 4.2.2 contém o módulo WASM a ser importado pelo arquivo ‘index.js’.

Os arquivos dos diretórios citados implicam diretamente no processo de execução do projeto. Conforme demonstra a figura 7, são um total de quatro estágios até que o projeto possa ser executado.

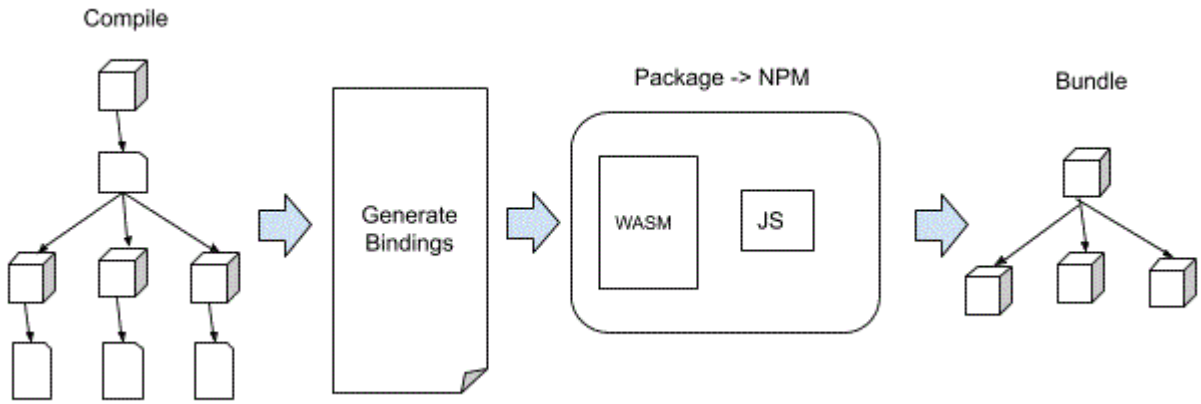

Figura 7: *Processo de execução em um projeto web utilizando WebAssembly (In: ENDEL C, 2021).*

O arquivo ‘lib.rs’ é compilado usando a biblioteca wasm-pack para gerar o módulo WebAssembly, representando os dois primeiros estágios. Em seguida, o módulo WASM é importado pelo ‘index.js’ e servido como bundle pelo Webpack, concluindo as duas últimas etapas do processo.

O projeto inclui ainda arquivos de configuração específicos para bibliotecas e gerenciadores de pacotes. Dois arquivos relevantes são o “Cargo.toml”, usado pelo gerenciador Cargo, e o “package.json”, essencial para o gerenciador NPM. Ambos são fundamentais na definição das dependências e configurações do projeto em suas respectivas linguagens, Rust e JavaScript.

Essa organização proporciona uma clara separação das responsabilidades entre os diversos aspectos do desenvolvimento do jogo Snake. Isso garante uma estrutura coesa, facilitando a manutenção e expansão do projeto.

4.4 DESENVOLVIMENTO

Nesta seção, será apresentado o desenvolvimento completo do jogo Snake, abrangendo cada componente do jogo, desde a renderização até a lógica por trás de cada elemento. O objetivo é proporcionar uma visão abrangente de como as linguagens Rust e Javascript se complementam para interagir com o WebAssembly.

4.4.1 Mapa

O mapa é uma matriz quadrada, possuindo o mesmo número de linhas e colunas. No Rust, essa matriz é representada na *struct* 'World', a qual possui os atributos 'width', que representa o número total de linhas, e 'size', correspondente ao tamanho real da matriz.

```
#[wasm_bindgen]
pub struct World {
        width: usize,
        size: usize,
}

#[wasm_bindgen]
impl World {
       pub fn new (width: usize) -> World {
                let size: usize = width * width;
                World {
                        width,
                        size
                }
        }
        pub fn get_width(&self) -> usize {
                self.width
        }
}
```

Código 7: Declaração e implementação da *struct* 'World'

Esse tamanho é obtido ao multiplicar o valor de 'width' por si mesmo, conforme demonstrado no trecho de código 7, localizado no método construtor da *struct*. No Javascript esse atributo será muito utilizado, por isto a função ‘get_width’ terá o encargo de retornar esse valor.

A *tag* do elemento canvas será definida no arquivo ‘index.html’, conforme é demonstrado no trecho de código 8. Algumas estilizações foram aplicadas para ajustar o tamanho e centralização do elemento na tela. É relevante destacar o atributo ‘id’, que permite que o arquivo JavaScript faça referência direta a esse componente.

```
        <style>
                .content-wrapper {
                        top: 8px;
                        left: 0;
                        width: 100%;
                        height: 100%;
                        position: absolute;
                        display: flex;
                        align-items: center;
                        justify-content: center;
                        flex-direction: column;
                }
        </style>
</head>
<body>
        <div class="content-wrapper">
                <canvas id="snake-canvas"></canvas>
        </div>
```

Código 8: Declaração e estilização do elemento canvas

Para utilizar as *structs* provenientes do Rust no JavaScript, é necessário importá-las do módulo WASM, acompanhado da função ‘init’. Essa função é responsável por iniciar a execução do código WebAssembly dentro do contexto JS, conforme demonstra o trecho de código 9.

Através da constante ‘WORLD_WIDTH’ conseguimos definir qual será o tamanho da matriz, considerando que, para o valor 10, a matriz terá um total de 100 células. Esse valor é passado então para a constante ‘world’, que possui referencia direta para o construtor da *struct* em Rust. As demais constantes são utilizadas para definir estilizações ao componente canvas, como o tamanho das células do mapa e a cor de cada linha.

```
import init, { World } from "../pkg";

init().then(wasm => {
        const CELL_SIZE = 50;
        const LINE_COLOR = "#0299";
        const LINE_WIDTH = 1.7;
        const WORLD_WIDTH = 10;

        const world = World.new(WORLD_WIDTH);
        const worldWidth = world.get_witdh();

        const canvas = document.getElementById("snake-canvas");

        canvas.height = worldWidth * CELL_SIZE;
        canvas.width = worldWidth * CELL_SIZE;

        const ctx = canvas.getContext("2d");
```

Código 9: Declaração de constantes e variáveis no Javascript a partir da importação da *struct* 'World'

