

**INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DA PARAÍBA
CAMPUS DE CAJAZEIRAS
CURSO SUPERIOR DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE
SISTEMAS**

DANIEL DE OLIVEIRA SOUSA

Figure Engine: UMA BIBLIOTECA PARA PROTOTIPAÇÃO DE JOGOS COM
TYPESCRIPT

CAJAZEIRAS
2024

Daniel de Oliveira Sousa

Figure Engine: UMA BIBLIOTECA PARA PROTOTIPAÇÃO DE JOGOS COM
TYPESCRIPT

Trabalho de Conclusão de Curso
submetido ao Instituto Federal de
Educação, Ciência e Tecnologia da
Paraíba, Campus Cajazeiras, como
requisito parcial para a obtenção do
grau de Tecnólogo em Análise e
Desenvolvimento de Sistemas.

Orientador: Prof. Dr. Fabio Gomes
de Andrade

Cajazeiras
2024

IFPB / Campus Cajazeiras
Coordenação de Biblioteca
Biblioteca Prof. Ribamar da Silva
Catalogação na fonte: Cícero Luciano Félix CRB-15/750

S725f Sousa, Daniel de Oliveira.
Figure engine : uma biblioteca para prototipação de jogos com
typescript / Daniel de Oliveira Sousa. – 2024.
45f. : il.
Trabalho de Conclusão de Curso (Tecnólogo em Análise e
Desenvolvimento de Sistemas) - Instituto Federal de Educação,
Ciência e Tecnologia da Paraíba, Cajazeiras, 2024.
Orientador(a): Prof. Dr. Fabio Gomes de Andrade.
1. Desenvolvimento de sistemas. 2. Motor de jogos. 3.
Desenvolvimento de *games*. 4. *Typescript*. I. Instituto Federal de
Educação, Ciência e Tecnologia da Paraíba. II. Título.

IFPB/CZ

CDU: 004.4(043.2)



MINISTÉRIO DA EDUCAÇÃO
SECRETARIA DE EDUCAÇÃO PROFISSIONAL E TECNOLÓGICA
INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DA PARAÍBA

DANIEL DE OLIVEIRA SOUSA

FIGURE ENGINE: UMA BIBLIOTECA PARA PROTOTIPAÇÃO DE JOGOS COM TYPESCRIPT

Trabalho de Conclusão de Curso apresentado junto ao Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas do Instituto Federal de Educação, Ciência e Tecnologia da Paraíba - Campus Cajazeiras, como requisito à obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas.

Orientador

Prof. Dr. Fabio Gomes de Andrade

Aprovada em: **16 de Outubro de 2024.**

Prof. Dr. Fabio Gomes de Andrade - Orientador

Prof. Me. Diogo Dantas Moreira - Avaliador

IFPB - Campus Cajazeiras

Prof. Tec.go. Antônio Ricart Jacinto de Oliveira Medeiros

IFPB - Campus Cajazeiras

Documento assinado eletronicamente por:

- **Fabio Gomes de Andrade**, PROFESSOR ENS BASICO TECN TECNOLOGICO, em 17/10/2024 19:40:14.
- **Diogo Dantas Moreira**, PROFESSOR ENS BASICO TECN TECNOLOGICO, em 19/10/2024 08:54:00.
- **Antonio Ricart Jacinto de Oliveira Medeiros**, PROF ENS BAS TEC TECNOLOGICO-SUBSTITUTO, em 21/10/2024 11:44:50.

Este documento foi emitido pelo SUAP em 16/10/2024. Para comprovar sua autenticidade, faça a leitura do QRCode ao lado ou acesse <https://suap.ifpb.edu.br/autenticar-documento/> e forneça os dados abaixo:

Código 620978
Verificador: ccef6fcf7e
Código de Autenticação:



Rua José Antônio da Silva, 300, Jardim Oásis, CAJAZEIRAS / PB, CEP 58.900-000
<http://ifpb.edu.br> - (83) 3532-4100

AGRADECIMENTOS

Agradeço, primeiramente, a Deus por ter me proporcionado saúde e tudo o que foi necessário para eu chegar até aqui.

Agradeço aos meus pais, Agenor Caboclo de Sousa e Luciene Batista de Oliveira, por me educarem com todo o amor e incentivo necessários para minha vida pessoal, acadêmica e profissional.

Agradeço ao meu irmão, José David de Oliveira Sousa, por ser um exemplo e me inspirar a cursar Análise e Desenvolvimento de Sistemas.

Agradeço aos meus amigos e amigas da faculdade, que me deixaram ótimas memórias do tempo em que estudamos juntos.

Agradeço ao meu orientador, Prof. Dr. Fábio Gomes de Andrade, por todo o suporte e atenção oferecidos para que este trabalho fosse concluído.

RESUMO

O desenvolvimento de jogos é um ramo de tecnologia crescente no Brasil. O número de empresas deste segmento vem subindo e, com isso, a quantidade de jogos produzidos também. Segundo a Pesquisa da Indústria Brasileira de Games, de 2022, realizada pela AbraGames, de 2018 para 2022 o número de estúdios desenvolvedores de jogos havia passado de 375 para 1009. A produção desses *softwares* pode ser realizada por meio de código puramente ou, mais frequentemente, com o auxílio de motores, que são ferramentas especializadas em ajudar no seu desenvolvimento. Tendo em vista que alguns motores, por serem muito complexos, impossibilitam que desenvolvedores interessados possam definir detalhes já abstraídos, fazendo com que eles tenham que criar seus próprios motores, esse trabalho propõe um motor de jogos que é simples o suficiente para permitir seu uso e sua customização por esses desenvolvedores. O motor proposto foi implementado em *Typescript* para possibilitar o seu uso em uma variedade de dispositivos com acesso a navegadores *web* e foi disponibilizado sob licença de código aberto para permitir que os desenvolvedores interessados possam usá-lo ou estendê-lo livremente de acordo com suas necessidades.

Palavras-chave: Biblioteca. Jogos. Motor de jogos. Protótipos.

ABSTRACT

Game development is a growing technology branch in Brazil. The number of enterprises of the genre is rising and, with that, the amount of produced games is growing as well. According to the Research of the Brazilian Game Industry, from 2022, realized by AbraGames, from 2018 to 2022 the number of game development studios had grown from 375 to 1009. The production of these softwares can be done with code purely or, more often, with the help of engines, which are tools specialized in helping their development. Considering that some engines, for their complexity, make it impossible for interested developers to define already abstracted details, making them have to create their own engines, this project proposes a game engine that's simple enough to allow its use and its customization by these developers. The engine proposed was implemented in Typescript to allow its use in a variety of devices with access to web browsers and was made available under an open source license to allow that interested developers can use it or extend it freely according to their necessities.

Keywords: Game Engine. Games. Library. Prototypes.

LISTA DE FIGURAS

Figura 1 - Principais motores do mercado em 2022.....	15
Figura 2 - Renderização no Canvas	22
Figura 3 - Arquitetura do projeto	26
Figura 4 - Trecho de código configurando um jogo sem a Figure Engine.....	28
Figura 5 - Trecho de código configurando um jogo com a Figure Engine.....	29
Figura 6 - Trecho de código acessando input com a Figure Engine.....	30
Figura 7 - Trecho de código definindo um objeto com a Figure Engine.....	31
Figura 8 - Trecho de código atualizando um objeto manualmente com a Figure Engine.....	32
Figura 9 - Estudo de caso após a segunda sprint ter sido desenvolvida.....	33
Figura 10 - Trecho de código criando uma fase com a Figure Engine.....	34
Figura 11 - Trecho de código importando recursos sem a Figure Engine.....	35
Figura 12 - Trecho de código importando recursos com a Figure Engine.....	35
Figura 13 - Trecho de código carregando recursos com a Figure Engine.....	36
Figura 14 - Versão do estudo de caso após a quinta sprint.....	37
Figura 15 - Trecho de código criando câmera com a Figure Engine.....	38

LISTA DE QUADROS

Quadro 1 - Comparação entre os principais motores do mercado em 2022..... 19

LISTA DE ABREVIATURAS E SIGLAS

2D	Bidimensional
3D	Tridimensional
API	Interface de Programação de Aplicação
CSS	Folhas de Estilo em Cascata
CSS3	Terceira versão do CSS
DOM	Modelo Documento Objeto
FPS	Frames por segundo
GNU GPL	Licença Pública Geral
HTML	Linguagem de Marcação de Hipertexto
HTML5	Quinta versão do HTML
IA	Inteligência Artificial
JS	Javascript
MIT	Licença de <i>software</i> permissiva
NPM	Node Package Manager
TCC	Trabalho de Conclusão de Curso
UI	Interface de Usuário

SUMÁRIO

1. INTRODUÇÃO	9
1.1 Motivação	11
1.2 Objetivos	12
1.2.1 Objetivo Geral	12
1.2.2 Objetivos Específicos	13
1.3 Metodologia	13
1.4 Organização do Documento	14
2. TRABALHOS RELACIONADOS	15
2.1 Unity	16
2.2 Unreal	17
2.3 Blender	17
2.4 Construct	17
2.5 GameMaker	18
2.6 Godot	18
3. FUNDAMENTAÇÃO TEÓRICA	19
3.1 A Linguagem TypeScript	19
3.2 A API Canvas	21
4. A SOLUÇÃO PROPOSTA	23
4.1 Stakeholders	23
4.2 Requisitos Funcionais	24
4.3 Arquitetura	25
4.4 Estudo de Caso	26
4.5 Implementação	27
4.5.1 Sprint 1: Quadro e FPS	27
4.5.2 Sprint 2: Entrada e Objetos	39
4.5.3 Sprint 3: Construção de Fases	33
4.5.4 Sprint 4: Importação de Recursos	34
4.5.5 Sprint 5: Carregamento de Fases	35
4.5.6 Sprint 6: Câmera e Customização	37
5. CONCLUSÃO	39
REFERÊNCIAS	40
GLOSSÁRIO	41

1. INTRODUÇÃO

A separação entre os conceitos de jogos eletrônicos e de motor de jogos (*game engine*) pode ser difícil de ser realizada e de fato não era no início da indústria dos jogos. Antigamente, era comum que as empresas escrevessem os jogos em linguagens de baixo nível, como a *Assembly*, e não se fizesse uso de *softwares* intermediários como motores na sua construção. Devido aos poucos recursos da época, não havia a prática de separar o que era jogo e o que era motor (BISHOP, EBERLY, FINCH, SHANTZ, e WHITTED, 1998). Somente depois, com o surgimento de *hardwares* mais avançados, começou-se a realizar essa tentativa de separar os dois conceitos para que o código muitas vezes repetitivo dos motores pudesse ser disponibilizado separadamente para auxiliar e facilitar a construção de novos jogos.

