

UNIVERSIDADE FEDERAL DE OURO PRETO
INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS
DEPARTAMENTO DE COMPUTAÇÃO

STHÉPHANY KAROLINE SOARES DE ARAÚJO TEZZA

TEZZA SPACE: UMA ABORDAGEM DIDÁTICA SOBRE CSS GRID

Ouro Preto
2026

STHÉPHANY KAROLINE SOARES DE ARAÚJO TEZZA

TEZZA SPACE: UMA ABORDAGEM DIDÁTICA SOBRE CSS GRID

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal de Ouro Preto como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Pedro Henrique Lopes Silva

Ouro Preto
2026



MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS
DEPARTAMENTO DE COMPUTAÇÃO



FOLHA DE APROVAÇÃO

Sthéphany Karoline Soares de Araújo Tezza

Tezza Space: Uma Abordagem Didática sobre CSS Grid

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal de Ouro Preto como requisito parcial para obtenção do título de Bacharel em Ciência da Computação

Aprovada em 20 de Julho de 2026.

Membros da banca

Pedro Henrique Lopes Silva (Orientador) - Doutor - Universidade Federal de Ouro Preto
Luan Patrik Silva Pinto (Examinador) - Bacharel - Programa de Pós-Graduação em Ciência da Computação - UFOP
Ederson Naves Fernandes Gonçalves Junior (Examinador) - Bacharel - Programa de Pós-Graduação em Ciência da Computação - UFOP

Pedro Henrique Lopes Silva, Orientador do trabalho, aprovou a versão final e autorizou seu depósito na Biblioteca Digital de Trabalhos de Conclusão de Curso da UFOP em 20/07/2026.



Documento assinado eletronicamente por **Pedro Henrique Lopes Silva, PROFESSOR DE MAGISTERIO SUPERIOR**, em 20/07/2026, às 13:14, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do [Decreto nº 8.539, de 8 de outubro de 2015](#).



A autenticidade deste documento pode ser conferida no site http://sei.ufop.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **1142647** e o código CRC **5C07B7F0**.

À minha mãezinha Eva, que me ensinou a existir com amor. Saudades eternas.

Agradecimentos

Agradeço à minha mãe, Eva, que foi quem me criou e me ofereceu amor, cuidado e suporte em todos os momentos. Muito do que sou hoje é resultado do que aprendi com essa pessoa única e do carinho que sempre recebi. Seus ensinamentos e sua presença continuam me acompanhando em cada passo desta caminhada. Sinto sua falta todos os dias.

Agradeço ao meu orientador, Professor Pedro, pela paciência, pela confiança e por permanecer ao meu lado mesmo em um momento difícil, sem nunca desistir deste trabalho. Sua forma de tratar tudo com leveza e otimismo tornou essa etapa mais suave, e seu apoio foi fundamental para a conclusão deste projeto e para a realização do meu sonho de me formar.

*“Ao verme que primeiro roeu as frias carnes do meu cadáver dedico como saudosa
lembrança estas Memórias Póstumas.”*

Machado de Assis, *Memórias Póstumas de Brás Cubas*.

Resumo

O ensino de *layout* no desenvolvimento Web, especialmente no que se refere ao uso do *Cascading Style Sheets (CSS) Grid*, apresenta desafios relacionados à abstração espacial e à compreensão da relação entre código e comportamento visual. Este trabalho tem como objetivo o desenvolvimento de uma aplicação Web interativa que auxilie no ensino dos conceitos fundamentais do *CSS Grid* por meio da experimentação prática. Na plataforma desenvolvida, busca-se estruturar um ambiente que permita a edição de código com retorno visual imediato, organizar o processo de aprendizagem em fases progressivas divididas entre propriedades dos *Grid Items* e propriedades do *Grid Container*, e integrar edição de estilos, gerenciamento de estado da aplicação e renderização do *layout* em um fluxo coerente. O resultado deste trabalho foi o desenvolvimento da primeira versão funcional de uma aplicação denominada *Tezza Space*, implementada com *React* e *TypeScript*, utilizando uma arquitetura baseada em componentes reutilizáveis e gerenciamento centralizado de estado. O sistema organiza o processo de aprendizagem em vinte fases, distribuídas em dois módulos didáticos, nas quais o usuário edita propriedades de *layout* e acompanha, em tempo real, sua aplicação em um tabuleiro orbital estruturado com *CSS Grid*. A correção dos mecanismos implementados foi verificada por meio de um conjunto de 151 testes automatizados, que alcançaram uma cobertura geral de 97,8%. A análise do desenvolvimento indica que a aplicação sustenta um fluxo completo e consistente entre a edição de regras, a conversão em estilos computáveis e a renderização visual, contemplando tanto as propriedades estruturantes do *Grid Container* quanto as propriedades de posicionamento e alinhamento dos *Grid Items*. Por fim, o *Tezza Space* estabelece uma base técnica e pedagógica consistente para o ensino interativo do *CSS Grid*, apresentando relevância no contexto acadêmico ao propor uma abordagem prática e visual para o aprendizado de *layouts*, além de apontar como trabalhos futuros a validação empírica da proposta junto a usuários reais e a ampliação do conjunto de fases para contemplar desafios de articulação simultânea entre múltiplas propriedades.

Palavras-chave: *Cascading Style Sheets Grid*. Aprendizagem interativa. Desenvolvimento Web.

Abstract

The teaching of layout in Web development, particularly regarding the use of *Cascading Style Sheets (CSS) Grid*, presents challenges related to spatial abstraction and to understanding the relationship between code and visual behavior. This work aims to develop an interactive Web application that supports the teaching of the fundamental concepts of *CSS Grid* through hands-on experimentation. In the developed platform, the goal is to structure an environment that allows code editing with immediate visual feedback, to organize the learning process into progressive phases divided between *Grid Items* properties and *Grid Container* properties, and to integrate style editing, application state management, and layout rendering into a coherent flow. The result of this work was the development of the first functional version of an application called *Tezza Space*, implemented with *React* and *TypeScript*, using an architecture based on reusable components and centralized state management. The system organizes the learning process into twenty phases, distributed across two didactic modules, in which the user edits layout properties and observes, in real time, their application on an orbital board structured with *CSS Grid*. The correctness of the implemented mechanisms was verified through a set of 151 automated tests, which achieved an overall coverage of 97.8%. The analysis of the development indicates that the application sustains a complete and consistent flow between rule editing, conversion into computable styles, and visual rendering, addressing both the structural properties of the *Grid Container* and the positioning and alignment properties of the *Grid Items*. Finally, *Tezza Space* establishes a consistent technical and pedagogical foundation for the interactive teaching of *CSS Grid*, presenting relevance in the academic context by proposing a practical and visual approach to layout learning, in addition to indicating, as future work, the empirical validation of the proposal with real users and the expansion of the set of phases to address challenges involving the simultaneous articulation of multiple properties.

Keywords: *Cascading Style Sheets Grid*. Interactive learning. Web development.

Lista de Figuras

Figura 2.1 – Renderização de elementos de bloco em modo de escrita horizontal.	13
Figura 2.2 – Inversão do fluxo de blocos em sistemas de escrita verticais.	14
Figura 2.3 – Representação dos eixos principal (<i>main axis</i>) e transversal (<i>cross axis</i>) no modelo <i>Flexbox</i>	16
Figura 2.4 – Representação visual dos principais conceitos e terminologias.	18
Figura 2.5 – Representação visual do comportamento em bloco com <i>display: grid</i>	21
Figura 2.6 – Representação visual do comportamento em linha com <i>display: inline-grid</i>	22
Figura 2.7 – Configuração com colunas fixas.	22
Figura 2.8 – Configuração com linhas fixas.	22
Figura 2.9 – Distribuição proporcional utilizando unidades percentuais.	23
Figura 2.10–Distribuição vertical proporcional via porcentagem.	23
Figura 2.11–Divisão dinâmica de espaço via unidades fracionárias.	23
Figura 2.12–Escalonamento vertical dinâmico com unidades fracionárias.	23
Figura 2.13–Simplificação estrutural com o uso da função de repetição.	24
Figura 2.14–Padronização de alturas via função de repetição.	24
Figura 2.15–Controle de limites dimensionais com <i>minmax()</i>	24
Figura 2.16–Gestão de limites verticais com a função <i>minmax()</i>	24
Figura 2.17–Mapeamento visual e organização de áreas nomeadas.	25
Figura 2.18–Controle de espaçamento estrutural via <i>column-gap</i> e <i>row-gap</i>	26
Figura 2.19–Comportamento padrão dos valores da propriedade <i>justify-content</i>	28
Figura 2.20–Comportamento padrão dos valores da propriedade <i>align-content</i>	30
Figura 2.21–Comportamento padrão das propriedades <i>justify-items</i>	31
Figura 2.22–Comportamento padrão dos valores da propriedade <i>align-items</i>	32
Figura 2.23–Posicionamento horizontal baseado em índices de <i>grid lines</i>	33
Figura 2.24–Uso da cláusula <i>span</i> para expansão de colunas.	34
Figura 2.25–Sobreposição de itens e controle de hierarquia visual.	34
Figura 2.26–Mapeamento bidimensional: definição de região espacial via <i>grid-area</i>	35
Figura 2.27–Posicionamento vertical baseado em índices de <i>grid lines</i> (<i>row: 2/4</i>).	35
Figura 2.28–Uso da cláusula <i>span</i> para expansão vertical de linhas (<i>1/span 2</i>).	36
Figura 2.29–Sobreposição de itens em eixos cruzados e controle de hierarquia com <i>z-index</i>	36
Figura 2.30–Impacto visual das opções de alinhamento horizontal individual no item.	37
Figura 2.31–Diferentes modos de distribuição vertical individual de um item na grade.	38
Figura 3.1 – Interface do jogo <i>Flexbox Froggy</i> , destacando o painel de instruções e o editor de código CSS.	45
Figura 3.2 – Área visual do jogo <i>Flexbox Froggy</i> , na qual os sapos devem ser alinhados às vitórias-régias por meio da aplicação de propriedades do <i>Flexbox</i>	46

Figura 3.3 – Interface do jogo CSS <i>Grid Garden</i> , destacando o painel de instruções e o editor de código CSS para configuração do <i>grid</i>	47
Figura 3.4 – Área visual do jogo CSS <i>Grid Garden</i> , evidenciando a organização bidimensional do tabuleiro em linhas e colunas.	47
Figura 4.1 – Interface geral do sistema <i>Tezza Space</i>	50
Figura 4.2 – Painel de instruções da missão.	50
Figura 4.3 – Comportamento do painel de regras	51
Figura 4.4 – Comportamento do painel de regras	51
Figura 4.5 – Console de comandos após o acionamento do botão “Solução”.	52
Figura 4.6 – Comportamento do tabuleiro orbital	52
Figura 4.7 – Tabuleiro com a sobreposição visual ativada.	53
Figura 4.8 – Plano mental utilizado como referência durante o desenvolvimento da sequência de missões do <i>Tezza Space</i>	54
Figura 4.9 – Linha do tempo do processo de refinamento visual dos <i>assets</i> : alienígena, painel solar e gosma.	57
Figura 4.10–Evolução visual do astronauta <i>asset</i> astronauta.	57
Figura 4.11–Diagrama de entidades da modelagem conceitual.	58
Figura 4.12–Visão geral do fluxo de edição e processamento de regras.	63
Figura 4.13–Relatório de cobertura de testes gerado pelo <i>Jest</i>	69
Figura 5.1 – O Astronauta Perdido.	71
Figura 5.2 – Vazamento no Reator.	72
Figura 5.3 – A Poça de Gosma Alienígena.	72
Figura 5.4 – Expansão do Hangar.	73
Figura 5.5 – Oficina do BOT-Z.	73
Figura 5.6 – Blindagem de Emergência.	74
Figura 5.7 – O Buraco Negro Portátil.	74
Figura 5.8 – Reparando Circuitos Sensíveis.	75
Figura 5.9 – Teletransportador Orbital.	75
Figura 5.10–A Última Defesa da Tezza-1.	76
Figura 5.11–Primeiro Contato com a <i>Grid</i>	77
Figura 5.12–Corredor de Suprimentos.	78
Figura 5.13–Setor de Painéis Solares.	78
Figura 5.14–Tubos de Ventilação.	79
Figura 5.15–Vazamento Criogênico.	79
Figura 5.16–Doca de Acoplamento Espacial.	80
Figura 5.17–Alinhamento de Drones.	80
Figura 5.18–Câmara de Gravidade Zero.	81
Figura 5.19–Redistribuição de Energia	81
Figura 5.20–O Mapa da Estação.	82

Lista de Tabelas

Tabela 4.1 – Estrutura em blocos do prompt-base utilizado na geração de assets.	56
Tabela 4.2 – Suítes de teste automatizado implementadas no sistema <i>Tezza Space</i>	68

Lista de Abreviaturas e Siglas

ChatGPT Chat Generative Pre-trained Transformer. [53](#)

CSS *Cascading Style Sheets*. [vi](#), [vii](#), [ix](#), [xii](#), [xiii](#), [1–7](#), [9–20](#), [25](#), [43–49](#), [51–53](#), [55](#), [58](#), [61](#), [63–65](#), [70](#), [71](#), [76](#), [83](#), [84](#), [89](#)

DOM *Document Object Model*. [5–7](#), [25](#), [39](#)

HTML *HyperText Markup Language*. [xii](#), [4–7](#), [9–11](#), [17](#), [25](#), [44](#)

JIT *Just-In-Time*. [7](#)

MDN *Mozilla Developer Network*. [5](#), [6](#)

MVC *Model–View–Controller*. [43](#)

REST *Representational State Transfer*. [44](#)

UML *Unified Modeling Language*. [43](#)

W3C *World Wide Web Consortium*. [5](#), [6](#), [16](#)

WHATWG *Web Hypertext Application Technology Working Group*. [5](#), [6](#)