O canvas no JavaScript é um objeto que suporta uma série de parâmetros. Para que o mapa desenhado na tela, é necessário definir alguns atributos importantes, como altura, largura, referência HTML e contexto. A variável ‘ctx’ guardará o objeto com esses atributos preenchidos de forma a ser utilizado na função de desenho do mapa.

```
function drawWorld() {
        ctx.clearRect(0,0, canvas.width, canvas.heigth);
        ctx.strokeStyle = LINE_COLOR;
        ctx.lineWidth = LINE_WIDTH;
        ctx.beginPath();

        // Desenha linhas horizontais
        for (let x = 0; x < worldWidth + 1; x++) {
                ctx.moveTo(CELL_SIZE * x, 0);
                ctx.lineTo(CELL_SIZE * x, worldWidth * CELL_SIZE);
        }

        // Desenha linhas verticais
        for (let y = 0; y < worldWidth + 1; y++) {
                ctx.moveTo(CELL_SIZE * y);
                ctx.lineTo(worldWidth * CELL_SIZE, CELL_SIZE * y);
        }
        ctx.stroke();
}
```

Código 10: Implementação da função ''drawWorld' que representa a construção do mapa do jogo

O próximo passo é desenhar o mapa na tela por meio da função ‘drawWorld’, denotada no trecho de código 10. Inicialmente, essa função limpa o conteúdo anteriormente desenhado usando o método ‘clearRect’, o que será especialmente útil para atualizar a posição dos elementos na matriz.

A seguir, o método ‘beginPath’ é utilizado para iniciar um novo desenho no canvas. Em seguida, os dois *loops* de repetição são responsáveis por desenhar as linhas horizontais e verticais, respectivamente, com base no tamanho das células e largura do mapa. Este processo de desenho é fundamental para a criação da representação visual do ambiente de jogo.

As linhas horizontais e verticais delineiam a estrutura do mapa, estabelecendo as fronteiras das células individuais. Isso proporciona uma grade que servirá como referência visual para a localização e movimentação da cobra.

Após o desenho dos elementos no Canvas, a matriz que representa o mundo do jogo é exibida usando o método ‘stroke’. A matriz ilustrada na figura 8 possui um tamanho de 6x6, mas este valor pode ser alterado modificando a constante ‘WORLD_WIDTH’.

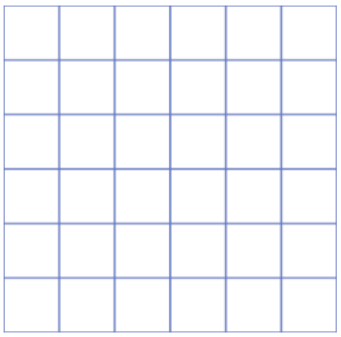

Figura 8: *Representação do mapa do jogo Snake gerado a partir da função* ‘drawWorld()’

4.4.2 Cobra

Na implementação, a estrutura base que define as características da cobra é definida através de duas *structs* ‘SnakeCell’ e ‘Snake’. Por momento, ambas possuem um único

atributo, ‘SnakeCell’ representa o tamanho em células da cobra, enquanto que ‘Snake’ possuí o atributo ‘body’, que nada mais é um vetor que armazena todos os índices ocupados pela cobra no mapa.

Além disso, o método construtor recebe dois parâmetros: ‘spawn’ e ‘size’. O primeiro indica em qual índice a cobra aparecerá no mapa, e o segundo indica qual será a sua extensão inicial. Dependendo dos valores desses parâmetros, o laço de repetição preencherá o vetor ‘body’ com os índices ocupados pela cobra, com base no seu ‘spawn’ e no tamanho ‘size’. O trecho de código 11 demonstra a implementação das duas *structs*.

```
pub struct SnakeCell(usize);

pub struct Snake {
	body: Vec<SnakeCell>
}

impl Snake {
	fn new(spawn: usize, size: usize) -> Snake {
		let mut body: Vec<SnakeCell> = vec!();

		for i in 0..size {
			body.push(SnakeCell(spawn - i));
		}

		Snake {
			body,
		}
	}
}
```

Código 11: Declaração e implementação da *struct* 'Snake'

A cobra é um dos principais elementos do mapa do jogo, sendo assim teremos uma instância sua dentro da *struct* ‘World’. Além disso, foram criados métodos *get* para que os atributos da cobra sejam acessados por outros métodos do Rust ou por funções Javascript. Esses métodos, juntamente a instância do objeto ‘Snake’, estão definidos no trecho de código 12.

```
        let snake: Snake = Snake::new(1, 1);

        World {
                width,
                size,
                snake
        }
}

pub fn get_snake_head(&self) -> usize {
        self.snake.body[0].0
}

pub fn get_snake_cells(&self) -> *const SnakeCell {
        self.snake.body.as_ptr()
}

pub fn get_snake_lenght(&self) -> usize {
        self.snake.body.len()
}
```

Código 12: Métodos *get* implementados para a *struct* 'Snake'

Assim como na função ‘drawWorld’, também será criada uma função equivalente ‘drawSnake’ em Javascript. No entanto, há um pequeno obstáculo em relação à memória para se obter determinados atributos da cobra no JS.

Para renderizar a cobra no mapa, é necessário utilizar um método *get* no JavaScript que retorne um vetor contendo todos os índices ocupados pela cobra. No Rust, esse método já existe e é chamado de ‘get_snake_cells’. No entanto, quando compilado para WASM, esse método cria um ponteiro que o JavaScript não consegue acessar diretamente.

Para possibilitar o acesso aos atributos da cobra no JavaScript, é preciso criar uma “visualização” do *buffer* de memória como um array de 32 bits (‘Uint32Array’). Assim, foi criada a constante ‘snakeCells’, que recebe o atributo ‘wasm.memory.buffer’.

O atributo ‘wasm.memory.buffer’ é uma referência direta à memória alocada pelo WebAssembly, permitindo que a função ‘get_snake_cells’ retorne o resultado esperado para a constante ‘snakeCells’. Isso é fundamental para possibilitar a renderização correta da cobra no mapa.

```
function drawSnake() {
	const snakeCells = new Uint32Array(
		wasm.memory.buffer,
		world.get_snake_cells(),
		world.get_snake_lenght()
	)

	snakeCells.forEach((cell, i) => {
		const col = cell % worldWidth;
		const row = Math.floor(cell / worldWidth);

		ctx.fillStyle = i == 0 ? "#4CBD71" : "4CBD49";

		ctx.beginPath();
		ctx.fillRect(
			col * CELL_SIZE,
			row * CELL_SIZE,
			CELL_SIZE,
			CELL_SIZE
		);
	})
	ctx.stroke();
}
```

Código 13: Implementação da função 'drawSnake' em Javascript

Por fim, como pode ser notado no trecho de código 13 a função segue um laço de repetição que irá pintar no mapa cada célula da cobra com base no valor de ‘snakeCells’. É importante observar que a função ‘fillStyle’ é utilizada exclusivamente para colorir cada célula da cobra de acordo com o seu índice. A extremidade frontal da cobra é pintada com uma cor específica, enquanto o restante do corpo é pintado com outra cor.