Atualmente, pode-se entender que um motor de jogos é fundamentalmente aquele software que gerencia toda a base necessária para que um jogo possa ser implementado, abstraindo o desenvolvimento de características comuns à maioria dos jogos ou à maioria dos jogos de certo gênero (no caso de motores específicos para um gênero). As funcionalidades que estes motores trazem consigo podem variar dependendo do motor, como, por exemplo, o intuito com o qual ele foi feito ou se ele foi desenvolvido para um gênero específico de jogos.. Entretanto, elas geralmente consistem em funções como a renderização de animações na tela, a reprodução de sons, um motor de física ou de colisões, inteligência artificial, a habilidade de implementar códigos personalizados, de reproduzir animações, de se conectar à rede e de gerenciar a memória e as cenas do jogo (ALEEM, CAPRETZ e AHMED, 2016).

Em contraponto aos motores de jogos, pode-se dizer que o jogo é todo o restante do software, que tem como função definir as regras, os comportamentos, a lógica e até mesmo a história que se constituem em pontos únicos de um jogo e não podem ser previamente descritos e implementados pelo motor (BISHOP, EBERLY, FINCH, SHANTZ, e WHITTED, 1998).

A implementação de um motor de jogos pode se dar de diferentes maneiras: o motor pode ser construído em forma de biblioteca, de *framework*, ou, como ocorre na maioria dos motores, em um formato no qual aplicativos inteiros são criados para facilitar o trabalho do desenvolvedor de jogos, trazendo consigo recursos com interfaces amigáveis que abstraem ainda mais o seu funcionamento. Em muitos

casos, os motores trazem também o uso de novas linguagens de programação feitas especificamente para o trabalho com esses ambientes.

Dentre os recursos mais comuns que costumam ser providos pelos motores de jogos estão: formas de se renderizar o jogo na tela, uma vez que os jogos mais convencionais utilizam recursos de interface, com menus, e diferentes tipos de artes gráficas; formas de se reproduzir músicas e efeitos sonoros, que também são normalmente partes intrínsecas dos jogos; formas de se realizar uma constante atualização da renderização do jogo a uma determinada taxa de quadros por segundo, uma vez que, novamente, os jogos e até mesmo outros tipos de mídia precisam disso para realizar suas animações e mostrar algum retorno visual às ações do jogador; maneiras de se criar *sprites* e outros tipos de artes para o jogo, além de possibilitar as animações dessas criações, visto que para os jogos terem a interação gráfica com os jogadores, eles necessitam desse tipo de recurso; formas de se exibir textos na tela para que o jogo possa ter esse tipo de comunicação mais direta com seus jogadores; além disso, é muito comum que os motores tragam formas de se criar níveis para o jogo, além dos recursos necessários para montá-los, tais como *sprites*, objetos, *backgrounds*, *tiles* entre outros. Outros recursos que os motores de jogos podem incluir são formas de se detectar e manusear colisões de objetos e funções embutidas para a realização de cálculos físicos (ALEEM, CAPRETZ e AHMED, 2016).

No momento atual do mercado existe um número diverso de motores de jogos. Segundo uma pesquisa de 2022 da *AbraGames*¹ (FORTIM, 2022), os principais motores utilizados atualmente consistem em aplicações desktop que trazem funcionalidades completas e complexas para o desenvolvimento de jogos. Alguns desses motores, como o *Unity*² e o *Unreal*³, focam nos jogos 3D, enquanto outros, como o *Construct*⁴ e o *GameMaker*⁵, focam nos jogos 2D. Ainda segundo essa pesquisa, também existe uma quantidade considerável de desenvolvedores (7% deles, o mesmo número de desenvolvedores que utilizam o *GameMaker* ou o *Godot*⁶) que preferem usar tecnologias próprias como seus motores devido à falta

¹ <https://www.abragames.org>

² <https://unity.com>

³ <https://www.unrealengine.com>

⁴ <https://www.construct.net>

⁵ <https://gamemaker.io>

⁶ <https://godotengine.org>

de flexibilidade e personalização oferecidas pelos motores existentes atualmente no mercado.

Focando nesse grupo de desenvolvedores, o intuito deste trabalho é construir um motor de jogos no formato de uma biblioteca utilizando-se a tecnologia *TypeScript*⁷ para jogos 2D, que tenha como principal conceito ser simples, porém robusto o suficiente para dar suporte ao desenvolvimento de jogos de pequeno porte, de modo que seja possível utilizá-lo para a construção de jogos mais simples ou adaptá-lo para versões próprias que atendam às preferências e necessidades individuais dos desenvolvedores. Assim, o trabalho desenvolvido provê um esqueleto inicial para os usuários que preferem desenvolver os seus próprios motores. Para cumprir esse objetivo, a ferramenta é mantida sob licença de código aberto, possibilitando, desse modo, que qualquer desenvolvedor interessado possa construir a partir dela, evoluindo a mesma para o seu caso de uso específico.

1.1 Motivação

Quando se pretende desenvolver um jogo eletrônico, ou até mesmo outros tipos de *softwares*, tem-se que fazer escolhas a respeito de quais tecnologias usar e se o seu uso será auxiliado por alguma ferramenta (bibliotecas ou *frameworks*). Essas escolhas dependem de muitos fatores que norteiam o jogo em questão, como, por exemplo, para qual plataforma ele será disponibilizado, qual é a performance esperada ou se ele terá funcionalidades que seriam facilitadas por alguma tecnologia específica.

O desenvolvimento de jogos é uma tarefa muito complexa, principalmente por ser uma atividade incrivelmente multidisciplinar, exigindo conhecimento em diversas áreas - desde o design de controles, de arte e de sons à programação e desenvolvimento de IA (ALEEM, CAPRETZ e AHMED, 2016).. Atualmente, há no mercado diversos motores de jogos para ajudar os desenvolvedores a realizarem o seu complicado trabalho com mais facilidade. Esses motores consistem em *softwares* que provêm uma estrutura básica para os jogos. Eles se apresentam nas mais diversas linguagens de programação, possibilitando a escolha do desenvolvedor.

⁷ <https://www.typescriptlang.org>

Apesar da existência dessas ferramentas, ainda há muitos programadores que preferem fazer os seus próprios motores. Eles representavam 7% dos desenvolvedores de jogos no Brasil em 2022, segundo pesquisa da *AbraGames*, um número considerável, quando se compara com a quantidade de programadores que preferem motores como o *Godot* ou o *GameMaker*, que também é de 7%.

Para ajudar esse público alvo, este trabalho propõe um novo motor para o desenvolvimento de jogos. Ele foi implementado em *TypeScript* e foi baseado em uma construção simplista, na qual tentou-se mantê-lo um bom motor sobretudo para jogos mais pequenos e protótipos. Entretanto, o motor desenvolvido é altamente customizável e adaptável para os desenvolvedores que preferem construir os seus próprios motores. Assim, ele oferece uma estrutura inicial para esse público. Com o fim de possibilitar essa adaptabilidade, o *software* construído aqui será disponibilizado sob licença de código aberto. Assim, os interessados poderão se sentir livres para usá-lo.

1.2 Objetivos

Esta seção descreve os principais objetivos que o projeto desenvolvido neste trabalho pretende cumprir, tanto o objetivo geral quanto aqueles mais específicos.

1.2.1 Objetivo Geral

O objetivo geral deste trabalho consiste na criação de uma biblioteca que funciona como um motor de jogos que pode ser utilizado por desenvolvedores para a criação de jogos ou protótipos 2D utilizando a linguagem *TypeScript* (ou, se desejado, *JavaScript*⁸).

Com o desenvolvimento dessa ferramenta, buscou-se permitir que desenvolvedores interessados em construir projetos simples possam ir direto ao ponto em suas implementações, sem perder tempo com demoradas configurações iniciais. Para possibilitar essa agilidade, foi escolhida a linguagem *TypeScript*, que permite que os desenvolvedores usuários possam programar com facilidade da maior parte dos computadores que tenham navegador web (DA CRUZ SILVA, 2016), o que é crescentemente a maioria, tanto com a linguagem citada ou com *JavaScript*, se a pessoa interessada não se importar com a tipagem estática.

⁸ <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript>

Além do desenvolvimento do *software* principal, também foi desenvolvido um protótipo de jogo que foi usado como estudo de caso para validar o motor construído.