Sumário

1	Introdução	1
1.1	Justificativa	2
1.2	Objetivos	3
1.3	Organização do Trabalho	3
2	Referencial Teórico	4
2.1	Fundamentos Tecnológicos do Ambiente Web	4
2.2	<i>HyperText Markup Language</i>	5
2.3	<i>JavaScript</i>	6
2.3.1	Arquitetura e Execução	7
2.3.2	Peculiaridades e Desafios Técnicos	7
2.4	<i>TypeScript</i>	8
2.4.1	O Sistema de Tipos do <i>TypeScript</i>	9
2.5	<i>Cascading Style Sheets</i>	9
2.5.1	A evolução das técnicas de <i>layout</i> em CSS	10
2.5.2	<i>Layouts</i> Baseados em Tabelas	11
2.5.3	<i>Layouts</i> Baseados em <i>Float</i>	12
2.5.4	<i>Layouts</i> Baseados em <i>Inline-block</i>	12
2.5.4.1	Elementos de Bloco e de Linha	13
2.5.5	Posicionamento Absoluto e Relativo	14
2.5.6	<i>Flexbox</i> : Modelo Unidimensional de <i>Layout</i>	15
2.6	<i>CSS Grid</i>	16
2.6.1	Componentes de um <i>CSS Grid Layout</i>	17
2.6.1.1	<i>Grid Container</i>	18
2.6.1.2	<i>Grid Items</i>	18
2.6.1.3	<i>Grid Line</i>	19
2.6.1.4	<i>Grid Track</i>	19
2.6.1.5	<i>Grid Cell</i>	19
2.6.1.6	<i>Grid Area</i>	20
2.6.2	Propriedades do Contêiner e dos Itens no <i>CSS Grid</i>	20
2.6.3	Propriedades do Contêiner Pai	20
2.6.3.1	Propriedades <i>display: grid</i> e <i>display:inline-grid</i>	21
2.6.3.2	Propriedades <i>grid-template-columns</i> e <i>grid-template-rows</i>	21
2.6.3.3	Propriedade <i>grid-template-areas</i>	25
2.6.3.4	Propriedade <i>gap</i> (<i>row-gap</i> e <i>column-gap</i>)	26
2.6.3.5	Propriedade <i>justify-content</i>	27
2.6.3.6	Propriedade <i>align-content</i>	28

2.6.3.7	Propriedade <i>justify-items</i>	31
2.6.3.8	Propriedade <i>align-items</i>	32
2.6.4	Propriedades dos Itens Filhos	33
2.6.4.1	Propriedade <i>grid-column</i>	33
2.6.4.2	Propriedade <i>grid-area</i>	34
2.6.4.3	Propriedade <i>grid-row</i>	35
2.6.4.4	Propriedade <i>justify-self</i>	36
2.6.4.5	Propriedade <i>align-self</i>	37
2.7	<i>React</i> : biblioteca da camada de visualização	38
2.7.1	Arquitetura baseada em componentes	39
2.7.2	Gerenciamento de estado em aplicações <i>React</i> : <i>Context API</i> e <i>useReducer</i>	41
3	Trabalhos Relacionados	43
3.1	Plataforma Interativa <i>Flexbox Froggy</i> como Abordagem Didática para o Ensino de <i>Flexbox</i>	45
3.2	Plataforma Interativa CSS <i>Grid Garden</i> como Referência Didática	46
4	CSS Tezza Space	49
4.1	Visão geral do sistema	49
4.2	Estrutura de fases e progressão didática	53
4.3	Produção dos <i>assets</i> visuais com Inteligência Artificial Generativa	55
4.4	Modelagem conceitual do sistema	57
4.5	Arquitetura da aplicação	60
4.5.1	Gerenciamento de estado	61
4.5.2	Camada de apresentação	61
4.6	Fluxo de edição e processamento de regras	62
4.7	Funções auxiliares de suporte à <i>Grid</i> unificada	64
4.8	Renderização no tabuleiro orbital	66
4.9	Testes automatizados do Tezza Space	67
5	As fases da missão <i>Tezza Space</i>	70
5.1	Visualização do posicionamento dos <i>Grid Items</i>	70
5.1.1	Fase 01 — O Astronauta Perdido	71
5.1.2	Fase 02 — Vazamento no Reator	71
5.1.3	Fase 03 — A Poça de Gosma Alienígena	72
5.1.4	Fase 04 — Expansão do Hangar	72
5.1.5	Fase 05 — Oficina do BOT-Z	73
5.1.6	Fase 06 — Blindagem de Emergência	73
5.1.7	Fase 07 — O Buraco Negro Portátil	74
5.1.8	Fase 08 — Reparando Circuitos Sensíveis	74
5.1.9	Fase 09 — Teletransportador Orbital	75
5.1.10	Fase 10 — A Última Defesa da Tezza-1	75

5.2	Visualização da estrutura e organização do <i>Grid Container</i>	76
5.2.1	Fase 11 — Primeiro Contato com a <i>Grid</i>	77
5.2.2	Fase 12 — Corredor de Suprimentos	77
5.2.3	Fase 13 — Setor de Painéis Solares	78
5.2.4	Fase 14 — Tubos de Ventilação	78
5.2.5	Fase 15 — Vazamento Criogênico	79
5.2.6	Fase 16 — Doca de Acoplamento Espacial	79
5.2.7	Fase 17 — Alinhamento de Drones	80
5.2.8	Fase 18 — Câmara de Gravidade Zero	80
5.2.9	Fase 19 — Redistribuição de Energia	81
5.2.10	Fase 20 — O Mapa da Estação	81
6	Considerações Finais	83
6.1	Trabalhos Futuros	84
	Referências	85

1 Introdução

O desenvolvimento de interfaces Web no contexto do *front-end* tem passado por transformações significativas nas últimas décadas, impulsionado pela crescente diversidade de dispositivos, resoluções e contextos de acesso às aplicações digitais. Esse cenário amplia a complexidade da construção de *layouts*, exigindo dos desenvolvedores soluções capazes de se adaptar a diferentes tamanhos de tela sem comprometer a usabilidade, a coerência estrutural e a consistência visual. Nesse contexto, as *Cascading Style Sheets (CSS)* assumem papel central na organização da apresentação, ao possibilitarem a separação entre estrutura e estilo e ao favorecerem a adoção de técnicas responsivas e modelos de organização espacial mais sofisticados (Doan, 2017).

Ao longo da evolução das *CSS*, a organização de *layouts* em páginas Web percorreu diferentes estágios, que incluíram o uso de tabelas com finalidade estrutural, técnicas baseadas na propriedade *float*, soluções com *inline-block* e, posteriormente, abordagens unidimensionais como o *Flexbox*. Embora tais estratégias tenham representado avanços importantes no contexto do desenvolvimento *front-end*, revelaram limitações relacionadas à manutenção, à escalabilidade e, sobretudo, ao controle simultâneo dos eixos horizontal e vertical em interfaces mais complexas (Pham, 2025). Nesse processo de amadurecimento técnico, o *CSS Grid Layout* consolidou-se como um modelo bidimensional nativo, permitindo a definição declarativa de linhas e colunas e oferecendo maior clareza estrutural na organização espacial dos elementos, ao superar restrições impostas por técnicas anteriores (Pham, 2025).

Com a consolidação do *CSS Grid Layout*, tornou-se possível estruturar *layouts* a partir de um modelo bidimensional nativo, no qual linhas e colunas podem ser definidas de forma declarativa (Pham, 2025). Essa abordagem amplia o controle sobre a organização espacial dos elementos e representa um avanço técnico relevante no ecossistema de estilização Web.

Entretanto, a presença de uma solução tecnicamente robusta não garante sua incorporação ampla na prática profissional. Levantamento conduzido pelo *State of CSS* junto a desenvolvedores de todo o mundo indica que o *CSS Grid* é reconhecido como o recurso que mais influenciou a maneira de escrever *CSS* nos últimos cinco anos, superando inclusive o *Flexbox* nesse quesito (Devographics, 2025). Esse reconhecimento, no entanto, não se traduz automaticamente em domínio prático da tecnologia por parte dos profissionais da área, o que aponta para a necessidade de refletir não apenas sobre aspectos técnicos, mas também sobre fatores relacionados à formação e à aprendizagem da tecnologia.

A compreensão do *CSS Grid* exige do desenvolvedor a internalização de uma lógica espacial bidimensional, distinta do fluxo tradicional de organização de elementos. Essa mudança demanda a construção de um modelo mental capaz de relacionar declarações em código com seus efeitos visuais na estrutura do *layout*. Conforme discutido por Kolb (1984), a aprendizagem

tende a ser mais significativa quando envolve participação ativa e experimentação concreta. De forma complementar, Mayer (2009) destaca que conteúdos de natureza estrutural e espacial são mais bem assimilados quando apresentados por meio da integração entre representação visual e interação.

Nesse contexto, propõe-se o desenvolvimento de uma plataforma interativa Tezza Space em formato de jogo como estratégia didática para o ensino do *CSS Grid*. A incorporação de elementos de gamificação busca estimular o engajamento do usuário e favorecer a construção progressiva do conhecimento por meio da experimentação orientada. Ao possibilitar que o estudante edite propriedades, teste configurações e observe, em tempo real, os efeitos produzidos na organização do *layout*, a proposta aproxima teoria e prática e dialoga com abordagens pedagógicas que valorizam a participação ativa no processo de aprendizagem.

1.1 Justificativa

Embora o *CSS Grid Layout* represente uma solução tecnicamente consolidada para a construção de *layouts* bidimensionais, sua adoção na prática profissional ainda não ocorre de forma proporcional ao seu potencial. Segundo o levantamento *State of CSS 2025*, o *Grid* foi apontado por 272 dos 1.259 respondentes, cerca de 21,6% do total, como o principal ponto de dificuldade relacionado a *layout* em *CSS*, à frente de problemas como os associados ao próprio *Flexbox* e ao suporte entre navegadores (Devographics, 2025). Embora o *Grid* seja inegavelmente poderoso, esse poder também vem acompanhado de uma curva de aprendizado acentuada, o que pode transformar o uso desse recurso bastante complexo em um verdadeiro problema. Esse cenário reforça que a limitação não está associada à capacidade técnica da ferramenta, mas sim às dificuldades relacionadas ao seu processo de aprendizagem.

Além disso, grande parte dos materiais disponíveis para o ensino de *CSS Grid* baseia-se em documentação técnica extensa e exemplos estáticos, o que pode dificultar a compreensão de seus conceitos fundamentais, especialmente por estudantes iniciantes. Essa limitação aproxima-se das observações de Henklain, Carvalho e Feitosa (2025), cujo mapeamento sistemático da literatura sobre dificuldades no aprendizado de programação identifica que grande parte dos obstáculos enfrentados por estudantes está relacionada à compreensão de características da linguagem e à leitura e interpretação de código, processos que exigem a construção de modelos mentais sobre comportamentos dinâmicos a partir de conteúdos frequentemente apresentados de forma estática, como textos, apresentações e explicações verbais. No contexto do *CSS Grid*, essa dificuldade torna-se ainda mais evidente, uma vez que a compreensão de propriedades relacionadas ao posicionamento, dimensionamento e alinhamento depende diretamente da visualização dos efeitos produzidos na interface.

Nesse contexto, justifica-se a proposição de uma plataforma interativa em formato de jogo como abordagem didática para o ensino do *CSS Grid*. A utilização de elementos de gamificação

busca promover maior engajamento, reduzir a curva de aprendizado e favorecer a construção progressiva do conhecimento. Ao permitir que o usuário experimente, teste e observe, em tempo real, o comportamento dos elementos de *layout*, a proposta alinha-se a metodologias pedagógicas ativas e responde às limitações identificadas nas abordagens tradicionais de ensino.

1.2 Objetivos

O objetivo principal deste trabalho é propor e desenvolver uma abordagem didática para o ensino do *CSS Grid Layout*, por meio de uma plataforma interativa em formato de jogo, que possibilite a compreensão e a aplicação prática de seus conceitos fundamentais no contexto do desenvolvimento *front-end*.

Para que o objetivo principal seja cumprido, têm-se os seguintes objetivos específicos:

- Sistematizar os conceitos fundamentais do *CSS Grid*, organizando-os de forma progressiva e didática, contemplando tanto as propriedades do *Grid Container* quanto as propriedades dos *Grid Items*;
- Promover a visualização direta e o *feedback* imediato como estratégias de apoio à compreensão do impacto das propriedades do *CSS Grid* na organização do *layout*;
- Desenvolver uma plataforma interativa inspirada em abordagens gamificadas, que possibilite a experimentação prática e o aprendizado incremental das propriedades do *CSS Grid*.

1.3 Organização do Trabalho

Este trabalho está organizado em seis capítulos, os quais abordam, de forma progressiva, o contexto, a fundamentação teórica, os trabalhos relacionados, o desenvolvimento da proposta, a apresentação detalhada das fases criadas e as considerações finais. No *Capítulo 2* é apresentada a fundamentação teórica necessária para a compreensão do trabalho. No *Capítulo 3* há uma análise de trabalhos relacionados, possibilitando uma visão crítica sobre abordagens já existentes para o ensino dessa tecnologia. O detalhamento da estrutura, do funcionamento do sistema proposto (*Tezza Space*) e dos testes realizados é apresentado no *Capítulo 4*, enquanto as fases criadas são detalhadas individualmente no *Capítulo 5*. Por fim, os desdobramentos desta proposta e os possíveis trabalhos futuros são descritos no *Capítulo 6*.

2 Referencial Teórico

Por meio deste capítulo, busca-se oferecer uma visão abrangente dos fundamentos conceituais que sustentam o presente trabalho. Inicialmente, na Seção 2.1, são discutidos os princípios que estruturam o funcionamento da Web, contemplando a arquitetura cliente-servidor, o papel do navegador e a organização em camadas estruturais, comportamentais e visuais, representadas por *HyperText Markup Language (HTML)*, *JavaScript* e *Cascading Style Sheets (CSS)*, temas retomados nas Seções 2.2, 2.3, 2.4 e 2.5. Em seguida, a Seção 2.5.1 apresenta a evolução histórica das técnicas de *layout* em CSS, preparando o terreno conceitual para a Seção 2.6, na qual são apresentados os conceitos específicos relacionados ao *CSS Grid Layout*, com ênfase em sua lógica bidimensional, em seus componentes estruturais e nas propriedades aplicadas tanto aos contêineres quanto aos elementos filhos.

Posteriormente, a Seção 2.7 aborda os fundamentos do desenvolvimento de interfaces com *React*, explorando o modelo declarativo, a arquitetura baseada em componentes e os mecanismos de gerenciamento de estado, como a *Context API* e o *useReducer*.

Essa organização busca articular dimensões tecnológicas e educacionais de forma integrada, estruturando o referencial teórico que fundamenta as decisões metodológicas e arquiteturais adotadas no desenvolvimento do *Tezza Space*.

2.1 Fundamentos Tecnológicos do Ambiente Web

O funcionamento da Web está fundamentado na arquitetura cliente-servidor, um modelo amplamente descrito na literatura de sistemas distribuídos (Coulouris *et al.*, 2012). Nesse paradigma, as responsabilidades são divididas entre duas entidades principais: o cliente, que inicia a comunicação por meio do envio de requisições, e o servidor, que processa essas solicitações e retorna as respostas correspondentes. Conforme discutido por Coulouris *et al.* (2012), essa organização favorece a distribuição de recursos e a coordenação entre componentes que operam em ambientes interconectados, estabelecendo um padrão estruturado de interação baseado em requisição e resposta.

No contexto da Web, o navegador assume o papel de cliente ao solicitar documentos, imagens e outros recursos hospedados em servidores conectados à internet. Esses servidores gerenciam os conteúdos disponibilizados e respondem às requisições recebidas, podendo inclusive realizar processamento prévio antes de enviar a resposta ao usuário (Mozilla Developer Network, 2025a). Essa dinâmica evidencia que a Web não se limita à entrega de arquivos estáticos, mas envolve também o processamento do lado do servidor, reforçando a importância da separação entre cliente e servidor na organização das aplicações Web.