4.4.3 Atualização Do mapa

Atualmente, estamos desenhando apenas elementos estáticos no mapa, mas na próxima etapa, implementaremos a movimentação da Cobra. Portanto, precisamos de uma função que redesenhe o conteúdo do mapa para refletir o movimento desejado. Isso ficará na responsabilidade da função ‘update’ que será utilizada para atualizar o estado do jogo, agendando a próxima atualização após um intervalo de tempo especificado dando a noção de movimento.

A função ‘update’ terá como elemento principal o método assíncrono ‘setTimeout’ do JavaScript, que permite o agendamento da execução de um bloco de código após um

intervalo de tempo específico. No contexto do jogo, esse intervalo foi definido como ‘10000 / FPS’, tendo em vista que a constante ‘FPS’ é previamente estabelecida na função ‘init’.

FPS é a abreviação de “Frames por Segundo” e representa a frequência com que as imagens são exibidas na tela a cada segundo. Em termos simples, quanto maior o valor de ‘FPS’, mais quadros de imagem serão mostrados por segundo. Isso significa que, ao utilizar a função ‘setTimeout’, haverá mais atualizações gráficas para os elementos desenhados. Assim, o valor da constante ‘FPS’ será o responsável por definir a velocidade do movimento da cobra dentro do jogo, tornando-o mais rápido ou mais lento.

Utilizando o método ‘setTimeout’, a função ‘update’ a cada ciclo de atualização, limpará o conteúdo que foi previamente desenhado e em seguida chamará a função ‘draw’. Esta nada mais é do que o conjunto dos métodos ‘drawWorld’ e ‘drawSnake’ encapsulados em um único lugar, conforme demonstra o trecho de código 14.

```
function draw() {
	drawWorld();
	drawSnake();
}

function update() {
	setTimeout(() => {
		ctx.clearRect(0, 0, canvas.width, canvas.height);
		draw();
		requestAnimationFrame(update)
	}, 10000 / FPS)
}

update();
```

Código 14: Implementação das funções 'draw' e 'update'

A partir disso, o ciclo de atualização é repetido continuamente por meio da função ‘requestAnimationFrame’, garantindo que os elementos do jogo sejam constantemente atualizados. Essa prática é de importância crucial para assegurar que ações do jogo, como o surgimento da maçã ou o movimento da cobra, aconteçam conforme planejado, e sejam exibidas na tela com a taxa de quadros desejada.

4.4.4 Movimentação

A movimentação, sendo a parte mais complexa da implementação, recebeu uma atenção especial neste trabalho. Na figura 9, é possível visualizar todas as funções envolvidas nesta etapa de desenvolvimento. Essa abordagem proporciona uma compreensão mais aprofundada de como a cobra se move e como seus atributos são atualizados ao longo do jogo.

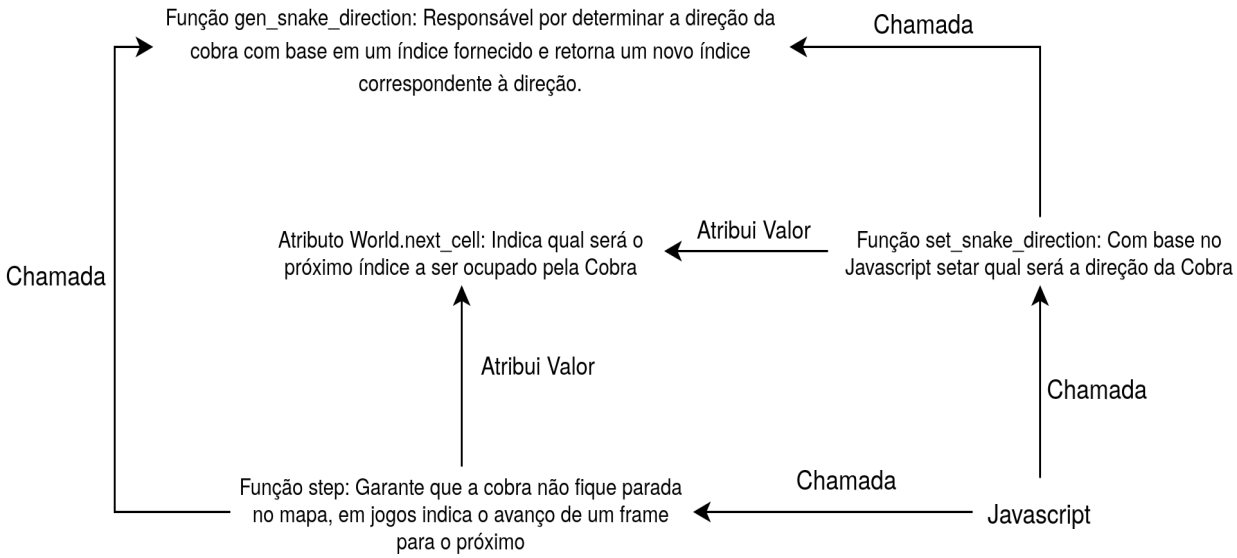

Figura 9: *Representação de todas as funções e atributos que compõe o movimento da cobra no jogo*

4.4.4.1 Enumeração de Direções

Para permitir que a cobra se mova no mapa, precisamos definir as quais são as direções possíveis. Para representar isso em código Rust, criaremos uma enumeração com valores para cada direção, conforme contido no trecho de código 15.

```
pub enum Direction {
        Up,
        Right,
        Down,
        Left
}
```

Código 15: Enumeração contendo as quatro direções possíveis para a cobra no jogo

Essa enumeração não apenas proporciona clareza na manipulação das direções, mas também permite que o código seja mais legível ao descrever o sentido da cobra.

Além disso, na *struct* ‘Snake’, será incluída uma referência a esse *enum*, assim como em seu método construtor, conforme demonstrado no trecho de código 16. Essa associação entre a *struct* e o *enum* ‘Direction’ permitirá que a cobra seja instanciada com uma direção inicial definida, proporcionando um ponto de partida para seu movimento no início do jogo.

```
#[derive(PartialEq, Clone, Copy)]
pub struct SnakeCell(usize);
pub struct Snake {
        body: Vec<SnakeCell>,
        direction: Direction
}

impl Snake {
        fn new(spawn: usize, size: usize) -> Snake {
                let mut body: Vec<SnakeCell> = vec!();
                for i in 0..size { ...

                Snake {
                        body,
                        direction: Direction::Right,
                }
        }
}
```

Código 16: Adição do atributo 'direction' na *struct* 'Snake'

4.4.4.2 Atributo next_cell e Função get_gen_next_cell

Na *struct* ‘World’, foi adicionado um novo atributo denominado ‘next_cell’, com a finalidade de armazenar o índice da próxima célula que a cobra ocupará no tabuleiro do jogo, antecipando seu próximo movimento. A escolha de utilizar o tipo ‘Option<SnakeCell>’

indica que essa próxima célula pode ser opcional, ou seja, pode não existir em alguns momentos.