1.2.2 Objetivos Específicos

Os seguintes objetivos específicos também são buscados pelo trabalho:

- permitir aos desenvolvedores usuários implementarem protótipos e jogos simples a partir da biblioteca criada;
- certificar-se de que a ferramenta permita a exportação de jogos para *web*;
- garantir que os usuários possam aprender a usar o projeto de um modo facilitado através de uma documentação robusta;
- apresentar uma validação da efetividade da ferramenta através da implementação de um caso de uso.

1.3 Metodologia

Para que houvesse uma boa execução do trabalho em questão, foi importante a sua organização em etapas. Essa separação proporcionou um melhor gerenciamento do projeto. A seguir estão as atividades que foram realizadas durante o desenvolvimento deste TCC:

- **Análise do estado da arte (A_1):** durante o desenvolvimento deste trabalho foi feita a análise dos projetos relacionados existentes (a exemplo dos outros motores de jogos citados), para garantir uma boa execução do projeto bem como para se ter ciência de como a solução a ser desenvolvida poderia se diferenciar das demais ferramentas;
- **Elaboração do documento de TCC (A_2):** essa atividade consistiu na escrita do documento do TCC em si;
- **Levantamento de requisitos (A_3):** etapa que se refere à escolha de que condições a ferramenta alvo do trabalho deveria atender, ou seja, quais deveriam ser suas funcionalidades e restrições de desenvolvimento e funcionamento;
- **Escolha da arquitetura do projeto (A_4):** processo no qual definiu-se qual seria o esqueleto do projeto - sua arquitetura - e organização;

- **Implementação (A₅):** etapa na qual ocorreu o desenvolvimento em si do projeto, ou seja, a escrita de seu código fonte.

1.4 Organização do Documento

O restante deste documento está organizado da seguinte forma: o Capítulo 2 mostra e explica alguns trabalhos relacionados; o Capítulo 3 descreve e fundamenta algumas tecnologias que foram utilizadas no desenvolvimento do projeto; o Capítulo 4 apresenta em detalhes o desenvolvimento da biblioteca, descrevendo, entre outras coisas, os seus requisitos funcionais, a sua arquitetura e o estudo de caso que foi desenvolvido em conjunto com a ferramenta; o Capítulo 5 descreve como foi feita a implementação do projeto; já o Capítulo 6 conclui o documento e aponta possíveis trabalhos futuros.

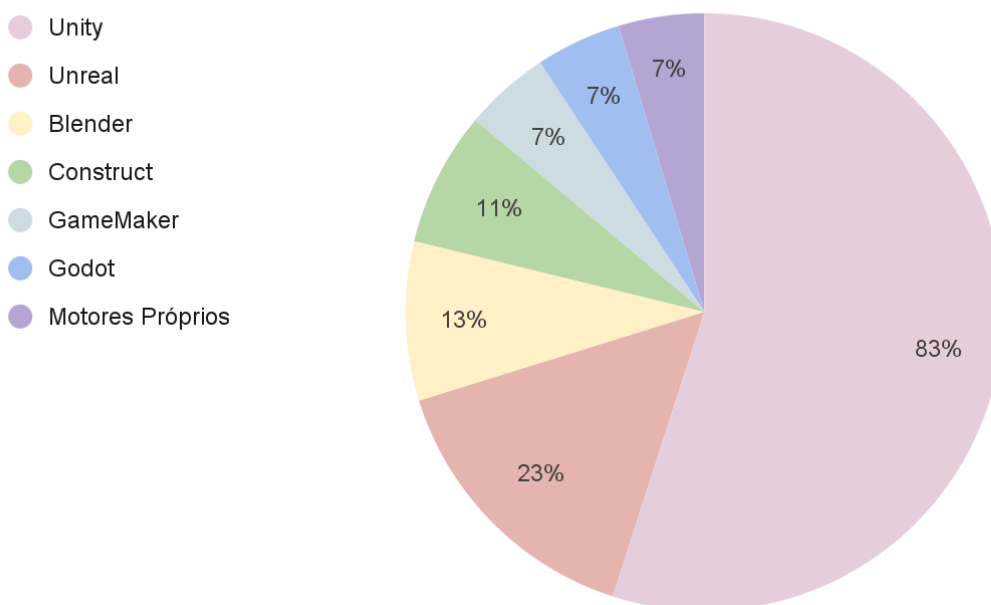
2. TRABALHOS RELACIONADOS

Dentre os principais trabalhos científicos utilizados para o desenvolvimento deste projeto, pode-se citar o artigo “Designing a PC Game Engine” (BISHOP, EBERLY, FINCH, SHANTZ, WHITTED, 1998) e o trabalho “Game development software engineering process life cycle: a systematic review” (ALEEM, CAPRETZ, AHMED, 2016), de onde foram extraídas informações importantes sobre os motores de jogos e a indústria do *video game* e a história desse mercado.

Além dos documentos supracitados, uma fonte essencial de dados para este projeto foi a Pesquisa da Indústria Brasileira de Games (FORTIM, 2022), realizada pela *AbraGames*, que mostra os principais motores de jogos do mercado nacional. A Figura 1 apresenta um gráfico que mostra quais são esses motores e os seus respectivos níveis de presença na época, de acordo com a pesquisa em questão.

Figura 1: Principais motores do mercado em 2022

Principais motores de jogos do mercado



Fonte: Adaptado de FORTIM, 2022

Como pode-se ver, já existem alguns motores de jogos populares no mercado. A Figura 1 mostra que os mais utilizados, no momento da pesquisa, eram: *Unity* (utilizado por 83% dos desenvolvedores), *Unreal* (que tem 23% de uso),

*Blender*⁹ (dono de 13% do cenário nacional), *Construct* (com 11% dos usuários), *GameMaker* (com 7% dos programadores) e *Godot* (também com 7% dos desenvolvedores), seguidos por motores autorais (com os mesmos 7% de adeptos), que correspondem aos casos nos quais o desenvolvedor decide construir seu jogo sem um motor de terceiros para auxiliá-lo.

Como anteriormente mencionado, depois desses motores, o que mais é utilizado para criação de jogos são os motores autorais, feitos pelos próprios desenvolvedores de jogos. Visando esse público alvo, foi desenvolvido nesse trabalho a *Figure Engine*, que, em contraponto aos motores supracitados, é uma biblioteca de código. Além disso, o motor desenvolvido neste trabalho tem código aberto e oferece funções básicas, tentando se manter o mais simples e enxuto possível, para que os desenvolvedores que costumam optar por criar seus próprios motores possam ter um ponto de partida por onde começar, construindo, a partir da *Figure*, seus próprios projetos.

Para trazer um maior embasamento sobre os trabalhos relacionados existentes no mercado atualmente, no final deste capítulo o Quadro 1 traz uma tabela comparativa entre as soluções existentes e a *Figure Engine*. Além disso, as próximas seções discorrem sobre esses outros projetos em detalhes.

2.1 Unity

O *Unity* é um motor de jogos proprietário comercial da empresa *Unity Technologies*. Ele serve tanto para jogos 2D quanto para os 3D e inclui diferentes planos de uso, sendo alguns deles gratuitos. Este motor se trata de uma aplicação desktop com interface gráfica, que normalmente deve ser instalada no computador do usuário para poder ser utilizada.

A linguagem de programação que os desenvolvedores usuários utilizam com esta *engine* é o C#, da *Microsoft*. O *Unity* permite o desenvolvimento para diferentes plataformas, sendo alguns exemplos de jogos conhecidos feitos usando essa ferramenta: *Genshin Impact*¹⁰, *Fall Guys: Ultimate Knockout*¹¹, *Escape from Tarkov*¹² e *Hollow Knight*¹³ (SALMELA, 2022).

⁹ <https://www.blender.org>

¹⁰ <https://genshin.hoyoverse.com>

¹¹ <https://www.fallguys.com>

¹² <https://www.escapefromtarkov.com>

¹³ <https://www.hollowknight.com>

2.2 Unreal

O *Unreal Engine* é um motor comercial da empresa *Epic Games*. Ele costuma ser usado para a construção de jogos 3D, sendo muito utilizado nos jogos AAA (jogos de alto orçamento feito por empresas consolidadas no mercado). Alguns exemplos de jogos que foram feitos com essa tecnologia são: *Final Fantasy VII Remake*¹⁴, *PUBG: BATTLEGROUNDS*¹⁵, a série *Borderlands*¹⁶, *Tekken 7*¹⁷, e o *Fortnite*¹⁸, que também pertence à *Epic Games*.

O *Unreal* também é um aplicativo desktop que possui planos pagos e gratuitos, assim como o *Unity*. Para o desenvolvimento de jogos, porém, a linguagem usada é C++. Entretanto o seu editor também provê uma forma de programação visual. Finalmente, o *Unreal* disponibiliza documentação para ajudar seus usuários a aprender a utilizá-lo (SALMELA, 2022).

2.3 Blender

O *Blender* é uma ferramenta de modelagem 3D que é distribuída sob a licença de código aberto GNU GPL. Sendo assim, esse *software* tem seu uso liberado de forma gratuita. Além de permitir a modelagem 3D, esse programa permite que os seus usuários o utilizem como um motor de jogos, já que ele provê funções para tal. O seu formato é de uma aplicação *desktop* que é fornecida tanto para *Windows*, quanto para *Mac* e *Linux*, e sua documentação está tem acesso disponível na internet.