A concretização dessa comunicação ocorre por meio de tecnologias responsáveis por estruturar e apresentar o conteúdo ao usuário final. Entre elas, destacam-se o [HTML](#), o [Cascading Style Sheets \(CSS\)](#) e o [JavaScript](#), que constituem a base do desenvolvimento *front-end*. Segundo as especificações do [Web Hypertext Application Technology Working Group \(WHATWG\)](#) e pelo [World Wide Web Consortium \(W3C\)](#), o [HTML](#) organiza semanticamente o documento ([WHATWG, 2026b](#)), o [CSS](#) define sua apresentação visual ([World Wide Web Consortium, 2018](#)) e o [JavaScript](#) acrescenta comportamento e interatividade ([ECMA International, 2023](#)). A integração dessas três tecnologias permite que os recursos recebidos sejam convertidos em interfaces navegáveis e dinâmicas.

Ao receber um documento [HTML](#), o navegador o interpreta e constrói o [Document Object Model \(DOM\)](#), uma estrutura hierárquica em forma de árvore que representa os elementos da página ([WHATWG, 2026a](#)). Essa estrutura viabiliza a aplicação de estilos por meio do [CSS](#) e a manipulação dinâmica por scripts [JavaScript](#), possibilitando a criação de interfaces interativas.

O processo de renderização também envolve o conceito de *viewport*, entendido como a área visível da página dentro da janela do navegador. Conforme descrito na documentação técnica da [Mozilla Developer Network \(2025b\)](#), suas dimensões variam de acordo com o dispositivo utilizado, influenciando diretamente o comportamento responsivo da interface. A compreensão desse conceito é fundamental para o estudo das técnicas de *layout* e para a adaptação do conteúdo a diferentes contextos de uso.

2.2 [HyperText Markup Language](#)

O [HyperText Markup Language \(HTML\)](#) corresponde à linguagem de marcação que fundamenta a organização dos documentos na Web. Conforme definido na especificação mantida pelo [W3C](#), sua função principal é estabelecer a estrutura lógica do conteúdo, permitindo que navegadores distintos interpretem as páginas de forma consistente e compatível entre diferentes ambientes ([WHATWG, 2026b](#)).

Segundo a [Mozilla Developer Network \(MDN\)](#), o [HTML](#) não executa instruções como uma linguagem de programação, mas utiliza elementos estruturais para delimitar e identificar as diferentes partes de um documento ([Mozilla Developer Network, 2026](#)). A composição básica de uma página inicia-se com a declaração `<!DOCTYPE html>`, seguida pelo elemento `<html>`, que atua como raiz da estrutura e abriga duas seções fundamentais: `<head>` e `<body>`. Essa organização hierárquica sustenta a construção de toda a interface Web.

A seção `<head>` concentra informações descritivas sobre o documento, como o título, a definição de codificação e as ligações com arquivos externos, enquanto a seção `<body>` contém o conteúdo que será apresentado ao usuário na interface ([Mozilla Developer Network, 2026](#)). Essa distinção estrutural está prevista na própria especificação da linguagem, reforçando a separação entre os metadados e o conteúdo visível ([WHATWG, 2026b](#)).

Conforme destacado pela [MDN](#), é frequente a menção aos chamados “novos elementos do HTML5” ou à ideia de que o [HTML5](#) representa uma nova versão da linguagem ([Mozilla Developer Network, 2026](#)). Historicamente, o [HTML5](#) sucedeu versões anteriores, como o [HTML 4.01](#), incorporando novos elementos e funcionalidades, além de revisar ou descontinuar recursos considerados obsoletos. Entretanto, o [HTML](#) passou a ser mantido sob um modelo de atualização contínua, o que implica que não existe mais uma versão numerada fixa. Diferentemente do modelo tradicional de versionamento numerado, a especificação do [HTML](#) é hoje mantida como um *Living Standard*, revisado e aprimorado de forma contínua pelo [WHATWG](#), em colaboração formal com o [W3C](#) desde 2019 ([WHATWG, 2026b](#)).

Entre os avanços associados ao [HTML5](#) destacam-se a ampliação do conjunto de elementos semânticos, como `<header>`, `<nav>`, `<main>`, `<section>` e `<article>`, bem como a incorporação de recursos nativos voltados à multimídia e à renderização gráfica, como `<audio>`, `<video>` e `<canvas>` ([Mozilla Developer Network, 2026](#)). Tais recursos contribuíram para fortalecer a expressividade semântica dos documentos e reduzir a dependência de soluções externas.

No processo de carregamento de uma página, o navegador interpreta o código [HTML](#) e o transforma em uma estrutura em árvore denominada *Document Object Model (DOM)*, conforme descrito na especificação oficial ([WHATWG, 2026a](#)). Essa representação permite que *scripts* modifiquem dinamicamente elementos, atributos e conteúdos, ampliando as possibilidades de interação e dinamismo nas aplicações Web ([Mozilla Developer Network, 2026](#)).

Assim, o [HTML](#) configura-se como a base estrutural das interfaces Web atuais, evoluindo continuamente por meio de um processo colaborativo de atualização mantido conjuntamente pelo [WHATWG](#) e pelo [W3C](#) ([WHATWG, 2026b](#)). Em conjunto com o [CSS](#), responsável pela apresentação visual, e com o *JavaScript*, encarregado da camada comportamental, o [HTML](#) compõe o núcleo tecnológico do desenvolvimento Web.

2.3 *JavaScript*

O *JavaScript* consolidou-se como um componente indispensável para o ecossistema de aplicações Web contemporâneas. De acordo com a [MDN de JavaScript \(Network, 2026\)](#), a linguagem transcendeu sua função original de *scripting* para navegadores, assumindo um papel central no desenvolvimento de sistemas complexos. Introduzida pela Netscape em 1995, a linguagem foi inicialmente proposta como uma ferramenta orientada a objetos de fácil utilização, visando a integração dinâmica de recursos entre cliente e servidor ([Network, 2026](#)).

Embora tenha se estabelecido como o padrão para execução no lado do cliente, a onipresença da linguagem em ambientes que exigem escalabilidade revelou lacunas estruturais. Conforme observado por [Laursen, Xu e Møller \(2024\)](#), a análise estática de programas JavaScript permanece, ainda hoje, mais desafiadora do que a de outras linguagens de programação, principalmente em razão de operações dinâmicas, como acessos a propriedades por meio de nomes

computados em tempo de execução, que seguem escapando das análises de tipo mais avançadas disponíveis.

2.3.1 Arquitetura e Execução

No modelo de camadas das tecnologias Web, o *JavaScript* atua como a camada de comportamento, complementando o [HTML](#) (estrutura) e o [CSS](#) (apresentação) para viabilizar a interatividade e a manipulação dinâmica do [DOM](#) ([Network, 2026](#)). A execução ocorre em ambientes processados por motores de alto desempenho, como o V8 ou SpiderMonkey, que empregam compilação *Just-In-Time* (JIT) para a otimização de recursos.

Contudo, por possuir tipagem dinâmica e fraca, a linguagem apresenta fragilidades no processo de desenvolvimento. Isso está diretamente relacionado à forma ambígua como tipos primitivos e objetos são manipulados, o que exige atenção redobrada na conversão implícita de valores. Nesse contexto, a análise de tipos torna-se fundamental para reduzir conversões implícitas indesejadas e preservar a consistência dos dados ao longo da aplicação.

2.3.2 Peculiaridades e Desafios Técnicos

Segundo [Jensen, Møller e Thiemann \(2009\)](#), consolidado por pesquisas mais recentes do mesmo grupo, [Laursen, Xu e Møller \(2024\)](#), as principais peculiaridades e desafios técnicos do *JavaScript* são:

- **Modelo de Herança:** Diferente de linguagens baseadas em classes, utiliza protótipos para a modelagem de objetos.
- **Dinamicidade de Objetos:** Propriedades podem ser adicionadas ou modificadas em tempo de execução, funcionando como mapeamentos de *strings* para valores.
- **Ambiguidade de Valores:** A distinção sutil entre a ausência de uma propriedade e o valor *undefined*, além da dicotomia entre valores primitivos e seus respectivos objetos *wrapper*.
- **Escopo e Execução:** A persistência de instruções como o *with*, que comprometem o escopo léxico, e o uso de *eval*, que permite a execução de código arbitrário, dificultando a análise estática.
- **Flexibilidade Funcional:** Funções operam como objetos de primeira classe, podendo atuar simultaneamente como métodos, construtores e procedimentos com número variável de parâmetros.

Para mitigar os riscos inerentes à flexibilidade da linguagem, a especificação do ECMAScript estabelece o uso de módulos como mecanismo formal de organização e encapsulamento de funcionalidades ([ECMA International, 2023](#)). Esse modelo permite a definição explícita de

exportações e importações entre arquivos, reduzindo o acoplamento e favorecendo a manutenção do sistema. Em suma, embora o *JavaScript* ofereça um ambiente versátil para o desenvolvimento rápido, as limitações de sua tipagem dinâmica permanecem, quase duas décadas depois dos primeiros esforços de análise de tipos para a linguagem, um desafio de pesquisa ativo, conforme evidenciam os trabalhos mais recentes do mesmo grupo de pesquisa (Laursen; Xu; Møller, 2024).

Nesse contexto, as deficiências estruturais na verificação de tipos em larga escala impulsionaram o surgimento de superconjuntos sintáticos (*supersets*). Entre as soluções propostas, o *TypeScript* destaca-se por introduzir uma camada de tipagem estática opcional ao *JavaScript*, permitindo a detecção de erros em tempo de compilação e oferecendo o suporte necessário para o desenvolvimento de software robusto e passível de manutenção a longo prazo.

2.4 *TypeScript*

O *TypeScript*, desenvolvido pela *Microsoft* e disponibilizado ao público em 2012, configura-se como uma resposta às limitações de escalabilidade historicamente associadas ao *JavaScript* (Microsoft, 2026). Definido formalmente como um superconjunto sintático estrito (*strict syntactic superset*) da especificação *ECMAScript*, o *TypeScript* adiciona uma camada de tipagem estática opcional ao *JavaScript* sem alterar sua semântica de execução (Microsoft, 2026).

De acordo com a documentação oficial (Microsoft, 2026), o *TypeScript* é concebido como um verificador de tipos estático, capaz de identificar erros antes da execução com base nos tipos de valores. Essa característica confere maior previsibilidade ao processo de desenvolvimento, permitindo que inconsistências sejam detectadas em tempo de checagem estática (*compile-time*) e mitigadas antes da efetiva execução do código em ambiente de *runtime*.

Do ponto de vista sintático, o *TypeScript* corresponde a um superconjunto estrito da especificação *ECMAScript*, o que implica que programas *JavaScript* padronizados são, por definição, códigos *TypeScript* válidos (Microsoft, 2026). Essa relação de superconjunto no desenvolvimento independe do alvo de compilação (*target*) selecionado no arquivo de configuração do projeto; este último determina estritamente para qual versão da especificação *ECMAScript* (como ES5 ou ESNext) o código final será transpilado, alinhando-se aos navegadores ou ambientes de execução (*runtimes*) alvo. Essa compatibilidade sintática assegura a adoção gradual da linguagem em projetos já consolidados, sem comprometer a interoperabilidade com o ecossistema existente (Microsoft, 2026).

Além disso, o *TypeScript* expande as capacidades do *JavaScript* ao integrar um sistema de módulos, classes, interfaces e um mecanismo de tipagem estática. Tais recursos não apenas enriquecem o processo de desenvolvimento, mas também promovem maior robustez e previsibilidade na construção de aplicações de grande porte, estabelecendo um equilíbrio entre a flexibilidade dinâmica do *JavaScript* e os princípios de segurança e organização característicos de linguagens fortemente tipadas.

2.4.1 O Sistema de Tipos do *TypeScript*

A proposta do *TypeScript* não é se estabelecer como uma linguagem independente, mas atuar como uma extensão que amplia as capacidades do *JavaScript*, mantendo a compatibilidade com seus programas existentes. Conforme descreve a documentação oficial ([Microsoft, 2026](#)), o sistema de tipos foi projetado para acomodar os padrões já consolidados da linguagem, permitindo que desenvolvedores adotem tipagem estática sem romper com o ecossistema previamente estabelecido. Essa orientação arquitetural resulta em um conjunto de características específicas que definem o funcionamento do *TypeScript*:

- **Apagamento de tipos:** No *TypeScript*, as anotações de tipo são utilizadas apenas durante a verificação estática e não permanecem no código *JavaScript* gerado. Não há representação de tipos em tempo de execução nem verificação dinâmica automática ([Microsoft, 2026](#)). Assim, o código resultante continua sendo essencialmente *JavaScript*, mantendo sua natureza dinâmica.
- **Tipagem estrutural:** A verificação de tipos é baseada na estrutura dos objetos, ou seja, nas propriedades e métodos que eles possuem, e não no nome do tipo declarado. Diferente de linguagens como Java e C#, que exigem correspondência nominal entre classes e interfaces, o *TypeScript* considera dois tipos compatíveis quando apresentam a mesma forma. Essa característica se adequa ao *JavaScript*, em que os objetos são frequentemente criados de maneira flexível e utilizados conforme a estrutura que oferecem.
- **Unificação de tipos de objetos:** Objetos em *JavaScript* podem assumir múltiplos papéis (funções, construtores, vetores). O *TypeScript* reflete essa característica ao permitir que tipos de objetos descrevam diferentes assinaturas de uso.
- **Inferência de tipos:** O compilador é capaz de deduzir automaticamente tipos em diversos contextos, reduzindo a necessidade de anotações explícitas e preservando a concisão sintática da linguagem.
- **Tipagem gradual:** O *TypeScript* possibilita a combinação de trechos fortemente tipados com outros dinamicamente tipados por meio do tipo especial *any*. Essa flexibilidade permite que os projetos evoluam de forma incremental em direção a uma maior segurança de tipos.

2.5 *Cascading Style Sheets*

O *Cascading Style Sheets* (CSS) consolidou-se como a linguagem fundamental para a definição da apresentação visual em documentos digitais. Historicamente, a expansão da rede mundial de computadores impôs o desenvolvimento de interfaces mais complexas, evidenciando a insuficiência dos recursos nativos do [HTML](#) em atender às demandas de design e usabilidade. Conforme apontam [Ruvalcaba e Boehm \(2021\)](#), o [HTML](#) foi concebido estritamente para a

estruturação de conteúdos, carecendo de mecanismos eficientes para o controle de atributos como cores, tipografia, espaçamentos e posicionamento. A lacuna técnica tornava a criação de interfaces esteticamente funcionais um desafio, posicionando o **CSS** como um complemento indispensável ao permitir a separação entre a estrutura lógica e a camada de apresentação (Ruvalcaba; Boehm, 2021).