Essa abordagem é relevante no jogo, considerando que a cobra está constantemente em movimento, e a próxima célula a ser ocupada depende de sua direção e posição atual. Em algumas situações, a próxima célula pode estar vazia, como quando a cobra se move para uma área sem maçã ou obstáculos (seu próprio corpo).

A função ‘gen_next_snake_cell’ será responsável por definir o valor do atributo ‘next_cell’, considerando a direção atual da cobra. Nessa função, toda a lógica para determinar a próxima célula da cobra e a mudança de direção será implementada. Além disso, a função ‘gen_next_snake_cell’ será utilizada pela função ‘set_snake_direction’, a qual será detalhada no item 4.4.4.4.

A função ‘gen_next_snake_cell’ recebe a enumeração de direções como parâmetro e obtém o índice da célula que representa a cabeça da cobra através do método ‘get_snake_head’. Em seguida, esse índice é utilizado para definir o valor da variável ‘row’, que representa a linha atual em que a cobra está presente. A seguir, o bloco *match* do Rust é utilizado para, com base na enumeração de direções, determinar o movimento e calcular a próxima célula que será ocupada pela cobra.

```
fn gen_next_snake_cell(&self, direction: &Direction) -> SnakeCell {
        let snake_index: usize = self.get_snake_head()
        let row: usize = snake_index / self.width;

        return match direction {
                Direction::Right => { ... },
                Direction::Left => { ... },
                Direction::Up => { ... },
                Direction::Down => {
                        let treshold = snake_index + ((self.width - row) * self.width);
                        if snake_index + self.width == treshold {
                                SnakeCell(treshold - ((row + 1) * self.width))
                        } else {
                                SnakeCell(snake_index + self.width)
                        }
                },
        };
}
```

Código 17: Implementação do método 'gen_next_snake_cell'

A função calcula o ‘threshold’ que representa a posição limite da linha ou coluna da cobra, dependendo da direção. Essa posição indica o final de uma linha ou coluna no mapa. No trecho de código 17, é ilustrado o cálculo do ‘threshold’ correspondente à direção para baixo, que é obtido somando um à linha atual e multiplicando pelo tamanho da largura do mapa.

É importante destacar que o cálculo do ‘threshold’ é realizado de forma similar para todas as outras direções do mapa, variando apenas nos aspectos específicos de cada direção. Devido à extensão do método, as demais correspondências do bloco *match* foram omitidas da figura.

Se o índice da cabeça da cobra mais um for igual ao ‘threshold’, isso indica que a cobra deve “atravessar” para a próxima linha do mapa. Caso contrário, a próxima célula será simplesmente a próxima célula livre. Esse comportamento é exemplificado na figura 10, onde a cobra alcança a última coluna de uma linha e reaparece no primeiro índice desta.

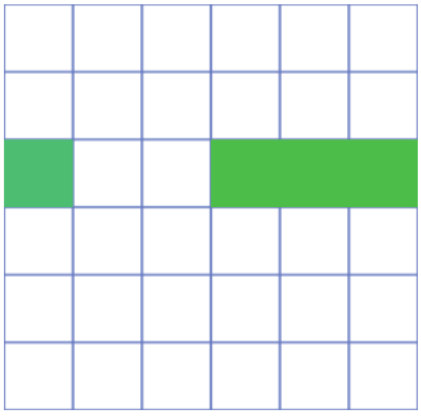

Figura 10: *Comportamento da cobra ao alcançar uma borda do mapa: transição para o lado oposto*

Esse aspecto pode variar dependendo da implementação do jogo Snake. Em algumas versões, ultrapassar o limite do mapa pode resultar em um caso de fim de jogo, mas não é o caso da implementação deste trabalho.

4.4.4.3 Função step

A função ‘Step’ desempenha um papel crucial no jogo, garantindo o movimento contínuo dos elementos relacionados no mapa. Ela trabalha em conjunto com a função update implementada no JavaScript para avançar o jogo quadro a quadro. Essa abordagem é comum em jogos, onde o *step* representa o progresso para o próximo estado do jogo.

O método começa criando uma cópia temporária do vetor que armazena os índices do corpo da cobra, com o propósito de preservar o estado atual das posições quando o método é chamado. Em seguida, o bloco *match* é utilizado para modificar o fluxo de execução do método dependendo do estado do atributo ‘next_cell’. Esse atributo é do tipo ‘Option<SnakeCell>’, o que significa que ele pode ou não conter um índice.

Caso o valor do atributo ‘next_cell’ seja diferente de *None*, a cobra receberá um novo índice com base na direção em que estiver seguindo através do método ‘gen_next_snake_cell’. Caso o atributo já tenha sido preenchido, quer dizer que a cobra já tem um sentido definido a partir de uma determinada direção, neste caso, a próxima célula a ser preenchida será o próprio valor de ‘next_cell’. Após isto, o valor deste atributo é marcado como *None*.

Considerando que o método ‘step’ será executado em um loop, a alternância entre as duas opções do bloco match assegurará o movimento contínuo da cobra no mapa. Essa execução será a partir da função ‘update’ definida em Javascript.

Após atualizar a posição da cabeça da cobra, o método continua atualizando as posições das células restantes no corpo da cobra. Para isso, utiliza-se um laço de repetição que percorre o tamanho total da cobra.

```
pub fn step(&mut self) {
	let temp: Vec<SnakeCell> = self.snake.body.clone();
	match self.next_cell {
		Some(cell) => {
			self.snake.body[0] = cell;
			self.next_cell = None;
		},
		None => {
			self.snake.body[0] = self.gen_next_snake_cell(&self.snake.direction);
		}
	}
	let len: usize = self.snake.body.len();
	for i in 1..len {
		self.snake.body[i] = SnakeCell(temp[i - 1].0)
	}
}
```

Código 18: Implementação do método 'step'

Com base na cópia anterior dos índices, cada célula do corpo da cobra é atualizada com a posição da célula anterior, movendo assim a cobra uma posição à frente. O trecho de código 18 ilustra a definição completa do método.