2.4 Construct

O *Construct* é o primeiro motor de jogos 2D desta lista. Ele é propriedade da empresa *Scirra*, e, assim como o *Unity* e o *Unreal*, possui versões pagas e gratuitas. Diferente das aplicações mencionadas anteriormente, porém, o *Construct* é um programa com interface gráfica executável em navegadores *web*. Assim, é possível usá-lo em diferentes sistemas operacionais. Por fim, a linguagem de programação

¹⁴ <https://ffvii-remake-intergrade.square-enix-games.com>

¹⁵ <https://pubg.com>

¹⁶ <https://borderlands.2k.com/pt-BR>

¹⁷ https://www.bandainamcoent.com/pt_br/games/tekken-7

¹⁸ <https://www.fortnite.com/?lang=pt-BR>

usada nesta *engine* é o *JavaScript* e tutoriais e documentação sobre como usá-la podem ser encontrados *online*.

2.5 GameMaker

O *Game Maker* é um motor proprietário da empresa *YoYo Games* focado em jogos 2D, assim como o *Construct*. O motor consiste em uma aplicação com interface gráfica que permite o desenvolvimento por programação visual ou através de sua linguagem especializada chamada de *GameMaker Language*. Semelhantemente ao *Unity*, *Unreal* e *Construct*, possui diferentes planos de uso, gratuitos e não gratuitos. Alguns exemplos de jogos feitos com o *GameMaker* são: *Undertale*¹⁹, *Forager*²⁰ e *Chicory: A Colorful Tale*²¹.

2.6 Godot

O *Godot* é um motor de código aberto tanto para jogos 2D quanto 3D. Ele é disponibilizado sob a licença permissiva MIT e recebe a contribuição de desenvolvedores independentes. Este *software* trata-se de um motor gratuito que oferece uma aplicação com interface gráfica para o uso dos desenvolvedores para a criação de seus jogos. O *Godot* tem uma linguagem de programação própria chamada *GDScript*, que tem uma sintaxe muito parecida com a linguagem *Python*. Entretanto, além dessa linguagem é possível utilizar C#, C++, ou até mesmo uma programação visual. O suporte com documentações e fóruns *online* para esse motor é maior com o *GDScript*, por ser a linguagem mais usada com a *engine*. Por fim, este motor permite que seus projetos sejam exportados tanto para *desktop* quanto para *mobile* e *web*.

¹⁹ <https://undertale.com>

²⁰ <https://hopfrogsa.net/forager>

²¹ <https://chicorygame.com>

Quadro 1: Comparação entre os principais motores do mercado em 2022

	Figure Engine	Unity	Unreal	Blender	Construct	GameMaker	Godot
Proprietário	Licença pública	<i>Unity Technologies</i>	<i>Epic Games</i>	Licença pública	<i>Scirra</i>	<i>YoYo Games</i>	Licença pública
Licença	MIT	Proprietária	Proprietária	GNU GPL	Proprietária	Proprietária	MIT
Alvo	2D	2D & 3D	2D & 3D	3D	2D	2D	2D & 3D
Preço	Gratuito	Vários planos, um grátis	Vários planos, um grátis	Gratuito	Vários planos, um grátis	Vários planos, um grátis	Gratuito
Tipo de software	Biblioteca	Desktop	Desktop	Desktop	Desktop	Desktop	Desktop

Fonte: Autor

3. FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são abordados os conceitos, definições e tecnologias que foram utilizados no desenvolvimento deste trabalho, além das motivações pelas quais elas foram escolhidas para o desenvolvimento deste projeto. A primeira seção fala sobre a linguagem *TypeScript*, enquanto que a segunda traz uma introdução sobre a API *Canvas*²².

3.1 A Linguagem TypeScript

A linguagem *TypeScript* é uma extensão da linguagem *JavaScript*, que foi desenvolvida com o objetivo de trazer recursos adicionais aos programadores e melhorar a manutenção e desenvolvimento de aplicações escritas em *JavaScript* (BIERMAN, ABADI, TORGERSEN, 2014).

Uma das principais características da *TypeScript* é a adição de tipagem estática às variáveis. Isso significa que nela é possível definir explicitamente o tipo de dados que uma variável pode armazenar, o que traz benefícios significativos em relação à detecção de erros e à manutenção do código. Ao se definir a tipagem das variáveis, é possível identificar erros de tipo durante o processo de compilação, antes mesmo de executar o código, evitando assim falhas comuns que podem

²² https://developer.mozilla.org/pt-BR/docs/Web/API/Canvas_API

ocorrer em tempo de execução quando se utiliza a linguagem *JavaScript* pura (BIERMAN, ABADI, TORGERSEN, 2014).

A *TypeScript* também traz recursos avançados, como a inferência de tipos, que permite que o compilador determine automaticamente o tipo de uma variável com base no contexto no qual ela é utilizada. Isso reduz a necessidade de especificar tipos explicitamente em determinados casos, tornando a escrita do código mais concisa e rápida.

A tipagem das variáveis na *TypeScript* permite que os programadores tenham um código mais robusto, pois podem definir tipos específicos para as variáveis, como números, *strings*, booleanos, *arrays* e até mesmo tipos personalizados. Isso proporciona uma melhor compreensão do código e facilita a colaboração em projetos de equipe, além de melhorar a manutenção do código ao longo do tempo.

Outro recurso importante da *TypeScript* é o suporte a recursos de orientação a objetos, como classes, interfaces e herança. Com isso, é possível criar estruturas mais complexas e organizadas, facilitando o desenvolvimento e a reutilização de código. Esses recursos de orientação a objetos e tipagem trazidos pela *TypeScript* são os principais motivos responsáveis pelo uso dessa linguagem na *Figure Engine*, visto que no contexto de desenvolvimento de jogos a orientação a objetos é muitas vezes preferível.

Além da orientação a objetos, a *Typescript* também traz recursos que oferecem suporte a outros paradigmas de programação, como a programação funcional, o que é uma ótima característica para linguagem escolhida para este projeto, visto que isso dá mais liberdade ao desenvolvedor usuário sobre como ele gostaria de programar. Outras propriedades dessa linguagem que motivaram sua escolha para este projeto foram sua segurança de tipos (*type safety*), o que promove uma experiência de desenvolvimento mais agradável, e o fato de ser uma linguagem que pode rodar em navegadores, o que é uma grande vantagem para que os jogos feitos possam ser usados em diferentes plataformas. Finalmente, no caso dos usuários que não querem trabalhar com a tipagem estática da linguagem por algum motivo, devido à flexibilidade da *Typescript*, eles podem fazê-lo, o que foi outro ponto para a escolha dessa linguagem para este projeto.

Por fim, para utilizar a linguagem em questão, é possível instalar o seu compilador globalmente por meio de um gerenciador de pacotes, como o NPM

(*Node Package Manager*). Além disso, pode-se criar um arquivo de configuração (*tsconfig.json*) para definir as opções de compilação e configuração do projeto.

3.2 A API Canvas

A API *Canvas* é uma ferramenta que pode ser utilizada com a linguagem *JavaScript* por meio do uso do elemento *canvas*, do HTML, que passou a ser disponibilizado na versão 5 da linguagem. Essa API permite a renderização de imagens e gráficos dinâmicos por meio do uso do *JavaScript*. O *canvas* é utilizado como uma área de desenho virtual, na qual os elementos gráficos podem ser criados, manipulados e exibidos de forma altamente customizável (DA CRUZ SILVA, 2016).

Uma das principais aplicações da API *Canvas* é a criação de animações que podem ser controladas de maneira dinâmica com o código *JavaScript* associado. Outra aplicação importante da API *Canvas* é a visualização de dados. Sendo ela uma API de manipulação de elementos gráficos, com ela é possível criar gráficos dinâmicos e interativos para exibir informações de maneira visualmente atraente. Através da manipulação dos elementos gráficos no *canvas* é possível representar dados de diferentes formas, como gráficos de barras, gráficos de linhas ou gráficos de pizza.

Outra possibilidade que a API *Canvas* abre é a de manipulação de fotos e vídeos. Com ela, pode-se carregar imagens e vídeos no *canvas* e realizar manipulações sobre eles aplicando filtros, recortes, redimensionamentos e ajustes diversos.

Além das animações, essa API também é amplamente utilizada na criação de jogos. Usando ela como base, é possível criar ambientes interativos e imersivos para os jogadores. Isso junto com sua simplicidade e facilidade de uso configuram os motivos pelos quais optou-se pelo uso dessa API como fundamento na construção da biblioteca *Figure*.

Para utilizar a API *Canvas*, é necessário obter uma referência a um elemento *canvas* através da linguagem *JavaScript*. Isso pode ser feito acessando o elemento por meio de seu identificador ou por seletores DOM. Uma vez obtida a referência ao elemento *canvas*, é necessário chamar o seu método *getContext* para obter um objeto de contexto, que permite a manipulação gráfica do elemento. Existem diferentes contextos disponíveis, como 2D (*2D rendering context*) e 3D (*WebGL*

rendering context), cada um oferecendo funcionalidades específicas para diferentes propósitos. Neste trabalho o contexto 2D foi usado, já que a *Figure* tem como propósito possibilitar a construção de jogos nessa dimensão.

Em resumo, a API *Canvas* é uma ferramenta poderosa que permite a criação de gráficos, animações, jogos, visualização de dados e manipulação de mídias no HTML. Com ela, é possível aproveitar todo o potencial do *JavaScript* para criar experiências visuais ricas e interativas, tornando-a uma opção versátil e amplamente utilizada no desenvolvimento web.

A Figura 2 mostra um exemplo de utilização do *Canvas* na renderização de imagens. Nesse exemplo, foram usadas imagens autorais e as funcionalidades providas pela API.