Essa distinção funcional é fundamentada na natureza intrínseca do **HTML**. De acordo com Blanc (2020), o **HTML** caracteriza-se como uma linguagem de marcação destinada a representar a estrutura de um documento por meio de etiquetas (*tags*) que indicam a função semântica do texto. Essa definição reforça que o propósito primordial do **HTML** é a organização do conteúdo e não a atribuição de propriedades visuais, o que justifica a necessidade de uma linguagem auxiliar dedicada à estilização.

Em resposta a essas limitações, Håkon Wium Lie e Bert Bos propuseram, em 1994, o modelo de Folhas de Estilo em Cascata. Segundo Bos e Lie (1997), a proposta visava dissociar a semântica estrutural da renderização visual, introduzindo seletores que permitiam a aplicação sistemática de estilos. O conceito de “cascata” foi um diferencial determinante, pois estabeleceu critérios de precedência para integrar múltiplas folhas de estilo oriundas de autores, navegadores e usuários, oferecendo uma solução técnica para o conflito de preferências de formatação (Bos; Lie, 1997).

Embora concebido na década de 1990, o **CSS** alcançou maturidade e implementação prática apenas no início dos anos 2000. O lançamento do Internet Explorer 5 em 1999 é frequentemente citado como um marco histórico por ter sido o primeiro navegador a oferecer suporte técnico robusto às especificações da linguagem. Consequentemente, o **CSS** foi consolidado como o padrão global para o desenvolvimento Web, viabilizando o avanço técnico de designers e programadores na criação de *layouts* sofisticados e consistentes (World Wide Web Consortium (W3C), 1996).

2.5.1 A evolução das técnicas de *layout* em **CSS**

A evolução cronológica dos métodos de layout em **CSS** reflete uma busca contínua pela superação de limitações técnicas, especialmente no que tange ao alinhamento de componentes e à adaptabilidade das interfaces em múltiplos dispositivos. Inicialmente, a prática comum consistia no uso de tabelas **HTML** para a organização de conteúdos em matrizes de linhas e colunas. Conforme observa Pham (2025), embora essa abordagem oferecesse um controle pragmático sobre a disposição visual, ela gerava um prejuízo significativo à semântica documental e à acessibilidade digital, evidenciando a urgência de métodos que dissociassem a estrutura lógica da formatação espacial.

Com o amadurecimento das especificações do **CSS**, foram introduzidas técnicas baseadas em propriedades como *float*, *inline-block* e posicionamento absoluto. Tais recursos ampliaram

a flexibilidade do desenvolvimento, permitindo maior precisão na distribuição de elementos na tela. No entanto, essas metodologias frequentemente demandavam o emprego de artifícios técnicos, como o uso de *clearfixes*, e apresentavam inconsistências de renderização entre diferentes navegadores. Segundo [Pham \(2025\)](#), a complexidade de manutenção desses códigos e a fragilidade de suas implementações impulsionaram a comunidade de desenvolvedores na busca por soluções mais padronizadas e eficientes.

Nesse contexto de transição tecnológica, o surgimento do modelo *Flexbox* no [CSS3](#) marcou um avanço qualitativo na engenharia de interfaces. Ao propor um sistema de *layout* unidimensional focado na responsividade e na facilidade de manutenção, o *Flexbox* consolidou-se como a principal alternativa às técnicas rudimentares de flutuação. Essa trajetória de superação técnica — da rigidez das tabelas à flexibilidade do modelo unidimensional — estabeleceu as bases conceituais para a introdução do [CSS Grid](#). Este último, ao introduzir uma abordagem bidimensional nativa, redefine a concepção de design para a Web e constitui o objeto central de análise desta pesquisa.

Nesse contexto de transição tecnológica, o surgimento do modelo *Flexbox* no [CSS3](#) representou um avanço importante na forma de estruturar interfaces Web. Diferente das soluções anteriores, como tabelas e propriedades como *float*, o *Flexbox* foi projetado para organizar elementos em uma única dimensão, facilitando o alinhamento, a distribuição de espaço e a adaptação a diferentes tamanhos de tela. Essa proposta trouxe mais clareza ao código e reduziu a complexidade envolvida na construção de *layouts* responsivos.

A consolidação do modelo unidimensional abriu espaço para o [CSS Grid](#), que introduz uma abordagem bidimensional capaz de organizar linhas e colunas de forma estruturada. Essa mudança na lógica de organização espacial fundamenta o foco deste trabalho, que tem [CSS Grid](#) como objeto central de análise.

2.5.2 *Layouts* Baseados em Tabelas

A gênese do design de interface na Web foi marcada pela adaptação de elementos estruturais para fins puramente estéticos. Conforme analisa [Pham \(2025\)](#), o uso de tabelas (`<table>`) constituiu uma das estratégias primordiais para contornar a ausência de mecanismos nativos de posicionamento no [HTML](#), permitindo o escalonamento de conteúdos em grades de linhas e colunas. Entretanto, essa prática revelou-se tecnicamente onerosa ao longo do tempo.

A rigidez inerente às estruturas tabulares impunha barreiras severas à escalabilidade e à responsividade, uma vez que a fixidez das células inviabilizava o ajuste fluido do conteúdo a diferentes resoluções de tela. Sob a perspectiva do desenvolvimento, [Pham \(2025\)](#) destaca que o excesso de aninhamento de etiquetas (*tags*) não apenas inflava o volume de código, elevando o tempo de carregamento (*overhead*), mas também tornava a manutenção do sistema extremamente complexa.

Além disso, o uso de tabelas para a formatação visual subverte a lógica semântica da linguagem de marcação. Ao empregar um recurso destinado a dados tabulares para o controle de *layout*, os desenvolvedores comprometem a acessibilidade para leitores de tela e prejudicam a indexação por mecanismos de busca. Essa dissonância entre forma e função evidenciou a obsolescência desse método, impulsionando a transição para padrões que priorizassem a separação definitiva entre o conteúdo estrutural e a camada de apresentação visual.

2.5.3 *Layouts Baseados em Float*

A evolução das especificações de estilo permitiu que a propriedade *float* emergisse como uma alternativa técnica para superar o modelo tabular na estruturação de interfaces. Conforme documentado por [Mozilla \(2025a\)](#), embora originalmente concebida para permitir que textos circundassem imagens, essa propriedade foi adaptada para o controle de posicionamento, deslocando elementos para as extremidades laterais (*left* ou *right*) de seus contêineres e forçando o ajuste do fluxo documental ao espaço remanescente. Tal versatilidade viabilizou a criação sistemática de *layouts* em múltiplas colunas sem a subversão semântica das tabelas, garantindo, por um longo período, a interoperabilidade necessária em sistemas Web complexos ([Mozilla, 2025a](#)).

Contudo, essa transição para o uso de *floats* impôs desafios estruturais significativos, notadamente o colapso do contêiner pai decorrente da remoção dos elementos filhos do fluxo normal de renderização. Para mitigar essa falha, [Pham \(2025\)](#) destaca que os desenvolvedores padronizaram o uso do *clearfix*, uma técnica baseada na inserção de pseudo-elementos destinada a forçar o reconhecimento da altura do contêiner e evitar distorções visuais. Embora essencial na era dos *layouts* flutuantes, essa solução paliativa elevava a complexidade do código e dificultava a manutenção em projetos responsivos. Atualmente, a consolidação de sistemas nativos como o *Flexbox* e o *CSS Grid* tornou o *clearfix* obsoleto, permanecendo relevante apenas no contexto de suporte e compreensão de sistemas legados [Pham \(2025\)](#).

2.5.4 *Layouts Baseados em Inline-block*

A adoção do valor *inline-block* para a propriedade *display* consolidou-se como uma alternativa intermediária para a superação das dicotomias entre elementos de linha (*inline*) e de bloco (*block*). Diferentemente dos elementos estritamente de linha, o *inline-block* preserva a integridade do modelo de caixa (*box model*), permitindo o controle preciso de dimensões e margens sem a necessidade de recorrer a artifícios técnicos como o *clearfix*. Sob essa ótica, [Pham \(2025\)](#) argumenta que essa abordagem representou um marco na evolução das metodologias de *layout*, proporcionando maior estabilidade estrutural e flexibilidade no posicionamento de componentes sem comprometer a semântica documental.

Embora tenha apresentado um comportamento mais estável do que o uso de *floats*, o

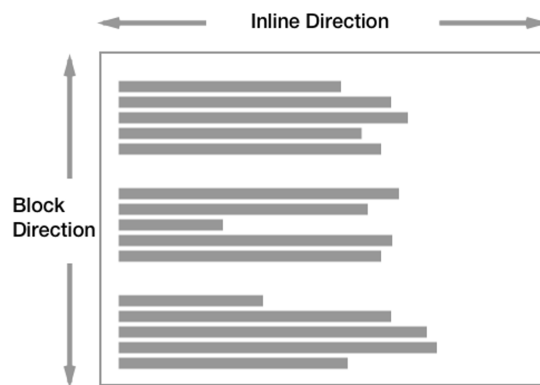
inline-block também revelou limitações quando aplicado a interfaces mais complexas. Conforme aponta [Coyier \(2016\)](#), um dos principais problemas estava na geração automática de espaços em branco entre os elementos, causada pelas quebras de linha no código-fonte. Além disso, havia restrições significativas nas opções de alinhamento vertical e na distribuição equilibrada de espaço entre os componentes.

Essas limitações reduziram a eficiência do método em *layouts* mais elaborados e responsivos, fazendo com que o *inline-block* se tornasse apenas uma solução intermediária dentro da evolução das técnicas de *layout*. Apesar de sua importância histórica no desenvolvimento do **CSS**, ele não foi suficiente para atender às exigências atuais de organização modular e adaptação a diferentes contextos de uso ([Coyier, 2016](#)).

2.5.4.1 Elementos de Bloco e de Linha

A organização visual de uma interface Web é regida pelos modos de escrita e pelos contextos de formatação definidos pelo **CSS**. Conforme detalha [Mozilla \(2025d\)](#), a disposição dos elementos de bloco está intrinsecamente ligada à direção do fluxo do texto. Em sistemas de escrita horizontais, como o português, os blocos são empilhados verticalmente, conforme mostrado na [Figura 2.1](#), onde cada novo componente se inicia abaixo do anterior.

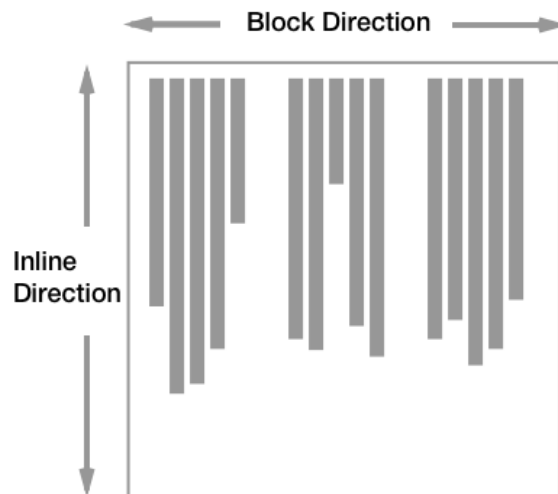
Figura 2.1 – Renderização de elementos de bloco em modo de escrita horizontal.



Fonte: Adaptado de [Mozilla \(2025d\)](#).

Essa lógica de posicionamento, contudo, é adaptável a diferentes culturas tipográficas. Em idiomas com escrita vertical, a orientação do fluxo é rotacionada, fazendo com que os elementos de bloco passem a ser alinhados horizontalmente, lado a lado, conforme demonstrado na [Figura 2.2](#). Essa flexibilidade evidencia que o comportamento de bloco não é estático, mas sim dependente do contexto de escrita estabelecido para o documento ([Mozilla, 2025d](#)).

Figura 2.2 – Inversão do fluxo de blocos em sistemas de escrita verticais.



Fonte: Adaptado de [Mozilla \(2025d\)](#).

A distinção entre as categorias de exibição é fundamental para o controle do *layout*, que é dividido em dois tipos: elementos de bloco e elementos de linha. A seguir, uma descrição de cada um:

- **Elementos de bloco (*block-level elements*):** São projetados para ocupar a largura total do contêiner e forçar uma quebra de linha. Estruturas como `<div>`, `<p>`, `<section>` e `<article>` permitem a manipulação completa das propriedades do modelo de caixa, incluindo altura, largura e margens.
- **Elementos de linha (*inline elements*):** Diferenciam-se por ocupar apenas o espaço necessário ao seu conteúdo, permitindo que múltiplos elementos coexistam na mesma linha. *Tags* como `<span>`, `<a>`, `<strong>` e `<em>` possuem restrições na alteração de dimensões físicas, sendo moldadas pelo fluxo textual.

A compreensão dessa base teórica, reforçada pelas evidências visuais das Figuras 2.1 e 2.2, é indispensável para o estudo de técnicas de posicionamento mais complexas, como o *Flexbox* e o *CSS Grid*, que surgiram justamente para oferecer um controle mais refinado sobre esses fluxos nativos.

2.5.5 Posicionamento Absoluto e Relativo

O controle de posicionamento via *CSS* estabeleceu uma das bases fundamentais para a customização de *layouts*, permitindo que desenvolvedores movessem elementos para além do fluxo documental padrão. Conforme as especificações da *W3C* (1998), a propriedade *position: relative* mantém o elemento em sua posição original dentro do fluxo, mas permite deslocamentos controlados sem afetar a disposição dos componentes adjacentes. Em contrapartida, a propriedade

position: absolute remove o elemento integralmente do fluxo normal, posicionando-o de forma coordenada em relação ao ancestral posicionado mais próximo ou, na ausência deste, ao *viewport*. Essa distinção técnica viabilizou avanços significativos na precisão das interfaces, possibilitando a implementação de elementos sobrepostos, menus flutuantes e componentes fixos em pontos estratégicos da tela (W3C, 1998).

Entretanto, a aplicação rigorosa desses métodos de posicionamento revelou lacunas críticas no que tange à escalabilidade e à adaptabilidade visual. Segundo W3C (1998), embora o posicionamento absoluto ofereça precisão milimétrica, sua natureza estática compromete a responsividade da interface, uma vez que elementos removidos do fluxo não se ajustam dinamicamente às variações de resolução de diferentes dispositivos. Da mesma forma, o posicionamento relativo carece de mecanismos nativos para a distribuição equitativa de espaço e o alinhamento complexo de múltiplos componentes. Assim, apesar de sua importância histórica para o design de precisão, essas técnicas evidenciaram a necessidade de modelos de *layout* mais robustos e fluidos, capazes de responder às demandas contemporâneas de dispositivos móveis.