4.4.4.4 Função set_snake_direction

A função ‘set_snake_direction’ permite que, através do JavaScript, seja possível mudar a direção da cobra com base em um evento de pressionamento de tecla. Essa função é um elo crucial entre a lógica do jogo implementada em Rust e a interação do jogador através do JavaScript. Através dos eventos de tecla, o jogador poderá controlar a direção da cobra, e essas ações serão transmitidas para o Rust, que atualizará a direção da cobra conforme a lógica dos demais métodos responsáveis pela direção descritos nesta sessão.

Como demonstra o trecho de código 19, a implementação deste método é bastante simples. Ele recebe um parâmetro do tipo ‘Direction’, que é uma enumeração definida contendo as quatro possíveis direções. Com base nesse parâmetro, o método atualiza o valor do atributo ‘next_cell’ e, em seguida, altera o atributo ‘snake.direction’ para a direção fornecida como parâmetro. Esse processo possibilita que a cobra mude sua direção de acordo com as escolhas do jogador.

```
pub fn set_snake_direction(&mut self, direction: Direction) {
        let next_cell: SnakeCell = self.gen_next_snake_cell(&direction);

        if self.get_snake_lenght() > 1 && self.snake.body[1].0 == next_cell.0 {
                return;
        }

        self.next_cell = Some(next_cell);
        self.snake.direction = direction;
}
```

Código 19: Implementação do método 'set_snake_direction'

Entretanto, antes que quaisquer alterações sejam persistidas, a função verifica uma condição importante antes de atualizar a direção da cobra. Ela avalia se o comprimento da cobra é maior que um e se a próxima célula gerada é igual à segunda célula do corpo da cobra. Essa verificação é realizada para evitar que a cobra volte abruptamente na direção oposta à sua atual. Caso essa condição seja verdadeira, a função é retornada sem fazer nenhuma alteração.

Em JavaScript, ‘document.addEventListener’ é uma função utilizada para registrar eventos que ocorrem no documento HTML. Ele permite que você adicione um *listener* (ouvinte) para esperar por determinado evento e realizar ações em resposta. Neste contexto, para capturar o pressionamento de teclas, o evento utilizado será o ‘keydown’.

Quando um evento de tecla ‘keydown" é acionado, uma a função de *callback* é ativada. Através do uso de um bloco *switch-case*, é possível identificar qual tecla foi pressionada com base no código associado a cada tecla. Neste trabalho, escolheu-se o padrão “W”, “A”, “S” e “D” para representar as direções cima, esquerda, baixo e direita, respectivamente.

Para cada caso, a função Rust ‘set_snake_direction’ será chamada, passando como parâmetro uma das quatro opções do *enum* ‘Direction’, também declarado no Rust. A implementação é demonstrada no trecho de código 20.

```
document.addEventListener("keydown", e => {
	switch (e.code) {
		case "KeyW": world.set_snake_direction(Direction.Up); break;
		case "KeyS": world.set_snake_direction(Direction.Down); break;
		case "KeyA": world.set_snake_direction(Direction.Left); break;
		case "KeyD": world.set_snake_direction(Direction.Right); break;
	}
})
```

Código 20: Captura dos eventos de tecla no Javascript

4.4.5 Geração da Maçã e Função Random

A maçã tem um papel crucial no jogo Snake, pois atua como um sistema de recompensa que aumenta o tamanho da cobra cada vez que é ingerida. Para gerar a maçã no mapa, é necessário considerar que ela deve ser posicionada aleatoriamente e, ao ser consumida, precisa ser imediatamente realocada em uma célula não ocupada. Essa dinâmica garante que a maçã apareça de forma imprevisível e gere um desafio contínuo ao jogador.

Antes de tudo, é essencial criar uma função que gere um índice aleatório com base no tamanho do mapa. Esse índice será utilizado tanto para posicionar a cobra como a maçã em posições aleatórias no início do jogo, além de adicionar uma nova posição aleatória para a maçã todas as vezes em que ela for consumida.

A função pode ser implementada tanto em Rust quanto em JavaScript. No entanto, neste projeto, optaremos por declará-la em JavaScript e exportá-la para o Rust, a fim de exemplificar o uso da biblioteca ‘wasm-bindgen’.

A declaração da função será salva no arquivo ‘random.js’ na pasta ‘utils’. Para gerar um número aleatório, basta utilizar o método nativo do JavaScript ‘Math.random’ e multiplicá-lo pelo tamanho do mapa. Antes de declarar a função, é necessário adicionar o termo ‘export’, conforme exemplificado no trecho de código 21, uma vez que essa função será importada em outro local.

```
export function random(max) {
	return Math.floor(Math.random() * max);
}
```

Código 21: Função 'random' em Javascript

O método ‘Math.random’ sempre gera um número decimal aleatório entre 0 e 1, por exemplo, 0,456. Se tivermos um mapa com 36 índices (6 células de altura e 6 células de largura), multiplicar 0,456 por 36 resulta em 16,416. No entanto, como os índices do mapa devem ser inteiros, é necessário utilizar a função ‘Math.floor’ para arredondar esse valor para o número inteiro mais próximo, transformando 16,416 no índice 16.

```
#[wasm_bindgen(module = "/web/utils/random.js")]
extern {
        fn random(max: usize) -> usize;
}
```

Código 22: Uso da função 'random' pelo Rust

Na subseção 4.2.1, foi mostrado que para utilizar uma função do JavaScript no Rust, é necessário utilizar a anotação ‘#[wasm_bindgen]’ e fornecer o caminho onde a função está declarada. Além disso, deve-se utilizar o atributo *extern* para indicar que a função não pertence nativamente ao Rust. No trecho de código 22, é apresentada a declaração da função ‘random’.

Na *struct* ‘World’, foi adicionado um novo atributo para representar a maçã no jogo. Além disso, um método chamado ‘get_apple_cell’ foi criado para permitir o acesso ao índice da posição da maçã em outras partes do código.

```
fn gen_apple_cell(size: usize, snake_body: &Vec<SnakeCell>) -> usize {
        let mut apple: usize;
        loop {
                apple = random(size);
                if !snake_body.contains(&SnakeCell(apple)) { break; }
        }
        apple
}
```

Código 23: Implementação do método 'gen_apple_cell'

No método construtor, a função ‘random’ é utilizada para obter um índice aleatório para posicionar a cobra no mapa no início do jogo. Além disso, o atributo ‘apple’ é inicializado com o valor retornado pela função ‘gen_apple_cell’.

A função ‘gen_apple_cell’, demonstrada no trecho de código 23, recebe como parâmetros uma referência ao corpo da cobra e o tamanho do mapa. Em seguida, é utilizado um laço de repetição para determinar um índice válido para posicionar a maçã. A cada iteração, um novo índice aleatório é obtido a partir da função ‘random’. O *loop* continua até que se

encontre uma posição que não esteja sendo ocupada pelo corpo da cobra. Ao final da função, o índice é retornado, representando a posição válida para a maçã no mapa.