Figura 2: Renderização no Canvas



Fonte: Autor

4. A SOLUÇÃO PROPOSTA

Este capítulo descreve em detalhes o desenvolvimento da *Figure Engine*, a biblioteca proposta por este trabalho. Nele, são apresentados seus potenciais usuários, os requisitos que deveria atender e a arquitetura que seguiria. Posteriormente, descreve-se como ocorreu a implementação do estudo de caso desenvolvido em conjunto com a biblioteca. Por fim, a última seção apresenta a implementação da ferramenta e os passos seguidos para sua finalização.

4.1 Stakeholders

Os stakeholders do sistema produzido neste TCC podem ser resumidamente descritos como desenvolvedores de jogos. No entanto, há três grupos principais de desenvolvedores para os quais este trabalho visa agregar maior valor: desenvolvedores de jogos que fazem *engines* autorais, desenvolvedores que buscam criar protótipos de jogos simples e desenvolvedores de jogos que precisam trabalhar em computadores não potentes, mas que possuem acesso a navegadores *web*. A seguir, cada um desses grupos será explicado mais detalhadamente.

O primeiro grupo, composto por desenvolvedores de jogos que fazem *engines* autorais, geralmente opta por usar tecnologias próprias devido à falta de flexibilidade dos motores já disponíveis no mercado. Esse cenário, no qual os indivíduos desenvolvem motores próprios, não é objetivamente ruim, mas pode ser prejudicial aos desenvolvedores que querem simplesmente produzir jogos e se veem impedidos pela falta de um motor adequado. Para auxiliar esse grupo, a *Figure Engine* foi criada com o objetivo de prover a esses desenvolvedores um motor básico que eles possam estender com a flexibilidade que os motores mais avançados não oferecem.

Outro grupo de *stakeholders* do sistema produzido com este trabalho são os desenvolvedores de jogos que visam prototipar ideias simples. Pensando também nesse grupo de desenvolvedores, optou-se por criar um motor com funcionalidades básicas, suficientes para a criação de protótipos de jogos convencionais, suprimindo a necessidade desse grupo interessado.

Finalmente, para suprir as necessidades do último grupo de interessados, composto por desenvolvedores de jogos que precisam trabalhar com computadores não potentes mas que possuam pelo menos acesso a navegadores *web*, o motor desenvolvido neste trabalho foi implementado como uma biblioteca em *TypeScript*

de baixo espaço de armazenamento, facilmente obtível através do NPM, para que esses desenvolvedores possam criar seus jogos apesar de suas limitações de hardware.

4.2 Requisitos Funcionais

Os requisitos funcionais que deveriam ser satisfeitos pela biblioteca desenvolvida neste TCC foram definidos com base em como outros motores solucionam os problemas necessários para o desenvolvimento de jogos, na experiência pessoal do autor como desenvolvedor de jogos e nas informações presentes nos artigos referenciados por este trabalho. São eles:

- **Definir a resolução do jogo (RF1):** a biblioteca deveria permitir a configuração da resolução do jogo na tela do usuário. Além disso, preferencialmente, ela deveria permitir a adaptabilidade dessa resolução e a visualização do jogo em tela cheia;
- **Deteccção de entradas do usuário (RF2):** esperava-se que a biblioteca provesse formas de detectar as entradas de dados do jogador, para que ele pudesse controlar o jogo;
- **Configurações de Frames Por Segundo (FPS) (RF3):** a biblioteca deveria permitir a definição de quantos FPS máximos o jogo deve ter;
- **Construção de fases (RF4):** a biblioteca deveria possibilitar a construção de fases para o jogo. Essas fases poderiam conter diferentes recursos, tais como objetos, *sprites*, cenários, sons e uma câmera para controle da visibilidade das fases;
- **Customização da câmera (RF5):** a biblioteca deveria permitir a configuração das câmeras das fases, seus tamanhos e comportamentos;
- **Criação de objetos (RF6):** a biblioteca deveria permitir a criação e customização de objetos do jogo e seus comportamentos, além da definição de sua representação gráfica (*sprite*);
- **Carregamento de fases (RF7):** a biblioteca deveria permitir que os recursos das fases fossem carregados antes de seu uso, usando-os no jogo somente quando prontos;

- **Importação de recursos (RF8):** a biblioteca deveria permitir a importação de recursos externos de imagem e áudio para uso no jogo, por meio de *sprites*, cenários, efeitos sonoros e música.

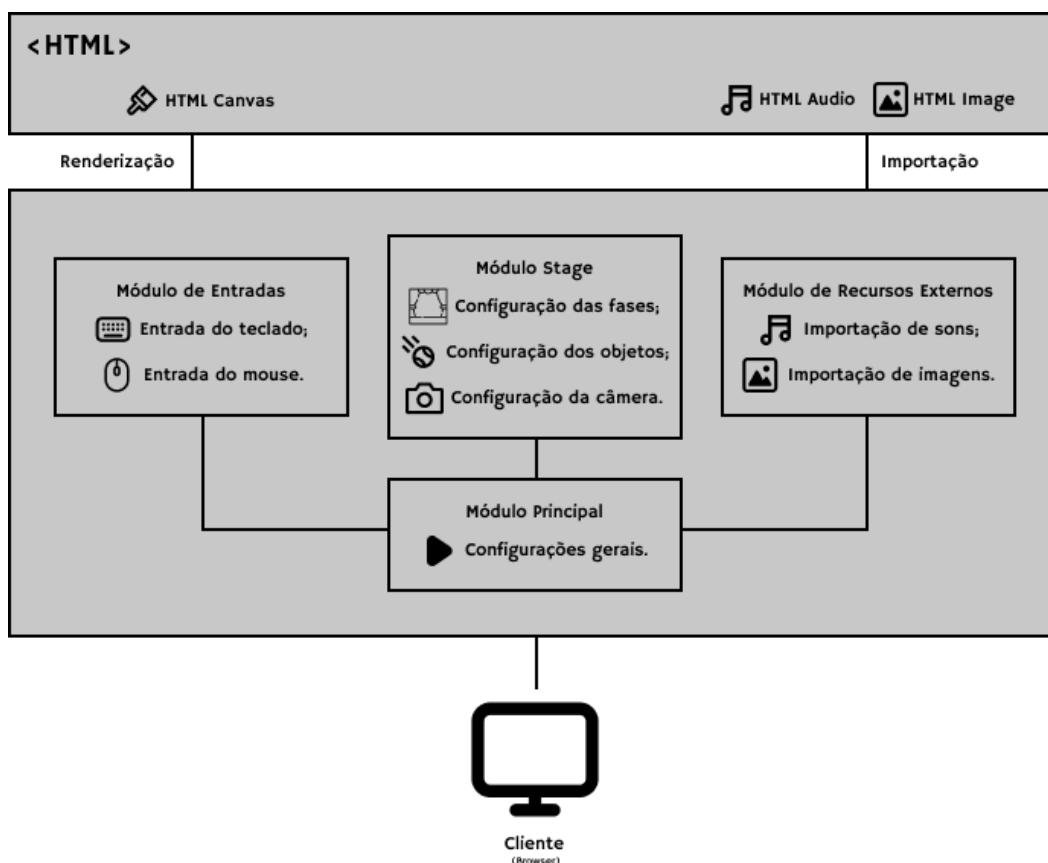
4.3 Arquitetura

A arquitetura usada para a implementação da biblioteca é exibida graficamente na Figura 3. Sua idealização foi pensada de modo a separar coesamente os principais recursos funcionais definidos para a sua implementação. Isso foi importante para deixar o software resultante desacoplado, permitindo que seu desenvolvimento possa ser continuado a longo prazo por outros desenvolvedores interessados.

Esse desenho da arquitetura também tem como base informações dos artigos presentes nas referências deste trabalho e a experiência do autor como desenvolvedor de jogos e usuário de outros motores. Na arquitetura da Figura 3, mostra-se a divisão dos principais módulos da biblioteca, bem como as tecnologias externas que foram utilizadas para a sua implementação, como o elemento *canvas* do HTML para a renderização do jogo para o jogador, e alguns outros elementos, como o elemento *img* e o elemento *audio* para importação de arquivos. Os módulos internos da biblioteca são os seguintes:

- **Módulo principal:** este módulo é responsável pelas configurações gerais do jogo, como a definição de seu tamanho na tela e a configuração da quantidade de FPS máxima esperada;
- **Módulo de entradas:** este módulo tem como função permitir que o desenvolvedor detecte as entradas fornecidas pelo jogador através dos dispositivos disponíveis, como o teclado e o *mouse*;
- **Módulo Stage (ou módulo das fases):** o intuito deste módulo é permitir a customização das fases do jogo e garantir que o jogo carregue os recursos necessários para reproduzir a fase;
- **Módulo de recursos externos:** este módulo tem como objetivo permitir que o desenvolvedor importe recursos de imagem e áudio para dentro do jogo, a fim de renderizá-los ou reproduzi-los para o jogador.

Figura 3: Arquitetura do projeto



Fonte: Autor

4.4 Estudo de caso

Além do desenvolvimento da *Figure Engine*, durante a implementação deste trabalho foi implementado um estudo de caso para acompanhar o crescimento e amadurecimento do *software* principal. O exemplo utilizado foi um protótipo de jogo simples, visto que este estudo só era destinado à validação da biblioteca desenvolvida. Na seção da implementação será discorrido como este estudo de caso foi se desenvolvendo junto do projeto principal, de modo a possibilitar um acompanhamento mais visual das funcionalidades ao longo do ciclo de vida do projeto.