2.5.6 *Flexbox: Modelo Unidimensional de Layout*

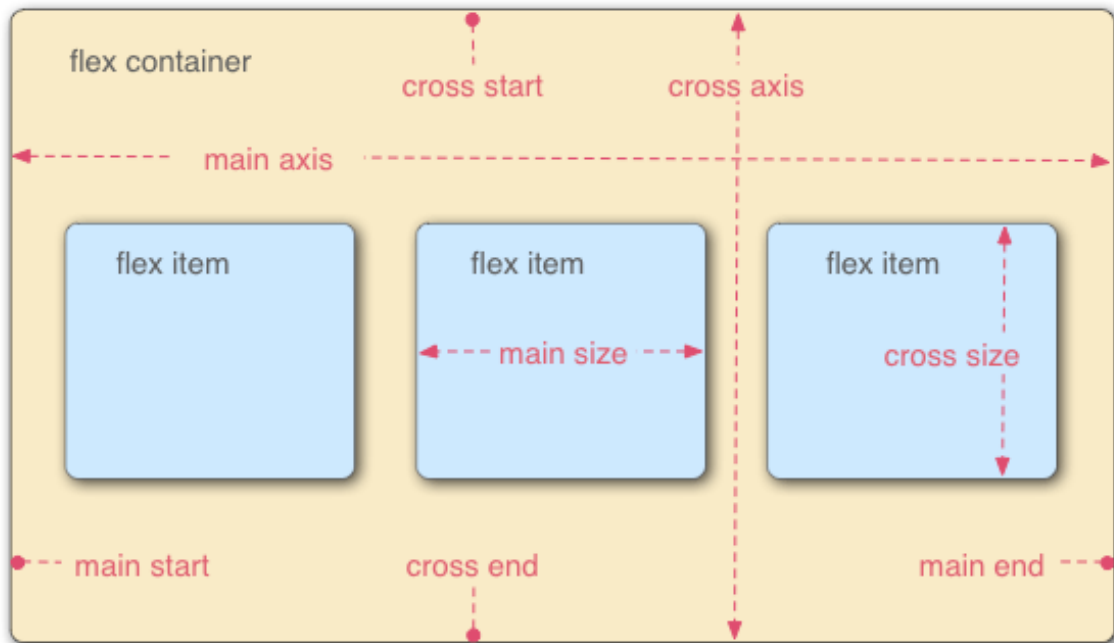
O surgimento do *Flexible Box Layout*, conhecido como *Flexbox*, representou uma mudança importante na construção de interfaces ao oferecer um sistema nativo capaz de superar as limitações das técnicas baseadas em flutuação e posicionamento. Conforme destaca Pham (2025), esse módulo do CSS3 foi desenvolvido para organizar elementos em uma única dimensão, atuando em linhas (*rows*) ou colunas (*columns*), proporcionando maior controle sobre alinhamento e distribuição de espaço em contêineres dinâmicos e tornando o desenvolvimento mais organizado (W3C, 2018).

A arquitetura deste modelo fundamenta-se na relação hierárquica entre o *flex container*, estabelecido pela propriedade *display: flex*, e seus respectivos *flex items*. De acordo com a Mozilla (2025c), essa estrutura permite que os elementos filhos se expandam ou se contraíam para preencher o espaço disponível, garantindo a integridade visual e a consistência da interface em cenários de resoluções variáveis. Tal comportamento responsivo resolve lacunas históricas de métodos anteriores, que frequentemente falhavam ao lidar com conteúdos de dimensões imprevisíveis.

A lógica operacional do *Flexbox* é regida pela dualidade entre o eixo principal (*main axis*) e o eixo transversal (*cross axis*). Segundo Coyier (2016), o *main axis* é definido pela propriedade *flex-direction* e determina o sentido da distribuição primária dos itens, enquanto o *cross axis* atua perpendicularmente, regendo o alinhamento transversal. Essa mecânica de eixos, exemplificada na Figura 2.3, proporciona uma versatilidade robusta, permitindo ajustes finos tanto na progressão linear quanto no nivelamento perpendicular dos componentes.

Em termos de aplicabilidade prática, Pham (2025) argumenta que o *Flexbox* demonstra

Figura 2.3 – Representação dos eixos principal (*main axis*) e transversal (*cross axis*) no modelo *Flexbox*.



Fonte: Adaptado de Mozilla (2025c).

eficácia superior na construção de componentes modulares, como barras de navegação, menus e sistemas de cartões, onde a organização unidimensional é preponderante. Embora sua natureza seja restrita a um único eixo por vez, sua ampla interoperabilidade com navegadores contemporâneos e a simplificação do código o consolidaram como um pilar do desenvolvimento moderno. Atualmente, o *Flexbox* é compreendido como uma tecnologia complementar ao *CSS Grid*, formando um conjunto de ferramentas que redefine a flexibilidade e a eficiência dos *layouts* responsivos (Mozilla, 2025c).

2.6 CSS Grid

O *CSS Grid Layout* configura-se como uma especificação do W3C que introduz um sistema de *layout* bidimensional nativo, revolucionando a organização de interfaces ao permitir o controle simultâneo de linhas e colunas. Conforme documentado por Mozilla (2025b), essa abordagem expande as fronteiras do design digital, viabilizando a fragmentação da página em grandes regiões estruturais e o estabelecimento de relações precisas de métricas, posicionamento e sobreposição de camadas. Embora compartilhe com o modelo tabular a capacidade de alinhar componentes em matrizes, a *Grid* dissocia-se da estrutura semântica do conteúdo, operando estritamente na camada de apresentação visual. Essa autonomia permite que elementos filhos de um *grid container* sejam posicionados, sobrepostos ou redistribuídos livremente, superando as restrições de fluxo de todas as técnicas predecessoras (Mozilla, 2025b).

A fundamentação conceitual do sistema, segundo Meyer e Weyl (2017), está estabelecida

na relação entre o *grid container* — elemento pai definido pela propriedade *display: grid* — e os *grid items*, que compreendem seus descendentes diretos. Por meio de propriedades como *grid-template-rows*, *grid-template-columns* e *grid-template-areas*, o desenvolvedor detém a capacidade de projetar arquiteturas visuais complexas de forma explícita. A flexibilidade deste modelo é potencializada por recursos avançados que garantem a independência entre a ordem de declaração no código-fonte e a renderização visual, permitindo que a hierarquia semântica do [HTML](#) permaneça intacta enquanto a interface é reorganizada dinamicamente (Meyer; Weyl, 2017).

Adicionalmente, a especificação introduz terminologias essenciais para a precisão técnica, como *grid tracks* (trilhas), *grid cells* (células resultantes da interseção de trilhas) e *grid areas* (regiões compostas por agrupamentos de células). De acordo com a documentação da [Mozilla \(2025b\)](#), funções lógicas como *repeat()* e *minmax()* conferem uma escalabilidade sofisticada ao sistema, permitindo a automação de padrões repetitivos e a definição de limites dimensionais fluidos. Complementado por propriedades de alinhamento, como *justify-content* e *align-items*, além do controle de espaçamento via *gap* (espaçamentos de separação), a *Grid* consolida-se como uma ferramenta robusta para o design responsivo contemporâneo. Em última análise, o *CSS Grid Layout* substitui as soluções paliativas acumuladas nas últimas décadas, estabelecendo um novo padrão de eficiência e clareza conceitual para a engenharia de interfaces digitais (Mozilla, 2025b).

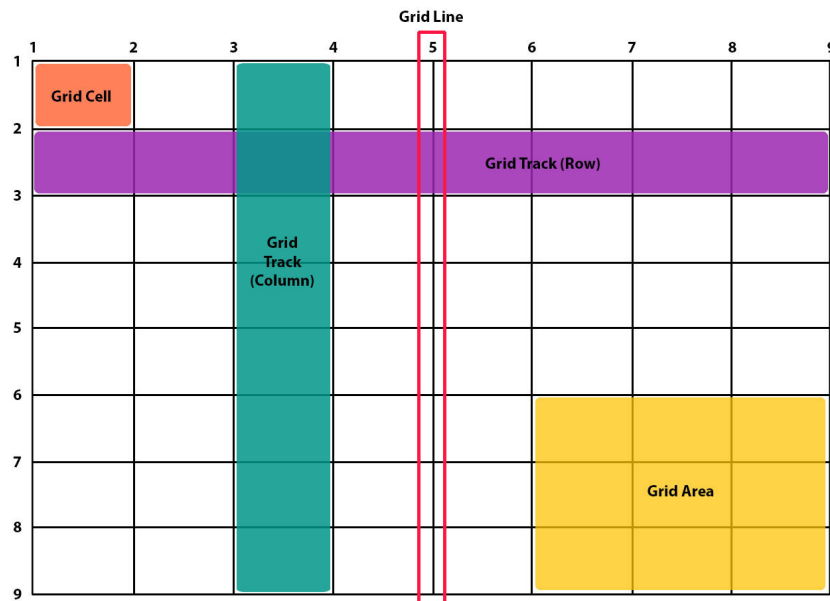
2.6.1 Componentes de um *CSS Grid Layout*

A aplicação correta do *CSS Grid* depende, fundamentalmente, do domínio de sua terminologia técnica, conforme destacam Meyer e Weyl (2017). Compreender esses conceitos básicos é o que permite estruturar *layouts* em duas dimensões (linhas e colunas) de forma precisa, impedindo erros comuns na distinção entre os elementos que compõem a interface.

De acordo com a documentação da [MDN Web Docs \(2025\)](#), o funcionamento do *CSS Grid* baseia-se em uma estrutura de componentes que trabalham juntos para organizar o espaço visual. A [Figura 2.4](#) detalha essa organização, mostrando como o “contêiner” (o elemento pai) gerencia suas subdivisões internas. A imagem exemplifica a hierarquia entre linhas, faixas (*tracks*), células e áreas, revelando como cada parte se encaixa para permitir um posicionamento flexível dos itens.

Cada componente apresentado na [Figura 2.4](#) tem sua função prática na construção de *layouts* que se adaptam a diferentes telas e mantêm uma estrutura lógica. Eles podem ser descritos da seguinte forma:

Figura 2.4 – Representação visual dos principais conceitos e terminologias.



Fonte: Codepen (2025).

2.6.1.1 *Grid Container*

O *Grid Container* é o elemento pai que estabelece o contexto de formatação da *grid*, sendo responsável por organizar os itens segundo as regras do [CSS Grid](#). Ao aplicar *display: grid* ou *display: inline-grid*, cria-se uma estrutura que define linhas e colunas, além de propriedades como espaçamento (*gap*) e alinhamento (*justify-content*, *align-items*).

A sua estrutura pode ser observada na [Figura 2.4](#), onde toda a área delimitada pela borda externa da grade representa o *grid container*. Dentro dessa área, estão dispostas as linhas numeradas que formam a base para os **grid tracks**, **grid cells** e **grid areas**. A imagem evidencia que o contêiner não é apenas um elemento visual, mas a camada estrutural que sustenta toda a malha, permitindo que os itens sejam posicionados de forma precisa e flexível.

2.6.1.2 *Grid Items*

Os *Grid Items* são os elementos posicionados dentro da *grid*, sendo obrigatoriamente os filhos diretos do contêiner. Na estrutura apresentada pela [Figura 2.4](#), eles ocupam os espaços gerados pelo cruzamento entre as linhas e as colunas. Na prática, cada “caixa” de conteúdo colocada dentro da *grid* torna-se um item que pode ser organizado de duas formas: automaticamente, onde o navegador decide o local de cada um, ou manualmente, utilizando comandos como *grid-column* e *grid-row* para definir exatamente quais células devem ser preenchidas.

Um ponto crucial da estrutura é que essa relação de organização só funciona com os elementos de primeiro nível, ou seja, apenas os filhos diretos são controlados pela *grid*, enquanto os elementos dentro desses itens seguem as regras normais de posicionamento, a menos que também sejam transformados em novos *grids*.

2.6.1.3 *Grid Line*

As *Grid Lines*, ou linhas de grade, constituem as unidades de divisão conceitual que estruturam o sistema de *grid*, delimitando as colunas e linhas no interior do contêiner. Elas operam como eixos de referência fundamentais para o endereçamento de componentes, viabilizando o posicionamento preciso dos elementos na interface. Nesse modelo, a localização de um item é determinada pela especificação de suas coordenadas de início e fim em relação a essas linhas. Por exemplo, ao aplicar a propriedade *grid-column: 1 / 3*, define-se que o elemento deve se estender da primeira à terceira linha de grade no eixo horizontal, ocupando o intervalo espacial correspondente a essas demarcações.

Na [Figura 2.4](#), essas linhas são representadas pelos números dispostos nas bordas da grade (de 1 a 9), que indicam a sequência das linhas. A linha vertical número 5, destacada em vermelho, exemplifica como as *grid lines* atuam como marcadores visuais para definir a posição dos itens, servindo como limites entre colunas e linhas. Essa representação evidencia a função das *grid lines* como base estrutural do [CSS Grid](#), permitindo ao desenvolvedor controlar a disposição dos elementos de forma precisa e semântica.

2.6.1.4 *Grid Track*

O *Grid Track* é o nome dado ao espaço formado entre duas linhas paralelas da grade. Ele representa a estrutura real das linhas e colunas dentro do *layout*. Como ilustrado na [Figura 2.4](#), as faixas funcionam de duas formas: a área verde mostra um *track* vertical (coluna), que se estende de cima a baixo no contêiner, enquanto a área roxa indica um *track* horizontal (linha), que atravessa toda a largura da *grid*.

Na prática, essas faixas são os espaços onde o conteúdo será distribuído, e seu tamanho é controlado diretamente pelas propriedades *grid-template-columns* e *grid-template-rows*. Ao definir essas propriedades, determina-se exatamente qual será a largura de cada coluna e a altura de cada linha, criando a base estrutural que sustenta todo o desenho da interface.

2.6.1.5 *Grid Cell*

Uma *Grid Cell* (célula da grade) é a unidade básica e indivisível do *layout*, funcionando como o “bloco” fundamental da *grid*. Ela é criada automaticamente sempre que uma linha horizontal e uma coluna vertical se cruzam. Na [Figura 2.4](#), essa unidade é exemplificada pela área destacada em laranja no canto superior esquerdo.

Na prática, a célula serve como a base de posicionamento: cada elemento colocado na *grid* ocupará, no mínimo, o espaço de uma célula, podendo expandir-se para várias outras, se necessário. Embora não exista um comando no [CSS](#) para estilizar uma célula isolada, ela é essencial para a lógica do sistema, pois define as coordenadas mínimas que permitem organizar o conteúdo com precisão total dentro da interface.

2.6.1.6 *Grid Area*

A *Grid Area* (área da grade) é um espaço retangular composto pelo agrupamento de uma ou mais células, sendo delimitada por quatro linhas da *grid* (duas horizontais e duas verticais). Em termos práticos, ela serve para definir seções maiores e funcionais no *layout*, como cabeçalhos ou barras laterais.

Conforme ilustrado na [Figura 2.4](#) pela cor amarela, a área se diferencia da célula (destacada em laranja) por sua capacidade de expandir-se e ocupar vários blocos simultaneamente. Uma característica importante é que essas regiões podem receber nomes específicos através da propriedade *grid-template-areas*. Dessa forma, é possível posicionar os elementos apenas chamando o nome da área correspondente, o que torna a estrutura do código muito mais clara e fácil de manter.