No JavaScript, será adicionada uma nova função chamada ‘drawApple’ para desenhar a maçã no jogo, seguindo a mesma estrutura das funções ‘drawWorld’ e ‘drawSnake’. A diferença é que a função ‘drawApple’ obterá o índice da maçã através do método Rust ‘get_apple_cell’. Com a adição dessa função, o jogo terá a representação visual da cobra e da maçã no mapa, sendo que a maçã será desenhada com a cor vermelha característica. Assim, a figura 11 traz uma demonstração do mapa com ambos os elementos desenhados.

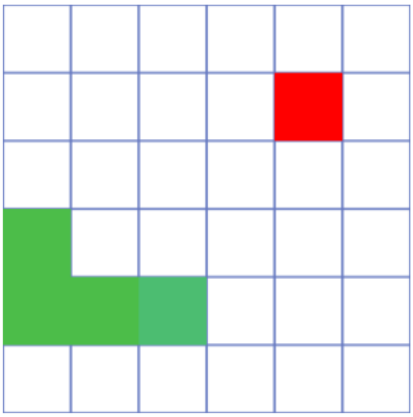

Figura 11: *Cobra e maçã sendo renderizados juntos no mapa*

4.4.6 Crescimento

A característica central que torna o jogo Snake desafiador é a habilidade da cobra de aumentar de tamanho ao comer uma maçã. Essa funcionalidade será implementada dentro do método ‘step’, o qual é executado em um laço contínuo de repetição, conforme apresentado no Item 4.4.4.3.

Para alcançar esse objetivo, é preciso implementar uma estrutura condicional que verifique se, em um dado momento, a célula frontal da cobra e a maçã estão ocupando o mesmo índice. Após confirmar a colisão, é necessário verificar se ainda existem espaços vazios no mapa para gerar uma nova maçã. Caso haja espaços livres disponíveis, a maçã consumida dará lugar a uma nova maçã, cujo índice será gerado aleatoriamente por meio do método ‘gen_apple_cell’.

Se a condição não for atendida, significa que a cobra aumentou tanto de tamanho que não há mais espaços vazios no mapa, levando-a a um estado de vitória no jogo. No entanto, esse cenário será deixado de lado por enquanto, uma vez que os estados de vitória ou derrota serão tratados na subseção seguinte.

```
        if self.apple == self.get_snake_head() {
                if self.get_snake_lenght() < self.size - 1 {
                        self.apple = World::gen_apple_cell(self.size, &self.snake.body);
                }

                let new_cell: usize = self.snake.body[self.get_snake_lenght() - 1].0;
                self.snake.body.push(SnakeCell(new_cell));
        }
}
```

Código 24: Bloco de verificação para detectar colisão entre a cobra e a maçã

O trecho de código 24 ilustra o bloco condicional, no qual após a verificação, um novo segmento é calculado usando o índice da última célula da cobra. Esse valor é, então, adicionado à posição final do vetor de células da cobra, resultando em um aumento efetivo do seu tamanho.

4.4.7 Estados de fim de jogo

No jogo Snake, o desfecho pode ser resumido em duas possibilidades: vitória ou derrota. Para expressar esses estados de forma clara no código, utilizaremos uma abordagem semelhante ao que foi feito no tratamento das direções da cobra. Será criado uma enumeração denominada ‘Status’, contendo os dois estados possíveis do jogo: *won* (vitória) e *lost* (derrota).

Na *struct* ‘World’, incluiremos um novo atributo chamado ‘state’, que será do tipo *Option* e representará o estado do jogo por meio do *enum* ‘Status’. Assim como o atributo ‘next_cell’, ‘state’ será inicializado com o valor ‘none’ no método construtor para indicar que o jogo começa sem nenhum estado definido. Além disso, será criado um método chamado ‘get_game_state’ para permitir o acesso ao atributo “state” também em JavaScript.

Com base nessas definições, podemos estabelecer as condições específicas do jogo. Como discutido na subseção anterior, quando a cobra preenche todas as células do mapa, o jogo é considerado em estado de vitória. Por outro lado, o estado de derrota ocorre quando a cobra colide com o seu próprio corpo. Ambas essas condições serão tratadas no método ‘step’.

Para determinar uma possível colisão da cobra consigo mesma, verificamos se o vetor que representa o corpo da cobra contém o mesmo elemento da primeira posição do vetor, que representa a célula frontal. Essa verificação é crucial, pois a célula frontal é onde qualquer colisão pode ocorrer. Realizamos essa verificação considerando o vetor da cobra a partir da posição 1 até o final de sua extensão. Caso a condição seja verdadeira, significa que ocorreu uma colisão interna, e, como resultado, o estado do jogo será atualizado para *Lost*.

```
if self.snake.body[1..self.get_snake_lenght()].contains(&self.snake.body[0]) {
        self.state = Some(Status::Lost);
}

if self.apple == self.get_snake_head() {
        if self.get_snake_lenght() < self.size - 1{ ...
        } else {
                self.apple = 1000;
                self.state = Some(Status::Won);
         }
```

Código 25: Adição da referência aos estados do jogo (*Won/Lost*) na função 'step'

O caso de vitória será obtido dentro do bloco condicional que verifica se a célula frontal da cobra esbarrou com a maçã. Caso não se tenha nenhuma posição livre para que a maçã seja desenhada, quer dizer que a cobra ocupou todos os espaços livres do mapa, sendo assim caso uma nova maçã não possa ser gerada, o estado do jogo será atualizado para W*on*. O trecho de código 25 ilustra as alterações realizadas para a transição de estados do jogo.

Em JavaScript, a função ‘gameStatus’ assumirá a responsabilidade de verificar o estado do jogo a cada atualização do método ‘update’. Essa verificação será realizada através da chamada da função ‘world.get_game_state’, que fornecerá o estado atual do jogo. Com base no estado obtido, a função utilizará a função ‘alert’ nativa do JavaScript para exibir

um *popup* na tela do jogador, informando-o sobre o resultado da partida, seja vitória ou derrota. Com isso a implementação do jogo Snake é finalizada.

5. CONCLUSÕES FINAIS

A web continua evoluindo incessantemente, com o surgimento diário de novas linguagens, bibliotecas e *frameworks*, que trazem abordagens inovadoras para o desenvolvimento. Nesse contexto, o WebAssembly se destaca como uma tecnologia consolidada, recebendo contribuições significativas de diversas empresas. Uma das inovações mais recentes é a versão do WASM para Docker, o que amplia ainda mais suas possibilidades e flexibilidade.