O estudo de caso desenvolvido consistiu em um protótipo de jogo, chamado de *The Ghost Game*, que tem uma visão *top down* (tipo de perspectiva comum em alguns jogos que mostra o mundo de cima para baixo) e um fantasma como personagem principal. O jogo não tem um final definido, possibilitando ao jogador apenas a funcionalidade de andar para onde ele desejar. Com o desenvolvimento desse pequeno projeto foi possível constatar a funcionalidade correta dos requisitos

da biblioteca criada, tais como a captação de entradas do jogador, a renderização de *sprites* e cenários na tela, o fluxo de atualização de quadros e a câmera que segue o personagem.

4.5. Implementação

Esta seção descreve como o projeto proposto por este trabalho foi desenvolvido, e quais foram os processos e ferramentas utilizadas para tal. Para a sua implementação, optou-se por utilizar uma metodologia baseada no desenvolvimento ágil, dividida por *sprints* com prazos variados que tinham como escopo a implementação de um dos requisitos funcionais. Durante o desenvolvimento, realizou-se também a implementação de testes unitários utilizando a biblioteca *Jasmine* do *JavaScript*. Além disso, ao longo do processo, o código foi documentado para que os interessados em utilizá-lo possam ter uma fonte de pesquisa legível para aprender a como usar a biblioteca construída.

Para acompanhar o versionamento do projeto, utilizou-se o programa *Git* juntamente com o *GitHub*. O código fonte do projeto está atualmente disponível em um repositório do *GitHub*²³. As próximas seções descrevem com mais detalhes como foram as *sprints* de desenvolvimento, quais foram seus escopos e como o estudo de caso feito em paralelo foi crescendo e amadurecendo com a implementação.

4.5.1 Sprint 1: Quadro e FPS

A primeira *sprint* teve como escopo dois requisitos funcionais: o RF1 (definir a resolução do jogo) e o RF3 (configurações de FPS). Essa etapa de desenvolvimento durou aproximadamente uma semana e trouxe os recursos mais básicos da biblioteca. Com a versão do produto gerada nesse período, já se podia definir uma área onde o jogo seria executado dentro da página HTML, e os FPS podiam ser atualizados, mas ainda não havia recursos para criar jogos interativos.

O intuito da versão gerada aqui era facilitar as configurações iniciais de um jogo, que, sem essa biblioteca, envolveriam a criação e configuração manual da resolução de um elemento *canvas* do HTML e o desenvolvimento de um mecanismo que executasse uma função a cada quadro desejado. Para exemplificação, na

²³ <https://github.com/nadjjel/figure-engine>

Figura 4 há um trecho de código que mostra como um *canvas* de resolução 1280 por 720 *pixels* seria anexado a uma página HTML, e como se realizaria a execução contínua de alguma funcionalidade repetidamente com uma determinada quantidade de FPS (neste exemplo, a funcionalidade seria exibir no console qual é o número do quadro atual) sem o uso da biblioteca.

Figura 4: Trecho de código configurando um jogo sem a Figure Engine

```
1 // O elemento Canvas é criado e configurado.
2 const canvas = document.createElement("canvas");
3 canvas.style.backgroundColor = "black";
4 canvas.width = 1280;
5 canvas.height = 720;
6 document.body.appendChild(canvas);
7
8 // É declarada uma função que executaria a uma determinada taxa
9 // de FPS e contaria cada execução, exibindo-a no console.
10 let frame = 0;
11
12 function update() {
13     frame++;
14
15     console.log(frame)
16
17     requestAnimationFrame(update);
18 }
19
20 // Começa a execução.
21 requestAnimationFrame(update);
```

Fonte: Autor

Utilizando o código desenvolvido, o trabalho necessário para reproduzir essa mesma funcionalidade seria reduzido em termos de linhas de código e de quantidade de código padrão para a configuração do *canvas*. A seguir, na Figura 5 está um exemplo de como esse mesmo efeito poderia ser alcançado com o uso do motor nesta etapa de seu desenvolvimento.

Figura 5: Trecho de código configurando um jogo com a Figure Engine

```
1 // Importa-se os dados necessários da biblioteca.
2 import { Game } from "../../node_modules/figure-engine/dist/index.js";
3
4 // Cria-se um jogo.
5 const game = new Game();
6
7 // Define-se o que deve ocorrer a cada quadro.
8 game.onUpdate = () => {
9   console.log(game.getCurrentFrame());
10 }
11
12 // Começa a execução.
13 game.start();
```

Fonte: Autor

O desenvolvimento do estudo de caso durante essa versão da biblioteca foi realizado utilizando os recursos disponibilizados por ela. Com isso, o protótipo já contava com uma área de cor de fundo preta onde o jogo seria executado e com a execução cíclica dos quadros (*frames*) do jogo.

4.5.2 Sprint 2: Entrada e Objetos

A segunda *sprint* do desenvolvimento, tomando como base a primeira, teve como propósito o desenvolvimento de dois requisitos funcionais um pouco mais complexos: o RF2 (Detecção de entradas do usuário) e o RF6 (Criação de objetos). Como o escopo lidava com conceitos mais complicados, a duração desse período foi estendida em relação ao anterior, totalizando aproximadamente três semanas.

A implementação da detecção de entradas do usuário foi realizada utilizando como base os eventos *onkeydown* e *onkeyup* oferecidos pelo DOM, de forma que as informações ficassem armazenadas em um objeto responsável por facilitar seu acesso em momentos diversos da execução do jogo. Sem esse armazenamento, as informações de entrada estariam disponíveis apenas no contexto dos eventos do DOM mencionados, o que as tornaria mais difíceis de acessar em escopos fora desses trechos. A Figura 6, a seguir, mostra um trecho de código exemplificando como é feito o acesso à entrada de dados do teclado, utilizando a abstração fornecida pela ferramenta nessa versão. Para exemplificação, os dados sobre o

estado atual do *input* da tecla *enter* do teclado são acessados e exibidos no console a cada quadro.

Figura 6: Trecho de código acessando input com a Figure Engine

```
1 // Armazena o objeto de input do teclado, que traz informações úteis
2 // a respeito das entradas atuais.
3 const keyboardInput = game.getKeyboardInput();
4
5 // Acessa dados de input da tecla Enter a cada quadro.
6 game.onUpdate = () => {
7   console.log([
8     keyboardInput.isPressed("Enter"),
9     keyboardInput.isHeld("Enter"),
10    keyboardInput.isReleased("Enter")
11  ]);
12 }
```

Fonte: Autor

A versão produzida nessa *sprint* também permitia que os desenvolvedores criassem e customizassem objetos para seus jogos. Para isso, foi criada uma classe abstrata que representa um objeto genérico, de modo que, para definir os seus próprios objetos, o desenvolvedor precisa estender essa classe e sobrescrever seus métodos para definir seu comportamento. A seguir na Figura 7 há um exemplo que mostra uma classe customizada que representa um fantasma em um jogo. Esta classe estende a classe *GameObject*, definindo sua própria cor de fundo como branca para que suas instâncias sejam visíveis no *canvas* do jogo. Além disso, no método *onUpdate* sobrescrito por esta classe, é definido um comportamento que faz com que ela responda a entradas do usuário através das setas do teclado.

Figura 7: Trecho de código definindo um objeto com a Figure Engine

```

1 // Estende-se a classe que define os objetos no jogo de modo a
2 // descrever um objeto customizado.
3 class Ghost extends GameObject {
4
5 // O construtor recebe uma coordenada e um tamanho para aplicar
6 // para o objeto construído.
7 constructor(coord, size) {
8     super(coord, size);
9
10 // Define-se que a cor de fundo dos objetos criados
11 // com essa classe será branca.
12 this.setColor(ColorFactory.createWhite());
13 }
14
15 onStart() {}
16
17 // Descreve o comportamento desse objeto a cada quadro do jogo.
18 // Nesse caso, ele acessa o input das teclas direcionais do teclado,
19 // checando quais delas estão pressionadas no momento e fazendo
20 // com que a posição deste objeto mude, o movendo para a direção
21 // para onde as setas pressionadas apontam.
22 onUpdate() {
23     const keyboardInput = game.getKeyboardInput();
24
25     const hInput = keyboardInput.isHeld("ArrowRight") - keyboardInput.isHeld("ArrowLeft");
26     const vInput = keyboardInput.isHeld("ArrowDown") - keyboardInput.isHeld("ArrowUp");
27
28     const speed = new Vector2(hInput, vInput);
29
30     this.setX(this.getX() + speed.getX());
31     this.setY(this.getY() + speed.getY());
32 }
33
34 onDraw(ctx) {}
35 onStop() {}
36
37 }

```

Fonte: Autor

Nesta versão, devido à ausência de uma maneira de organizar os objetos em um único lugar para que o jogo pudesse atualizá-los automaticamente quando necessário, era preciso chamar manualmente os métodos de atualização dos objetos instanciados através dos métodos de atualização da própria instância do jogo. Assim, para que o código do objeto tivesse efeito no jogo, era necessário implementar um comportamento semelhante ao mostrado no código da Figura 8 a seguir:

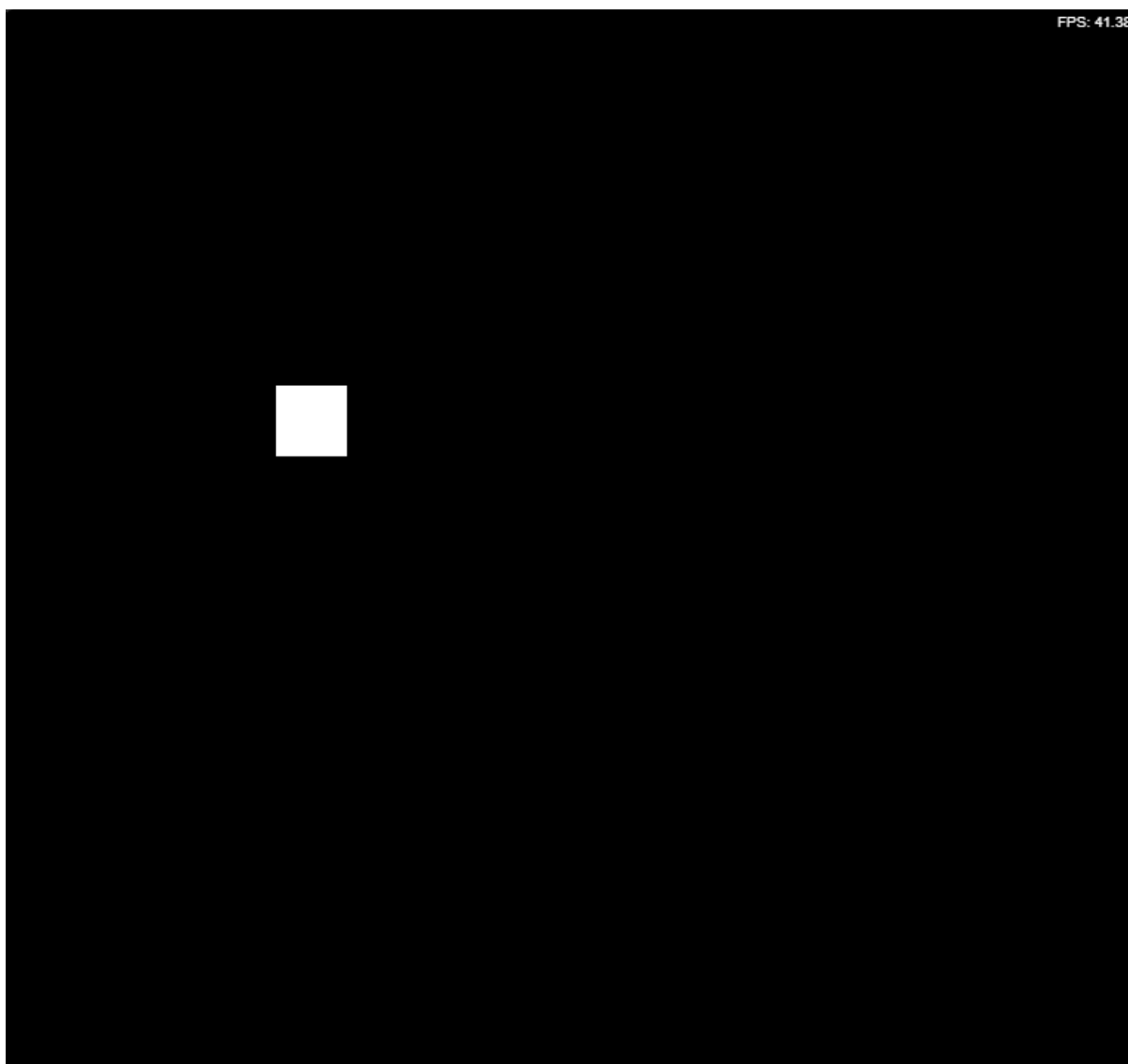
Figura 8: Trecho de código atualizando um objeto manualmente com a *Figure Engine*

```
1 // Cria-se o objeto em questão, com coordenadas (100, 100)
2 // e tamanho de 80 pixels por 80 pixels.
3 const ghost = new Ghost(
4   new Vector2(100, 100),
5   new Vector2(80, 80)
6 );
7
8 // Atualiza-se o objeto ghost a cada quadro do jogo.
9 game.onUpdate = () => {
10  ghost.update();
11 }
12
13 // Desenha-se o objeto ghost a cada quadro do jogo.
14 game.onDraw = (ctx) => {
15  ghost.draw(ctx);
16 }
```

Fonte: Autor

O estudo de caso que estava sendo realizado em conjunto com a biblioteca nessa etapa de seu desenvolvimento utilizou essas novas funcionalidades, o que permitiu que ele adquirisse interatividade com o jogador através das entradas do teclado pudesse renderizar o *feedback* dessas entradas no *canvas*. É possível visualizá-lo na Figura 9, a seguir.

Figura 9: Estudo de caso após a segunda sprint ter sido desenvolvida



Fonte: Autor

4.5.3 Sprint 3: Construção de Fases

Esta etapa do projeto teve como objetivo apenas a implementação de um dos requisitos funcionais: o RF4, que consistia na Construção de Fases, e durou cerca de duas semanas. No final desse período, a versão resultante já trazia recursos que permitiam aos desenvolvedores dividirem os seus jogos em fases diferentes. Isso ajudaria a manter uma maior organização e estrutura nos jogos criados, além de permitir um maior desacoplamento no uso dos recursos externos de cada fase, o que seria implementado na *sprint* seguinte. Um dos benefícios trazidos por esse novo sistema de organização em fases é que não seria mais necessário atualizar manualmente os objetos criados; bastaria adicioná-los a uma fase específica e

adicionar essa fase ao jogo em questão. Depois disso, seria necessário apenas que essa fase fosse selecionada para iniciar sua execução. A Figura 10, a seguir, contém um trecho de código que mostra como isso poderia ser utilizado pelos desenvolvedores para que eles pudessem criar suas fases.

Figura 10: Trecho de código criando uma fase com a Figure Engine

```
1 // Cria-se uma fase.
2 const stage = new Stage();
3
4 // Adiciona-se a fase criada ao jogo criado.
5 game.addFirstStage(stage);
6
7 // Adiciona-se o objeto criado a fase criada.
8 stage.addFirstObject(ghost);
9
10 // Seleciona a fase criada.
11 game.selectNextStage();
```

Fonte: Autor

4.5.4 Sprint 4: Importação de Recursos

Na *sprint 4*, foi estabelecido que seria implementado o requisito RF8, que consistia na Importação de Recursos, um dos requisitos essenciais para um jogo com gráficos e sons. Para que esses elementos fossem exibidos, era necessário importá-los do sistema de arquivos. Essa etapa durou duas semanas e, ao final, os programadores que usavam a biblioteca já podiam importar imagens e sons de arquivos armazenados no computador para dentro do jogo.

Para realizar a importação de arquivos de imagem e áudio, a biblioteca utilizou, respectivamente, os elementos *img* e *audio* do HTML. Sem a abstração realizada pela ferramenta, a importação de arquivos já poderia ser realizada, porém não seria possível saber se os arquivos desejados estavam carregados ou não sem a implementação de um mecanismo mais complexo. Isso poderia dificultar o gerenciamento das fases, que ficariam suscetíveis a tentar utilizar recursos não carregados.

O trecho de código na Figura 11, a seguir, mostra como a importação de uma imagem do sistema de arquivos do computador poderia ser feita sem o uso desta versão do motor.

Figura 11: Trecho de código importando recursos sem a Figure Engine

```
1 // Cria-se uma imagem HTML a qual é adicionado um caminho onde o arquivo
2 // a importar está localizado.
3 const largeImage = document.createElement("img");
4 largeImage.src = "../../assets/large_image.jpg";
```

Fonte: Autor

Utilizando a nova funcionalidade de importação do motor, essa ação seria um pouco diferente, visto que primeiro seria criado um recurso com o caminho do arquivo desejado, e depois esse recurso seria adicionado a um gerenciador de recursos responsável por lidar com o manuseio deles.

A seguir, a Figura 12 exibe um trecho de código que mostra como a importação de um arquivo de imagem seria realizada, em comparação ao exemplo anterior.

Figura 12: Trecho de código importando recursos com a Figure Engine

```
1 // Cria-se um recurso de imagem com o caminho do arquivo.
2 const largeImageResource = new ImageResource("../../assets/large_image.jpg");
3
4 // Adiciona-se esse recurso ao gerenciador de arquivos, com o nome desejado.
5 ResourceManager.addResource("large-image", largeImageResource);
```

Fonte: Autor

Com essa implementação, era possível trazer os recursos importados para dentro do sistema, mas ainda não era possível carregá-los, o que impossibilitava o uso bem-sucedido de alguns arquivos grandes. Esse problema seria corrigido na versão seguinte.

4.5.5 Sprint 5: Carregamento de Fases

Depois de permitir a importação de recursos externos, era necessário lidar com o carregamento desses recursos para evitar erros de acesso a recursos não prontos. Essa *sprint* abordou este problema, que já estava descrito nos requisitos

funcionais como RF7. Assim como a iteração anterior, esta *sprint* durou duas semanas e, ao seu término, era possível usar recursos externos independentemente de seu tamanho e complexidade de carregamento, o que proporcionava aos desenvolvedores uma liberdade criativa maior.

A implementação dessa funcionalidade possibilitou aos desenvolvedores garantir que suas fases não fossem executadas sem que os recursos marcados como necessários fossem carregados. O código da Figura 13 a seguir mostra um exemplo de como o recurso importado no trecho de código anterior poderia ser carregado e utilizado como *sprite* de um objeto.