2.6.2 Propriedades do Contêiner e dos Itens no *CSS Grid*

A definição clara da separação de papéis entre o contêiner e os itens é um aspecto central para compreender a lógica do *CSS Grid*. Essa divisão define quem controla a estrutura geral e quem controla o posicionamento individual. As propriedades aplicadas ao contêiner são responsáveis por criar a *grid* em si: elas determinam o número de linhas e colunas, o tamanho de cada faixa, os espaços de separação (*gaps*) e como todo o conjunto se alinha na tela. Em síntese, o contêiner define os parâmetros operacionais do sistema, provendo a base estrutural sobre a qual os elementos serão organizados.

Por outro lado, as propriedades dos itens servem para ajustar o comportamento de cada elemento específico dentro dessa *grid*. Elas permitem definir, por exemplo, se um item deve ocupar mais de uma coluna ou se deve ficar alinhado de forma diferenciada dentro de sua célula. Essa organização em dois níveis é o que torna o *CSS Grid* tão eficiente: um *layout* global consistente e, ao mesmo tempo, a possibilidade de mover ou redimensionar peças individuais sem quebrar o restante da estrutura. Nos tópicos a seguir, as funções serão detalhadas de forma prática para cada uma dessas propriedades.

2.6.3 Propriedades do Contêiner Pai

O contêiner atua como a base estrutural do sistema, sendo o elemento encarregado de instituir o contexto da *grid* e gerenciar sua configuração. Conforme detalhado na documentação da [Mozilla \(2025b\)](#), essa função é exercida por meio de um conjunto de propriedades que determinam a organização bidimensional do espaço. Em suma, o contêiner estabelece as diretrizes normativas da malha, configurando o espaço estrutural necessário para a distribuição sistemática do conteúdo. Entre as principais propriedades voltadas à definição dessa arquitetura, destacam-se: (1) *display:grid* e *display:inline-grid*, (2) *grid-template-columns*, (3) *grid-template-row*, (4)

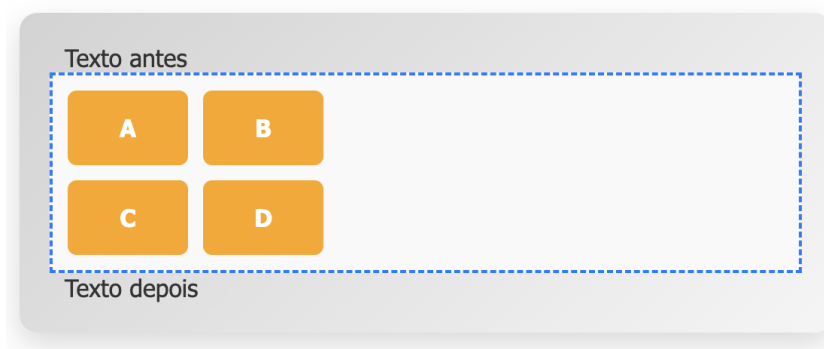
grid-template-areas, (5) *gap* (*row-gap* e *column-gap*), (6) *justify-content*, (7) *align-content*, (8) *justify-items*, e (9) *align-items*. Todos estes são descritos a seguir.

2.6.3.1 Propriedades *display: grid* e *display: inline-grid*

A utilização da propriedade *display* com os valores *grid* ou *inline-grid* constitui o passo inicial para a ativação do sistema de *grid*. Esta declaração altera a natureza do elemento para um *grid container*, submetendo todos os seus filhos diretos às regras de posicionamento da grade.

O valor *grid* configura o contêiner como um elemento de nível de bloco. Na prática, isso significa que ele ocupará toda a largura disponível em seu contêiner pai, forçando a quebra de linha para qualquer conteúdo adjacente. Conforme demonstrado na Figura 2.5, o contêiner (delimitado pela borda tracejada) expande-se horizontalmente, o que faz com que o elemento “Texto depois” seja deslocado para a linha subsequente, isolando a grade no fluxo vertical.

Figura 2.5 – Representação visual do comportamento em bloco com *display: grid*.



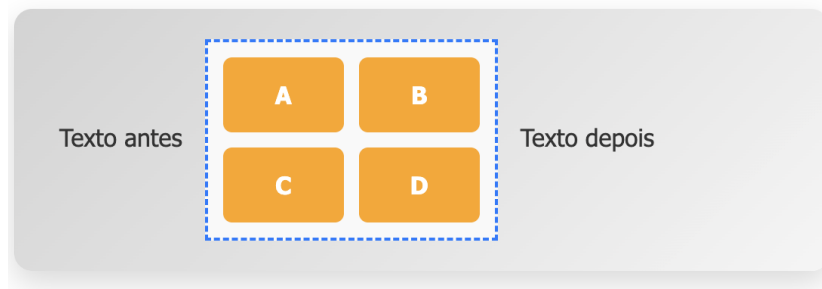
Fonte: própria autora.

Por outro lado, o valor *inline-grid* permite que o contêiner mantenha a organização interna da grade, mas se comporte externamente como um elemento de linha. Como ilustrado na Figura 2.6, nesta configuração a largura do contêiner é limitada ao espaço estritamente necessário para acomodar seus itens. Isso possibilita que elementos de texto ou outros componentes *inline* permaneçam na mesma linha que o contêiner, garantindo a continuidade do fluxo sem interrupções bruscas. Essa distinção é fundamental para o desenvolvimento de interfaces onde pequenos módulos de grade precisam coexistir horizontalmente com textos ou outros elementos gráficos.

2.6.3.2 Propriedades *grid-template-columns* e *grid-template-rows*

As propriedades *grid-template-columns* e *grid-template-rows* definem a estrutura bidimensional do *grid*, sendo responsáveis, respectivamente, pela configuração das colunas e das linhas. Enquanto a primeira organiza a divisão vertical do espaço, a segunda controla a segmentação horizontal. Ambas utilizam a mesma lógica sintática e permitem a definição explícita das trilhas (*tracks*) por meio de diferentes unidades de medida, oferecendo precisão e flexibilidade na construção do *layout*.

Figura 2.6 – Representação visual do comportamento em linha com *display: inline-grid*.



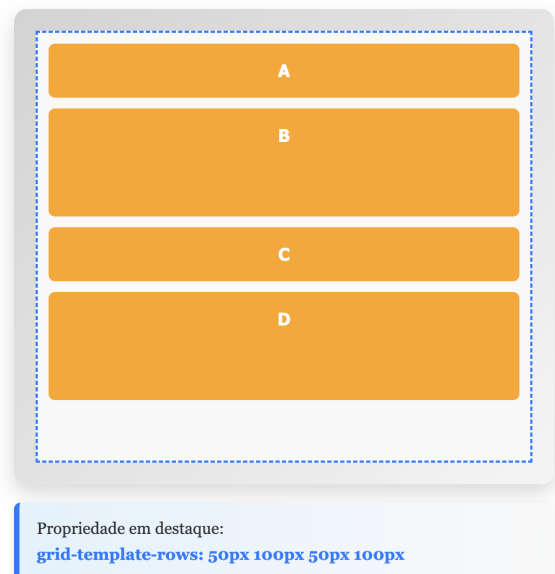
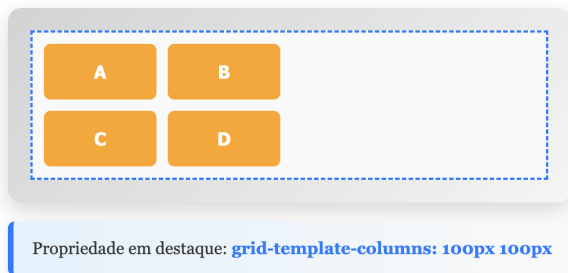
Fonte: própria autora.

- *Valores Fixos (px)*

A utilização de valores em pixels estabelece dimensões rígidas e invariáveis. No caso das colunas, a configuração *100px 100px* mantém larguras constantes, conforme ilustrado na [Figura 2.7](#). Para as linhas, valores como *50px 100px 50px 100px* produzem alturas fixas, como mostrado na [Figura 2.8](#). Embora proporcionem controle preciso, essas definições reduzem a adaptabilidade em diferentes tamanhos de tela.

Figura 2.8 – Configuração com linhas fixas.

Figura 2.7 – Configuração com colunas fixas.



Fonte: própria autora.

- *Valores relativos (%)*

O uso de porcentagens permite distribuir o espaço proporcionalmente ao tamanho do contêiner. A configuração *50% 50%* divide horizontalmente o espaço de forma equilibrada, como demonstrado na [Figura 2.9](#). De maneira semelhante, a regra *20% 30% 30% 20%* organiza verticalmente as linhas conforme ilustrado na [Figura 2.10](#). Essa abordagem torna o *layout* mais fluido, embora exija atenção ao modelo de caixa para evitar transbordamentos.

Figura 2.9 – Distribuição proporcional utilizando unidades percentuais.

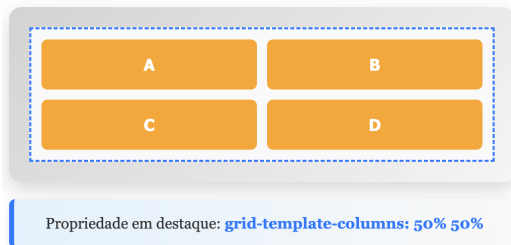
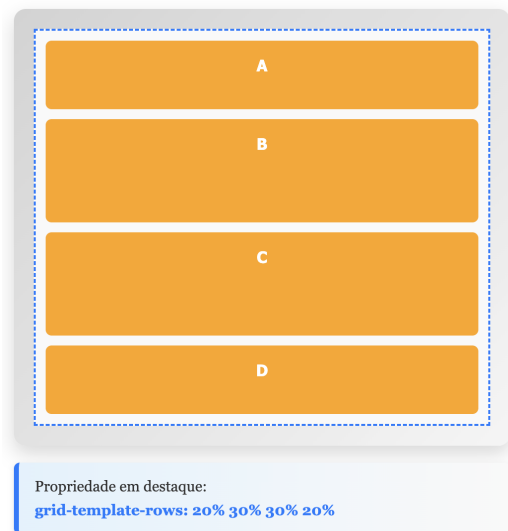


Figura 2.10 – Distribuição vertical proporcional via porcentagem.



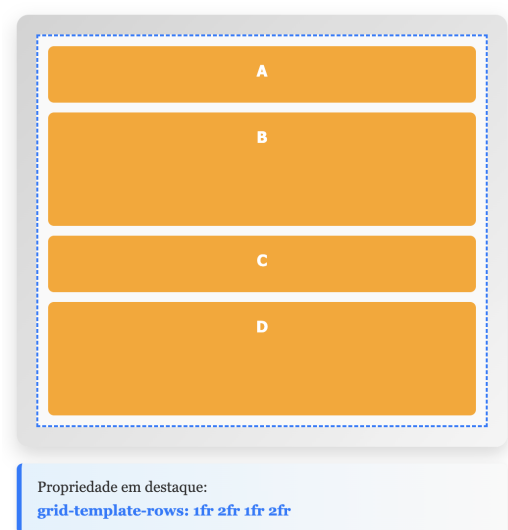
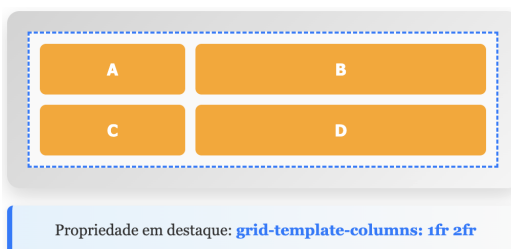
Fonte: própria autora.

- *Unidade Fracionária (fr)*

A unidade *fr* representa frações do espaço disponível e é exclusiva do sistema de *grid*. No eixo horizontal, a regra *1fr 2fr* distribui o espaço proporcionalmente entre as colunas, como mostrado na Figura 2.11. No eixo vertical, a configuração *1fr 2fr 1fr 2fr* organiza as linhas de maneira proporcional, conforme a Figura 2.12. Essa unidade é amplamente utilizada em *layouts* responsivos por permitir cálculos automáticos pelo navegador.

Figura 2.12 – Escalonamento vertical dinâmico com unidades fracionárias.

Figura 2.11 – Divisão dinâmica de espaço via unidades fracionárias.



Fonte: própria autora.

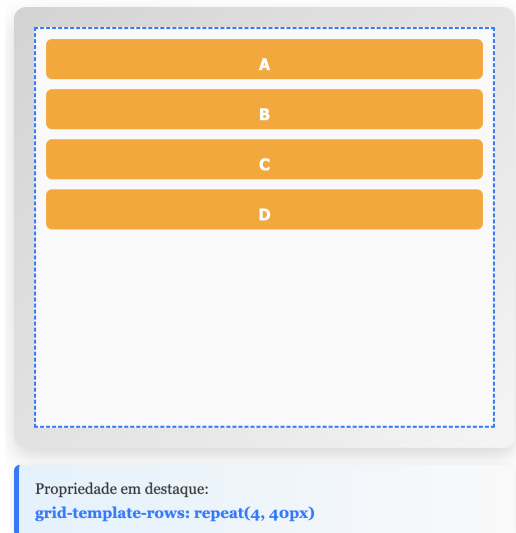
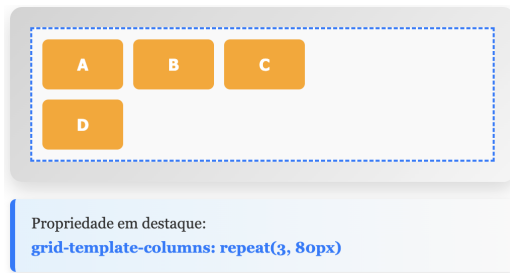
- *Função repeat()*

A função *repeat()* simplifica a declaração de múltiplas trilhas idênticas. A instrução *repeat(3, 80px)* cria três colunas uniformes, conforme ilustrado na Figura 2.13. De forma

equivalente, `repeat(4, 40px)` estabelece quatro linhas com a mesma altura, como apresentado na Figura 2.14. Esse recurso contribui para a organização e legibilidade do código.

Figura 2.14 – Padronização de alturas via função de repetição.

Figura 2.13 – Simplificação estrutural com o uso da função de repetição.