Contudo, devido a essa mesma evolução contínua da web, torna-se desafiador afirmar, com certeza, quais tecnologias permanecerão amplamente utilizadas. Ao analisar o capítulo 2, fica evidente que diversas tecnologias web emergiram e, em algum momento, tornaram-se obsoletas, abrindo espaço para o surgimento do WebAssembly.

É importante ressaltar que, durante a elaboração deste trabalho, o WebAssembly ainda se encontra na sua primeira versão, com melhorias incrementais sendo lançadas desde 2015. No entanto, a comunidade aguarda com expectativa uma nova versão, pois a WebAssembly Working Group tem prometido diversas mudanças significativas para atualizações futuras.

Sendo assim, somente o tempo poderá dizer, com certeza, se o WebAssembly permanecerá relevante no mercado ou se uma nova tecnologia surgirá, tornando-o obsoleto. No cenário web atual, com o crescimento da computação em nuvem e o aumento da preferência por aplicações web em detrimento das desktop, fornece um terreno fértil para o contínuo crescimento do WASM. A sua capacidade de executar códigos de alto desempenho no navegador tem o potencial de atender às demandas em constante mudança da indústria e dos usuários.

A linguagem Rust se mostra altamente promissora, demonstrando competência ao rivalizar com linguagens de baixo nível, como C e C++. Atualmente, ela possui uma comunidade forte e uma vasta quantidade de bibliotecas desenvolvidas. A decisão de Linus Torvalds de adotar Rust para escrever alguns módulos do Kernel Linux também confere credibilidade à linguagem.

Mesmo sendo relativamente nova, a linguagem Rust já exibe uma notável maturidade em seus conceitos. Como evidenciado neste trabalho, tanto em associação com o

WebAssembly e o JavaScript quanto de forma independente, Rust certamente continuará a desempenhar um papel fundamental no campo do desenvolvimento.

5.1 TRABALHOS FUTUROS

A pesquisa sobre o WebAssembly abre a possibilidade de realizar uma série de testes de desempenho, incluindo comparações diretas com o JavaScript. Além disso, esses testes de velocidade podem abranger comparações entre diferentes linguagens combinadas com o WASM, como por exemplo, comparar o desempenho do C++ com o C, entre outras opções.

No que se refere à implementação, é possível realizar melhorias que abrangem desde o aprimoramento da mecânica do jogo até a implementação de gráficos com modelos de sprites para a cobra e a maçã. Adicionalmente, o jogo pode incluir um placar de pontuações para fornecer informações ao jogador, bem como a opção de um possível modo de dois jogadores.

A base utilizada na confecção do jogo Snake pode ser aproveitada na elaboração de outros jogos semelhantes, como o Pong ou o Tetris, que compartilham conceitos similares aos demonstrados neste trabalho.

REFERÊNCIAS

BATTAGLINE, Rick. **The Art of Webassembly**: build secure, portable, high-performance applications. São Francisco: No Starch Press, 2021.

BENEDETTI, Leila. **Snake: conheça a história do jogo que revolucionou o mundo mobile**. Disponível em: <https://universoretro.com.br/snake-conheca-a-historia-do-jogo-que-revolucionou-o-mundo-mobile/>. Acesso em: 12 jun. 2023.

BOEIRA, Julia Naomi. **Programação Funcional e Concorrente em Rust**. São Paulo: Casa do Código, 2018.

BRITO, Lucas Edi Cordeiro de. **ANÁLISE COMPARATIVA ENTRE WEBASSEMBLY E JAVASCRIPT COMO ALVOS DE COMPILAÇÃO**. 2019. 7p. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Centro de Engenharia Elétrica e Informática, Universidade Federal de Campina Grande, Campina Grande, 2019.

BYTECODE ALLIANCE. **Bytecode Alliance**. Bytecode Alliance. Disponível em: <https://bytecodealliance.org/>. Acesso em: 05 mar. 2023.

CASTELLANI, Marcelo. **Rust: Concorrência e alta performance com segurança**. São Paulo: Casa do Código, 2018.

Cresce o uso de Internet durante a pandemia e número de usuários no Brasil chega a 152 milhões, é o que aponta pesquisa do Cetic.br. CETIC (Centro Regional de Estudos para o Desenvolvimento da Sociedade da Informação). Disponível em <https://cetic.br/pt/noticia/cresce-o-uso-de-internet-durante-a-pandemia-e-numero-de-usuarios-no-brasil-chega-a-152-milhoes-e-o-que-aponta-pesquisa-do-cetic-br/>. Acesso em: 02 jun. 2022.

DIPIERRO, Massimo. **The Rise of JavaScript. Computing in Science & Engineering**, v. 20, n. 1, p. 9-10, jan./fev. 2018. Disponível em: <https://www.computer.org/csdl/magazine/cs/2018/01>. Acesso em: 03 jun. 2022.

EATON, Phil. **Enumerating and analyzing 40+ non-V8 JavaScript implementations**. Disponível em: <https://notes.eatonphil.com/javascript-implementations.html>. Acesso em: 06 mai. 2023.

ENDEL A: **WebAssembly – What it is & Why is it so important**. Disponível em: <https://arghya.xyz/articles/webassembly-wasm-wasi/>. Acesso em: 14 mar. 2023.

ELDEL B: NEVES, Vinicius Kiatkoski. **Webpack concepts: Loaders and Plugins.** Disponível em: <https://dev.to/viniciuskneves/webpack-concepts-loaders-and-plugins-5ed0>. Acesso em: 14 jun. 2023.

ELDEL C: IOHK SUPPORT. **Rust, WebAssembly, JavaScript, and npm**. Disponível em: <https://iohk.zendesk.com/hc/en-us/articles/900000930823-Rust-WebAssembly-JavaScript-and-npm\>. Acesso em: 15 jul. 2023.

FROTA, Harderson. **HotSpot e JIT Compiler, entendendo como funciona**. Disponível em: <https://www.handersonfrota.com.br/hotspot-e-jit-compiler-entendendo-como-funciona/> Acesso em: 06 mai. 2023.

GRILLO, Filipe del Nero; FORTES, Renata Pontin de Mattos. **Aprendendo javascript**. São Carlos: ICMC-USP. Disponível em: <https://repositorio.usp.br/directbitstream/4cd7f9b7-7144-40f4-bfd0-7a1d9a6bd748/nd_72.pdf>. Acesso em: 26 fev. 2023.

HOFFMAN, Kevin. **Programming WebAssembly with Rust**. 1. ed. Raleigh: The Pragmatic Programmers, 2019.