Figura 13: Trecho de código carregando recursos com a Figure Engine

```
1 // Cria-se um sprite que utiliza o recurso "large-image".
2 const largeImageSprite = new Sprite(largeImageResource);
3
4 // Estende-se a classe que define os objetos no jogo de modo a
5 // descrever um objeto customizado.
6 class Ghost extends GameObject {
7
8     constructor(coord, size) {
9         super(coord, size);
10
11         // Define-se o sprite desse objeto como sendo o largeImageSprite,
12         // que utiliza o recurso "large-image".
13         this.setSprite(largeImageSprite);
14
15         // No objeto, deve-se usar o método usesResource para "avisar" ao
16         // motor quais recursos são necessários para que ele possa executar.
17         // Desse modo, as fases que utilizam esse objeto só executarão quando
18         // esse recurso estiver carregado.
19         this.usesResource("large-image");
20     }
21
22     onStart() {}
23     onUpdate() {}
24     onDraw(ctx) {}
25     onStop() {}
26
27 }
```

Fonte: Autor

Na Figura 14, é mostrado como o estudo de caso desenvolvido estava quando essa versão foi implementada, a título de demonstração.

Figura 14: Versão do estudo de caso após a quinta sprint



Fonte: autor

4.5.6 Sprint 6: Câmera e Customização

O último requisito implementado foi o RF5 - Customização da câmera - considerado o menos crucial do projeto, visto que é um recurso não necessário para alguns jogos 2D. O período de implementação dessa funcionalidade também durou aproximadamente duas semanas e finalizou o desenvolvimento das partes da biblioteca proposta por este trabalho. No fim desta sprint, o desenvolvedor que

usava a *Figure Engine* já podia criar jogos 2D um pouco mais avançados, com uma câmera que pudesse seguir algum personagem ou realizar outras ações que o programador desejasse.

A Figura 15 a seguir mostra um trecho de código que ilustra como criar uma câmera que siga um objeto, mantendo-o centralizado na tela do jogo. Considera-se que o objeto a ser seguido seja o *ghost* e que a fase à qual ela será adicionada já esteja criada.

Figura 15: Trecho de código criando câmera com a Figure Engine

```
1 // Cria-se uma câmera seguidora que sempre busca deixar o objeto
2 // passado na propriedade "target" como alvo no centro da tela.
3 const camera = new FollowingCamera({
4   target: ghost,
5   x: 0,
6   y: 0,
7   width: game.getWidth(),
8   height: game.getHeight()
9 });
10
11 // Adiciona-se a câmera criada à fase criada.
12 stage.setCamera(camera);
```

Fonte: Autor

5. CONCLUSÃO

No contexto atual da indústria brasileira de jogos, o principal meio utilizado para programá-los é o motor de jogos, *software* feito justamente para facilitar o desenvolvimento desse tipo de programa. Entretanto, ainda há quem prefira fazer seus jogos a partir de motores próprios. Isso significa que essas pessoas programam, além do jogo em si, o seu motor, o que não é necessariamente ruim. Entretanto, esta tarefa poderia ser simplificada se houvesse um programa que esse público pudesse utilizar como apoio na construção de seus próprios motores.

Para resolver esta limitação, este trabalho propôs a biblioteca *Figure Engine*, um novo motor de jogos que possibilita aos programadores interessados usá-la como uma ferramenta de apoio no desenvolvimento de seus jogos ou adaptá-la por conta própria para que ela melhor se adeque aos interesses de seus projetos.

A biblioteca desenvolvida permite a criação de jogos básicos e protótipos, o que foi validado através do estudo de caso implementado ao longo do trabalho. A biblioteca está disponível publicamente em um repositório do *GitHub*²⁴ e outro do *NPM*²⁵, para que desenvolvedores interessados possam usá-la como base para seus projetos de jogos ou motores próprios. Por fim, o estudo de caso feito para validar a biblioteca também está disponível publicamente²⁶.

Tendo em vista que a biblioteca desenvolvida oferece uma estrutura básica de motor de jogo, há ilimitadas oportunidades de expansão que podem ser realizadas em trabalhos futuros. Algumas sugestões de melhorias ou adições que estão presentes em muitas engines (ALEEM, CAPRETZ, AHMED, 2016) e poderiam ser incorporadas à biblioteca incluem: a criação de um sistema de simulação de física, que facilitaria a implementação de mecanismos como o de colisão de objetos ou elementos baseados em leis físicas do mundo real; o desenvolvimento de um sistema que permita a criação de jogos multijogador através de comunicação na rede; a facilitação da implementação de elementos de interface gráfica através de componentes de UI pré-disponibilizados; e a validação da documentação criada em paralelo à biblioteca, de modo a comprovar sua eficácia em permitir que desenvolvedores interessados possam se envolver com o projeto de maneira conveniente.

²⁴ <https://github.com/nadjiel/figure-engine>

²⁵ <https://www.npmjs.com/package/figure-engine>

²⁶ <https://github.com/nadjiel/the-ghost-game>

REFERÊNCIAS

ALEEM, Saiqa; CAPRETZ, Luiz Fernando; AHMED, Faheem. Game development software engineering process life cycle: a systematic review. *Journal of Software Engineering Research and Development*, v. 4, p. 1-30, 2016.

BIERMAN, Gavin; ABADI, Martín; TORGERSEN, Mads. Understanding typescript. In: *ECOOP 2014–Object-Oriented Programming: 28th European Conference*, Uppsala, Sweden, July 28–August 1, 2014. *Proceedings 28*. Springer Berlin Heidelberg, 2014. p. 257-281.

BISHOP, Lars; EBERLY, Dave; FINCH, Mark; SHANTZ, Michael; WHITTED, Turner. Designing a PC Game Engine. *IEEE COMPUTER GRAPHICS AND APPLICATIONS*, v. 272, p. 98, 1998.

DA CRUZ SILVA, Rodrigo Cardoso. DESENVOLVIMENTO DE JOGOS PARA A WEB UTILIZANDO O PADRÃO HTML5 E A API CANVAS. *Revista Unifev: Ciência & Tecnologia*, v. 1, n. 1, p. 9-19, 2016.

FORTIM, Ivelise (Org). Pesquisa da indústria brasileira de games 2022. *ABRAGAMES*: São Paulo, 2022.

MENARD, Michelle. *Game development with Unity*. Course Technology Press, 2011.

SALMELA, Tero. *Game development using the open-source Godot Game Engine*. 2022.

ŠMÍD, Antonín. Comparison of unity and unreal engine. *Czech Technical University in Prague*, p. 41-61, 2017.

GLOSSÁRIO

Array: tipo de dados comum nas linguagens de programação que permite armazenar uma lista de valores.

Assembly: Linguagem de programação de baixo nível, muito próxima da linguagem de máquina.

Background: Nos jogos 2D os *backgrounds* são imagens que são normalmente renderizadas no fundo dos níveis para fins decorativos.

Booleano: Tipo de dado comum nas linguagens de programação que armazena como valor ou verdadeiro ou falso.

Classe: Conceito presente nas linguagens de programação orientadas a objetos que permite a representação do modelo de como um objeto deve ser.

C++: Linguagem de programação compilada com base na linguagem C, desenvolvida por Bjarne Stroustrup.

C#: Linguagem de programação multiparadigma desenvolvida pela *Microsoft* baseada em linguagens como o C++ e o Java.

Colisão: Nos jogos colisão diz respeito a detecção de sobreposição entre dois formatos geométricos. Isso é muito usado para implementação de barreiras como paredes ou chão.

Framework: Um conjunto de códigos e metodologias prontos que são usados para acelerar o desenvolvimento de outros *softwares*.

Hardware: Diz respeito a parte física dos equipamentos eletrônicos: aparelhos como mouse, teclado, CPU etc.


Objeto: No contexto desse motor de jogos, um objeto se refere a qualquer conceito no jogo que pode ser programado, sendo ou não aparente ao jogador. Exemplos de objeto podem ser um personagem no jogo, um item coletável, ou até mesmo um sistema abstrato do jogo.

Python: Linguagem de programação interpretada de alto nível muito usada para análise de dados.

Software: Referência a qualquer componente não físico do computador. Normalmente diz respeito aos programas de computador.

Sprite: Sprites são objetos gráficos que podem se mover pela tela. Geralmente são usados nos jogos para representar objetos e personagens e até mesmo elementos de interface.

Tile: *Tiles* dizem respeito a imagens que são utilizadas para compor cenários nos jogos (principalmente os 2D). Essas imagens são utilizadas de um modo similar à ladrilhos (tradução de *tile* em inglês), sendo montadas em uma grade.

	INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DA PARAÍBA
	Campus Cajazeiras - Código INEP: 25008978
	Rua José Antônio da Silva, 300, Jardim Oásis, CEP 58.900-000, Cajazeiras (PB)
	CNPJ: 10.783.898/0005-07 - Telefone: (83) 3532-4100

Documento Digitalizado Ostensivo (Público)

Documento do TCC com folha de aprovação e ficha catalográfica

Assunto:	Documento do TCC com folha de aprovação e ficha catalográfica
Assinado por:	Daniel Sousa
Tipo do Documento:	Tese
Situação:	Finalizado
Nível de Acesso:	Ostensivo (Público)
Tipo do Conferência:	Cópia Simples

Documento assinado eletronicamente por:

- Daniel de Oliveira Sousa, DISCENTE (202112010006) DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS - CAJAZEIRAS, em 23/10/2024 10:15:40.

Este documento foi armazenado no SUAP em 23/10/2024. Para comprovar sua integridade, faça a leitura do QRCode ao lado ou acesse <https://suap.ifpb.edu.br/verificar-documento-externo/> e forneça os dados abaixo:

Código Verificador: 1288456

Código de Autenticação: eaf217fae6