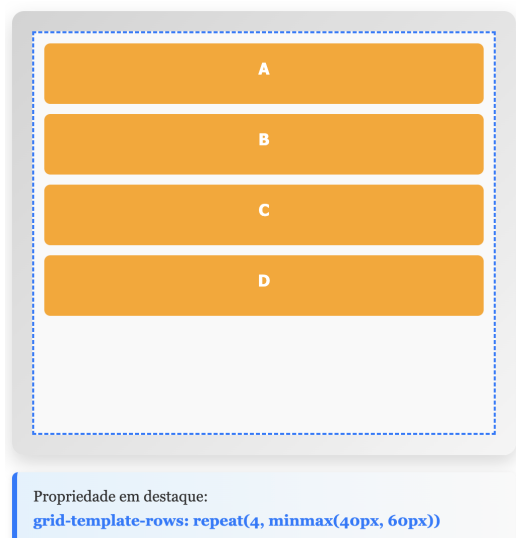
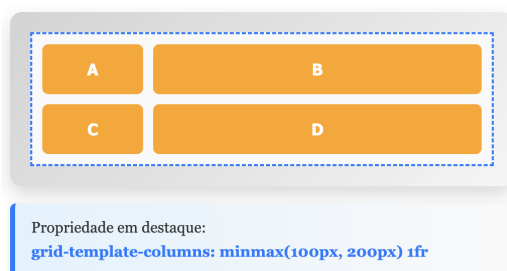
Fonte: própria autora.

- *Função minmax()*

A função `minmax()` define limites mínimo e máximo para as trilhas. No eixo horizontal, a configuração `minmax(100px, 200px)` impede que a coluna ultrapasse os valores estabelecidos, como observado na Figura 2.15. No eixo vertical, `minmax(40px, 60px)` garante que as linhas mantenham legibilidade sem perder flexibilidade, conforme a Figura 2.16. Essa técnica é essencial para manter a estabilidade visual em interfaces adaptativas.

Figura 2.16 – Gestão de limites verticais com a função `minmax()`.

Figura 2.15 – Controle de limites dimensionais com `minmax()`.



Fonte: própria autora.

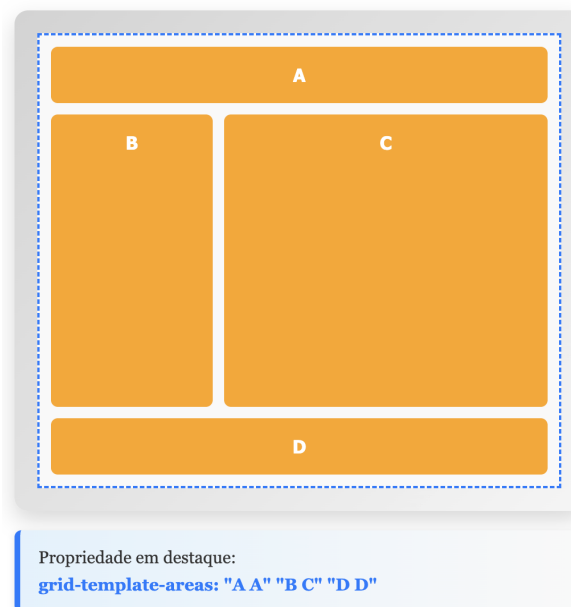
2.6.3.3 Propriedade *grid-template-areas*

A propriedade *grid-template-areas* estabelece uma lógica declarativa ao sistema de *grid*, permitindo a atribuição de identificadores textuais a regiões específicas da malha. Conforme observa a [Mozilla \(2025b\)](#), essa técnica substitui a dependência de índices numéricos por um mapeamento nominal, aproximando a estrutura do código da concepção visual do design.

A eficácia dessa funcionalidade é demonstrada pela instrução "A A" "B C" "D D", que projeta uma matriz organizada em três níveis horizontais e duas colunas. Como ilustrado na [Figura 2.17](#), a distribuição espacial ocorre da seguinte forma:

- **Área A:** Atua como um cabeçalho consolidado, ocupando a totalidade da primeira trilha horizontal mediante a expansão por ambas as colunas.
- **Áreas B e C:** Segmentam a segunda linha da grade, permitindo a coexistência de fluxos de conteúdo distintos em um arranjo lateral.
- **Área D:** Configura-se como a base estrutural, estendendo-se integralmente pela largura do contêiner na última linha.

Figura 2.17 – Mapeamento visual e organização de áreas nomeadas.



Fonte: própria autora.

Sob o ponto de vista da engenharia de software, a principal vantagem dessa abordagem reside na dissociação entre a hierarquia do conteúdo [HTML](#) e sua disposição geométrica ([CSS](#)). Essa independência é crucial para o desenvolvimento de interfaces responsivas: a reordenação completa de seções de uma página pode ser executada estritamente via [CSS](#), redefinindo o mapa de nomes sem a necessidade de manipular o [DOM](#). Tal característica assegura uma manutenção

ágil e preserva a acessibilidade, mantendo a ordem lógica do código-fonte inalterada (Mozilla, 2025b).

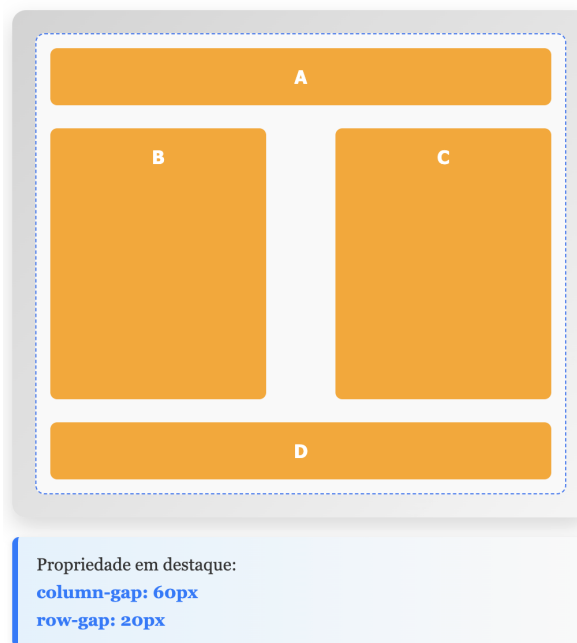
2.6.3.4 Propriedade *gap* (*row-gap* e *column-gap*)

A propriedade *gap* define os espaços entre as trilhas da grade. Em vez de aplicar margens individualmente em cada item, o espaçamento é configurado diretamente no contêiner, o que garante maior controle sobre a organização interna da *grid*. Dessa forma, evita-se o acúmulo de espaços nas bordas externas e mantém-se uma distribuição mais equilibrada do *layout*, visto que o navegador considera esses espaços antes de calcular o espaço disponível entre as células.

A operacionalização desse controle é evidenciada na Figura 2.18, que ilustra um *layout* organizado em três níveis. A análise técnica do exemplo revela a seguinte configuração:

- ***row-gap: 20px***: Estabelece a segmentação vertical entre as linhas, assegurando um distanciamento uniforme que isola o Item A (cabeçalho) da seção intermediária (Itens B e C) e esta do Item D (base).
- ***column-gap: 60px***: Define o espaçamento horizontal entre as colunas, criando um espaço entre os elementos B e C, sem interferir no alinhamento das bordas externas do contêiner.

Figura 2.18 – Controle de espaçamento estrutural via *column-gap* e *row-gap*.



Fonte: própria autora.

A centralização do controle de distanciamento no *grid container* é fundamental para a integridade do design responsivo atualmente. Esta metodologia garante que a proporção dos espaços permaneça constante durante o redimensionamento do *viewport*, preservando a hierarquia

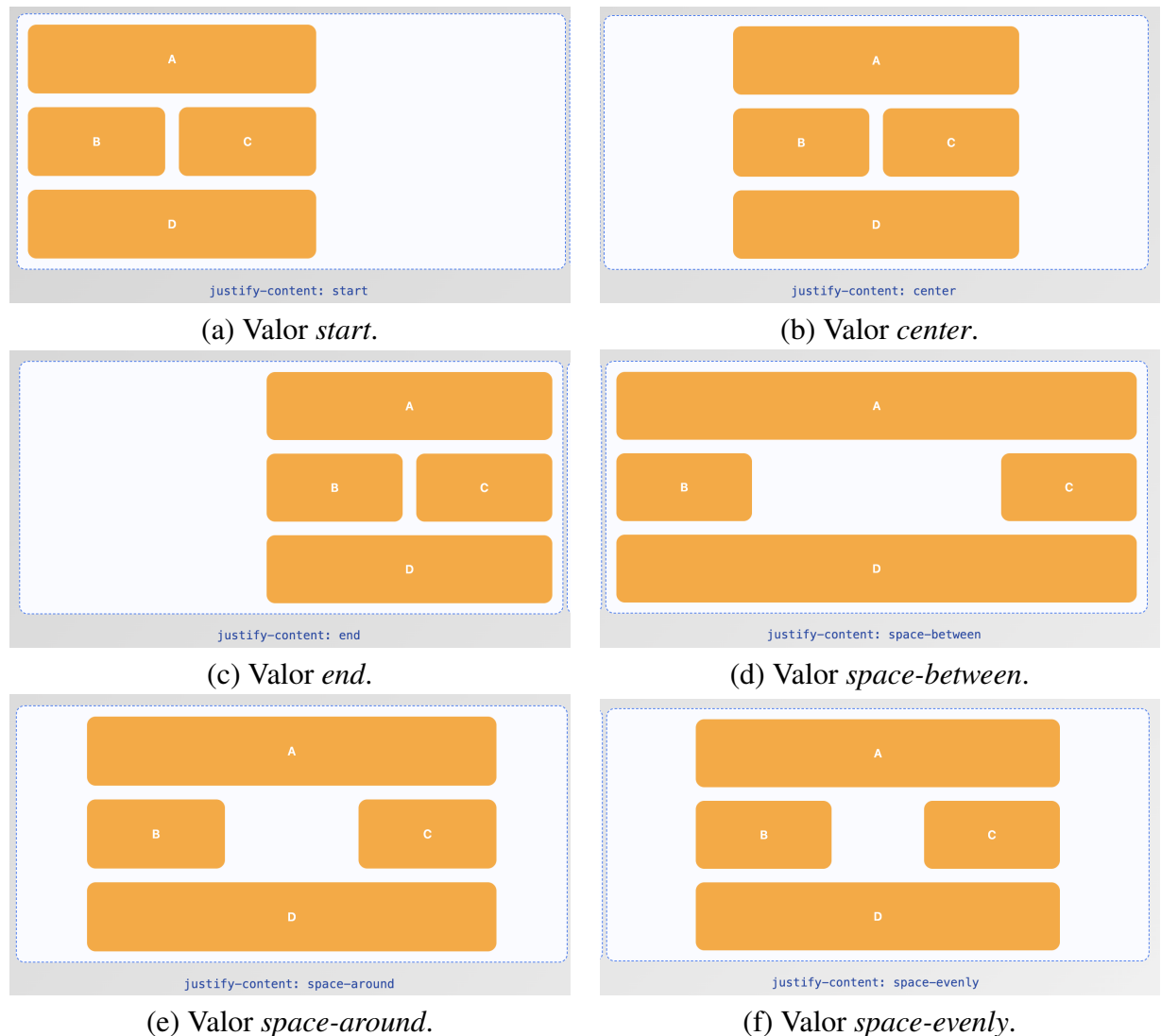
visual. Além disso, otimiza a manutenção do sistema, visto que ajustes na densidade da malha podem ser executados globalmente em uma única instrução, sem repercussões negativas no posicionamento individual de cada componente da interface.

2.6.3.5 Propriedade *justify-content*

Esta propriedade coordena o alinhamento dos itens ao longo do eixo principal. Ela é fundamental para gerenciar o espaço restante no contêiner quando a largura total dos itens é menor do que a largura disponível.

- *start*: Promove o agrupamento de todos os itens no início do eixo principal. Como ilustrado na [Figura 2.19 \(a\)](#), os componentes são deslocados para a extremidade esquerda do contêiner, resultando na concentração integral do espaço residual à direita.
- *center*: Estabelece a centralização do bloco de conteúdo em relação à largura total da malha. Na [Figura 2.19 \(b\)](#), observa-se que o espaço excedente é particionado equitativamente entre as margens esquerda e direita, posicionando o conjunto de itens no eixo central.
- *end*: Projeta os itens para o limite final do eixo horizontal. Conforme demonstrado na [Figura 2.19 \(c\)](#), o conteúdo é ancorado à direita, o que desloca o vazio estrutural para o início do contêiner.
- *space-between*: Realiza a distribuição do espaço livre exclusivamente nos intervalos entre os itens. A [Figura 2.19 \(d\)](#) evidencia que os elementos das extremidades (A e D) permanecem colados às bordas, enquanto o distanciamento ocorre apenas entre os componentes adjacentes internos.
- *space-around*: Distribui o espaço ao redor de cada elemento de forma que os intervalos externos correspondam à metade do valor dos intervalos internos. Na [Figura 2.19 \(e\)](#), verifica-se que a distância entre os itens B e C é o dobro daquela observada entre os itens e as bordas do contêiner.
- *space-evenly*: Garante a equanimidade matemática em todos os intervalos da grade. Como demonstrado na [Figura 2.19 \(f\)](#), o espaçamento entre a borda e o primeiro item é rigorosamente idêntico aos vãos entre os itens subsequentes e à borda final, resultando em simetria absoluta.

Figura 2.19 – Comportamento padrão dos valores da propriedade *justify-content*.



Fonte: própria autora.

2.6.3.6 Propriedade *align-content*

A propriedade *align-content* coordena o alinhamento e a distribuição das trilhas de linha (*grid rows*) ao longo do eixo transversal (vertical). De acordo com as especificações da [Mozilla \(2025b\)](#), sua aplicação é indispensável em cenários onde a altura total das linhas é inferior à altura disponível no *grid container*, permitindo a gestão eficiente do espaço excedente. A lógica operacional deste recurso é análoga à de *justify-content*, porém aplicada à dimensão vertical da interface:

- *start*: Promove o agrupamento de todas as linhas no topo do contêiner. Como ilustrado na [Figura 2.20 \(a\)](#), o conjunto de linhas é deslocado para o início do eixo transversal, resultando na concentração do espaço residual na parte inferior.
- *center*: Estabelece a centralização vertical do bloco de linhas. Na [Figura 2.20 \(b\)](#), observa-se que o espaço excedente é particionado equitativamente acima e abaixo do conteúdo,

estabilizando a malha no centro do eixo vertical.

- *end*: Projete todas as linhas para a base do contêiner. Conforme demonstrado na [Figura 2.20](#) (c), o conteúdo é ancorado no limite inferior, o que desloca o vazio estrutural para a extremidade superior do *layout*.
- *space-between*: Distribui o espaço livre exclusivamente nos intervalos entre as linhas. A [Figura 2.20](#) (d) evidencia que a primeira linha permanece colada ao topo e a última à base, enquanto o distanciamento ocorre apenas nas interseções internas.
- *space-around*: Aloca o espaço de maneira uniforme ao redor de cada linha. Como observado na [Figura 2.20](#) (e), os vazios nas extremidades superior e inferior correspondem à metade do valor do espaçamento aplicado entre as linhas adjacentes.
- *space-evenly*: Garante a equanimidade matemática em todos os intervalos verticais. Na [Figura 2.20](#) (f), demonstra-se uma simetria perfeita entre as bordas do contêiner e os vãos situados entre as trilhas.
- *stretch*: Representa o comportamento padrão em 2026 para linhas com dimensões flexíveis. Conforme a [Figura 2.20](#) (g), as linhas expandem-se verticalmente para preencher a totalidade da altura do contêiner, eliminando espaços ociosos.