JOHNSON, Paul; Kaihlavirta, Vesa. **Learning Rust: A comprehensive guide to writing Rust applications**. Birmingham: Packt Publishing Ltd, 2017.

KACHAN, Dana. **7 Reasons Why Businesses Are Migrating to Web Applications**. Disponível em <https://dev.to/danakachan/7-reasons-why-businesses-are-migrating-to-web-applications-epg>. Acesso em: 02 jun. 2022.

KOCH, Thilo. **Um panorama sobre Rust**. 2015. 12 p. Monografia – Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2015. Disciplina: MAC 5742 – Introdução à Computação Paralela e Distribuída. Professor: Alfredo Goldman Vel Lejbman.

KLABNIK, Steve; Nichols, Carol. **The Rust Programming Language**. São Francisco: No Starch Press, 2018.

LYAMKIN, Ilya. **How JavaScript Works: Under the Hood of the V8 Engine**. Disponível em: https://www.freecodecamp.org/news/javascript-under-the-hood-v8/. Acesso em: 02 mai. 2023.

MOZILLA. **WebAssembly**. Mozilla Developer Network. Disponível em: <https://developer.mozilla.org/pt-BR/docs/WebAssembly> Acesso em: 05 mar. 2023.

MDN WEB DOCS. **asm.js.** Disponível em: <https://developer.mozilla.org/en-US/docs/Games/Tools/asm.js?utm_campaign=feed&utm_medium=rss&utm_source=developer.mozilla.org\>. Acesso em: 01 mai. 2023.

MDN WEB DOCS. **Canvas API Tutorial: Basic usage**. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial/Basic_usage\>. Acesso em: 10 mai. 2023.

MDN WEB DOCS. **Rust to WebAssembly**. Disponível em: <https://developer.mozilla.org/en-US/docs/WebAssembly/Rust_to_Wasm\>. Acesso em: 13 jun. 2023.

NELLAIYAPEN, Sendil Kumar. **Rust And WebAssembly – wasm-pack**. Disponível em: <https://sendilkumarn.com/blog/wasm-pack\>. Acesso em: 13 jun. 2023.

OLIVEIRA, Felipe Rodrigues; MAZIERO, Ronaldo Colucci; ARAÚJO, Liriane Soares de. ESTUDO SOBRE A WEB 3.0. **Revista Interface Tecnológica**, v. 15, n. 2, dez, 2018, doi: 10.31510/infa.v15i2.492. p. 60-71.

PEHAM, Thomas. **O que é WebAssembly e por que ele afeta todos os desenvolvedores web!** Disponível em <https://imasters.com.br/back-end/o-que-e-webassembly-e-por-que-ele-afeta-todos-os-desenvolvedores-web>. Acesso em 04 jun. 2022.

RESIG, John. **Asm.js: The JavaScript Compile Target**. Disponível em: <https://johnresig.com/blog/asmjs-javascript-compile-target/>. Acesso em: 01 mai. 2023.

RODRIGUES, João Paulo Mendes. **Linguagem Orientada a Aspetos para Transformação de Webassembly**. 2022. 181p. Dissertação (Mestrado) - Curso de Engenharia Informática, Departamento de Informática e Sistemas, Instituto Superior de Engenharia de Coimbra, Coimbra, 2022.

ŠIPEK, Matija; MUHAREMAGIĆ, Dino; MIHALJEVIĆ, Branko; RADOVAN, Aleksander. **Next-generation Web Applications with WebAssembly and TruffleWasm**. In: INTERNATIONAL CONVENTION ON INFORMATION, COMMUNICATION AND ELECTRONIC TECHNOLOGY, 44, 2021, Zagreb, Croácia. doi: 10.23919/MIPRO52101.2021.9596883. p. 1695-1700.

SLETTEN, Brian. **WebAssembly: The Definitive Guide**: safe, fast, and portable code. 1. ed. Sebastopol: O'Reilly Media, 2022.

SMITH, Eric. **Game Development with Rust and WebAssembly**. 1. ed. Birmingham, UK: Packt Publishing, 2022.

STACK OVERFLOW. **Stack Overflow Annual Developer Survey 2022**. Disponível em: <https://survey.stackoverflow.com/2022/>. Acesso em: 01 mar. 2023.

STENDER, Kleber. **Webpack – Uma introdução completa**. Disponível em: <https://kiberstender.github.io/pt_br/webpack-introduction/>. Acesso em: 13 jun. 2023.

SUBRAMANIAN, Srikumar. **Talk: The Nuts and Bolts of WebAssembly**. Codaholic. Disponível em: <https://sriku.org/blog/2019/08/24/talk-the-nuts-and-bolts-of-webassembly/>. Acesso em: Acesso em: 03 mar. 2023.

TANENBAUM, Andrew Stuart; VAN STEEN, Maarten. **Sistemas Distribuídos**: princípios e paradigmas. 2. ed. São Paulo: Pearson Education Inc, 2007.

VAUGHAN-NICHOLS, Steven J. **Rust in the Linux Kernel**. The New Stack. Disponível em: <https://thenewstack.io/rust-in-the-linux-kernel/>. Acesso em: 01 mar. 2023.

WAGNER, Geoff. **What is the Difference Between Interpreted and Compiled Languages in DevOps**. Disponível em: <https://www.valewood.org/devops-interpreted-vs-compiled-languages/>. Acesso em: 03 mai 2023.

WEBASSEMBLY. **WebAssembly**. WebAssembly, Disponível em: <https://webassembly.org/>. Acesso em: 06 mar. 2023.

WEBASSEMBLY. **WebAssembly Specification**. Disponível em: <https://webassembly.github.io/spec/>. Acesso em: 15 mar. 2023.

WIRFS-BROCK, Allen; EICH, Brendan. Javascript: the first 20 years. Proceedings Of The Acm On Programming Languages, v. 4, 12 jun. 2020. **Association for Computing Machinery (ACM)**. doi: 10.1145/3386327. p. 1-189

WOZNIEWICZ, Brendon. **The Difference Between a Framework and a Library**. Disponível em: https://www.freecodecamp.org/news/the-difference-between-a-framework-and-a-library-bd133054023f/. Acesso em: 12 jun. 2023.

W3C. **WebAssembly Working Group**. World Wide Web Consortium. Disponível em: <https://www.w3.org/wasm/>. Acesso em: 05 mar. 2023.

ZHOU, Heyang. **A Comparison between WebAssembly and RISC-V**. Disponível em: <https://medium.com/@losfair/a-comparison-between-webassembly-and-risc-v-e8fb9d37e6cc>. Acesso em: 05 mai. 2023.