Figura 2.20 – Comportamento padrão dos valores da propriedade *align-content*.



Fonte: própria autora.

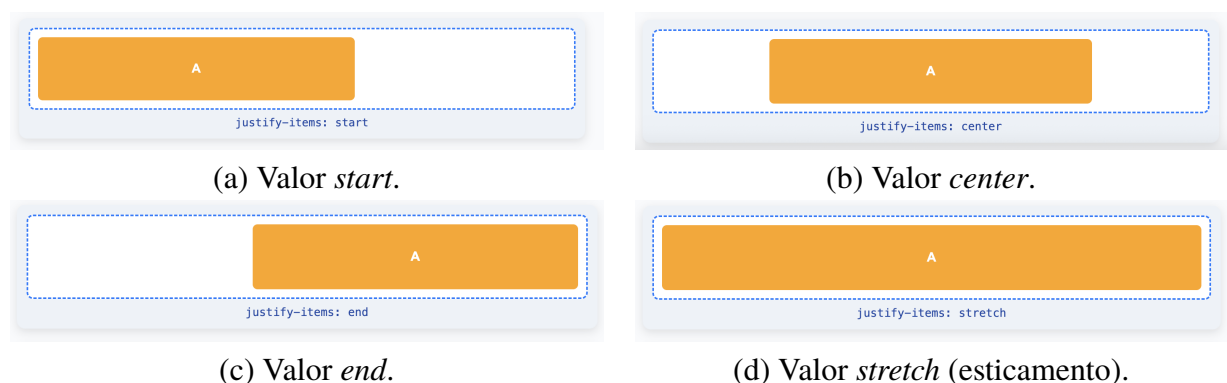
2.6.3.7 Propriedade *justify-items*

A propriedade *justify-items* coordena o alinhamento dos itens individuais dentro de suas respectivas células da grade, operando ao longo do eixo horizontal (eixo principal). Conforme as especificações da [Mozilla \(2025b\)](#), este recurso é indispensável para determinar a posição exata de um componente quando sua largura intrínseca é inferior ao volume total da célula que ele ocupa.

Diferente das propriedades de alinhamento de conteúdo, que gerenciam a massa total de itens, a *justify-items* atua de forma granular em cada unidade da malha. Para fins de demonstração técnica nas ilustrações subsequentes, utilizou-se uma configuração de grade simplificada, restrita a uma única célula, visando isolar e evidenciar o comportamento do alinhamento sob as seguintes diretrizes:

- *start*: Esta configuração determina a fixação do componente no início da célula. Conforme a [Figura 2.21 \(a\)](#), o motor de renderização posiciona o item rente à margem esquerda, forçando o deslocamento de todo o volume de espaço remanescente para o oposto da unidade.
- *center*: A aplicação deste valor resulta no equilíbrio axial do elemento. Na [Figura 2.21 \(b\)](#), verifica-se que o sistema particiona as lacunas horizontais em frações idênticas, neutralizando a posição do conteúdo no epicentro da célula e assegurando simetria bilateral.
- *end*: Esta instrução comanda o recuo do item em direção ao limite terminal do eixo principal. Como se observa na [Figura 2.21 \(c\)](#), o conteúdo é disposto no extremo direito do espaço disponível, resultando na concentração do espaço excedente na margem esquerda da célula.
- *stretch*: O valor em questão define a máxima ocupação permitida pela arquitetura da malha. Pela análise da [Figura 2.21 \(d\)](#), nota-se que o componente abdica de suas dimensões intrínsecas para ocupar integralmente o espaço da célula.

Figura 2.21 – Comportamento padrão das propriedades *justify-items*.



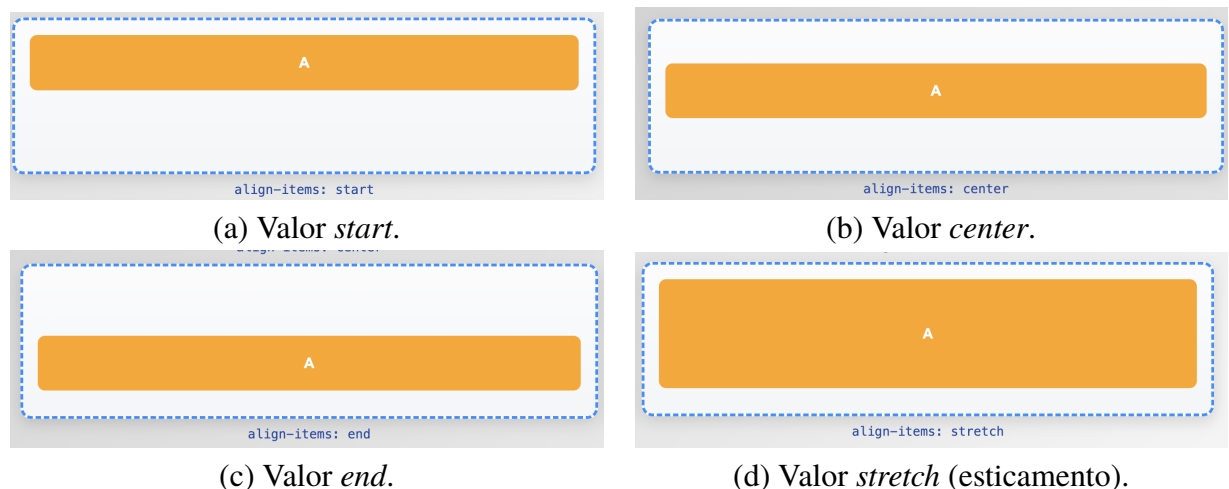
Fonte: própria autora.

2.6.3.8 Propriedade *align-items*

A propriedade *align-items* atua na regulação do posicionamento dos componentes individuais dentro de suas respectivas células, operando estritamente sobre o eixo transversal (vertical). Sob uma perspectiva didática, as evidências visuais subsequentes sintetizam o comportamento deste alinhamento em uma estrutura simplificada de unidade única (1x1). Os principais valores que essa propriedade pode assumir são:

- *start*: Este parâmetro dita a ascensão do elemento ao limite superior do eixo transversal. Como evidenciado na [Figura 2.22 \(a\)](#), o objeto adere à fronteira inicial da trilha, o que projeta todo o vazio estrutural para o setor inferior da unidade, mantendo a integridade do topo.
- *center*: A referida opção mantém o elemento equilibrado no meio da célula. Como se observa na [Figura 2.22 \(b\)](#), os espaços vazios acima e abaixo do conteúdo são iguais, garantindo que o item fique posicionado exatamente à mesma distância do topo e da base da unidade.
- *end*: Este ajuste puxa o elemento para o topo da célula. Como mostra a [Figura 2.22 \(c\)](#), o objeto fica encostado na parte superior, o que faz com que todo o espaço restante fique acumulado na parte inferior da unidade.
- *stretch*: Esta opção faz o componente se esticar para ocupar todo o espaço. Pela [Figura 2.22 \(d\)](#), nota-se que o elemento cresce verticalmente até tocar as bordas superior e inferior, eliminando qualquer vazio e preenchendo a altura total da célula.

Figura 2.22 – Comportamento padrão dos valores da propriedade *align-items*.



Fonte: própria autora.

2.6.4 Propriedades dos Itens Filhos

Enquanto o contêiner define a estrutura geral da *grid*, seus elementos internos, representados pelos filhos diretos, possuem propriedades específicas que permitem ajustar posicionamento e dimensões dentro do *layout*. De acordo com [Mozilla \(2025b\)](#), essas propriedades possibilitam que cada item ocupe múltiplas células ou seja alinhado de maneira particular em determinados eixos, ampliando a flexibilidade na organização do conteúdo.

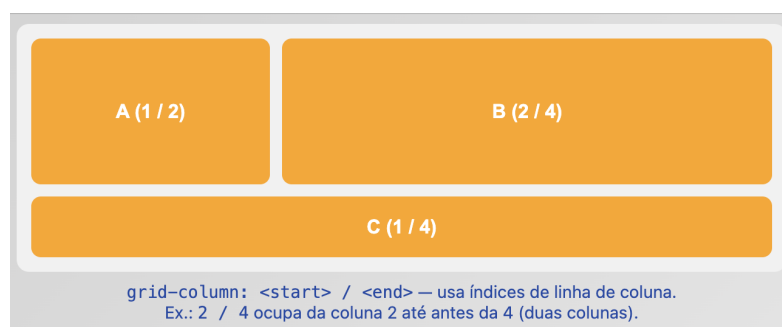
Dessa forma, a manipulação das propriedades dos itens é o que viabiliza a criação de *layouts* assimétricos e complexos, garantindo que o conteúdo se comporte de maneira responsiva e precisa conforme as coordenadas definidas pelos eixos de linha e coluna. Em última análise, essas propriedades regulam o comportamento microscópico da interface, permitindo um controle individualizado sobre a área de influência de cada componente. Dentre as principais funcionalidades destinadas à gestão desses itens, destacam-se: (1) *grid-column*, (2) *grid-area*, (3) *grid-row*, (4) *justify-self*, e (5) *align-self*. Todos estes são descritos a seguir.

2.6.4.1 Propriedade *grid-column*

A propriedade *grid-column* é responsável por determinar a ocupação horizontal de um item dentro da malha. Em vez de uma localização pontual, ela mapeia o intervalo entre as linhas de grade (*grid lines*) de origem e término, definindo a extensão do elemento sobre o eixo das colunas. Conforme as especificações da [Mozilla \(2025b\)](#), essa declaração permite o uso de índices numéricos ou da cláusula *span*, viabilizando a transposição de múltiplas divisões de forma programática.

- **Definição por Índices de Linha:** Esta metodologia utiliza as linhas divisórias como referência para delimitar a área horizontal. Como observado na [Figura 2.23](#), o Item B é configurado com o valor *2 / 4*. Na prática, o sistema posiciona o componente no espaço compreendido entre a segunda e a quarta linha vertical, resultando na abrangência contígua das colunas 2 e 3.

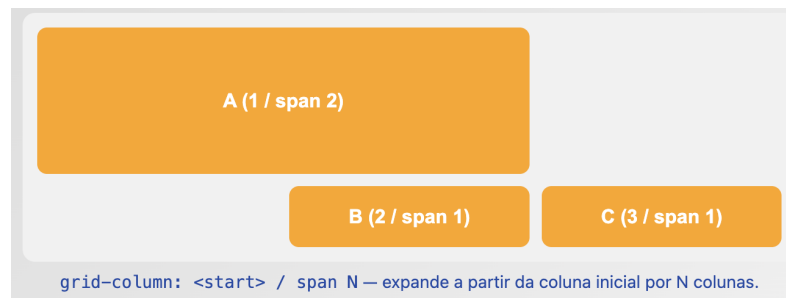
Figura 2.23 – Posicionamento horizontal baseado em índices de *grid lines*.



Fonte: própria autora.

- **Expansão Quantitativa (*span*):** A cláusula *span* substitui a indicação da linha de término por um parâmetro de volume. Na Figura 2.24, o Item A (*1 / span 2*) fixa-se na linha inicial e ocupa duas unidades estruturais. Já o Item B (*2 / span 1*) inicia-se na linha 2 e limita sua presença a uma única coluna. Essa abordagem é ideal para o design responsivo, pois foca na largura desejada para o item e não apenas em coordenadas fixas.

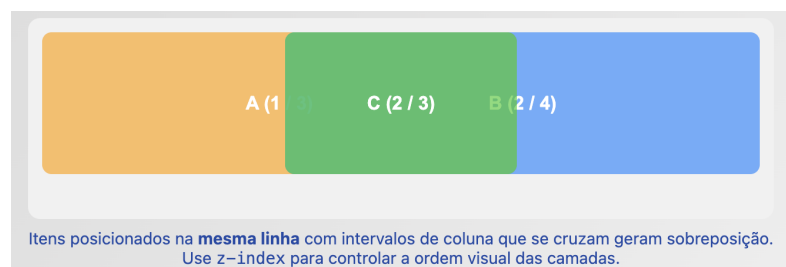
Figura 2.24 – Uso da cláusula *span* para expansão de colunas.



Fonte: própria autora.

- **Gestão de Sobreposição e Camadas:** O sistema de coordenadas permite que diferentes elementos compartilhem o mesmo intervalo de colunas. Como demonstrado na Figura 2.25, ao sobrepor itens, a prioridade de visualização é definida pela ordem do código ou pela propriedade *z-index*. Esse recurso é essencial para a criação de interfaces com profundidade visual e elementos que precisam interagir sobrepostos à malha principal.

Figura 2.25 – Sobreposição de itens e controle de hierarquia visual.



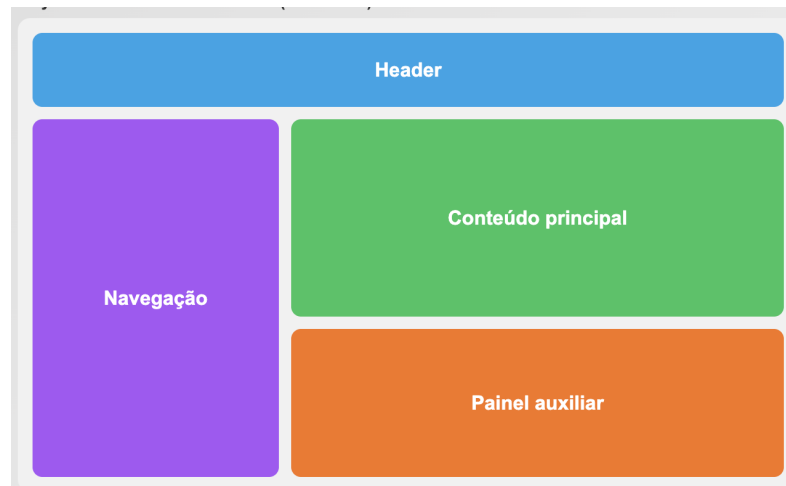
Fonte: própria autora.

2.6.4.2 Propriedade *grid-area*

A propriedade *grid-area* permite vincular um item específico a uma região nomeada da malha, estabelecida previamente por meio da diretriz *grid-template-areas*. De acordo com as especificações da Mozilla (2025b), essa abordagem promove uma organização semântica do código, abstraindo o uso de índices numéricos e substituindo-os por identificadores textuais que refletem as funções lógicas do *layout*, como cabeçalhos (*header*), áreas principais (*main*) ou rodapés (*footer*).

Conforme ilustrado na [Figura 2.26](#), o mapeamento bidimensional ocorre de forma declarativa: ao atribuir o nome da área ao item, o elemento é posicionado automaticamente nas coordenadas correspondentes.

Figura 2.26 – Mapeamento bidimensional: definição de região espacial via *grid-area*.



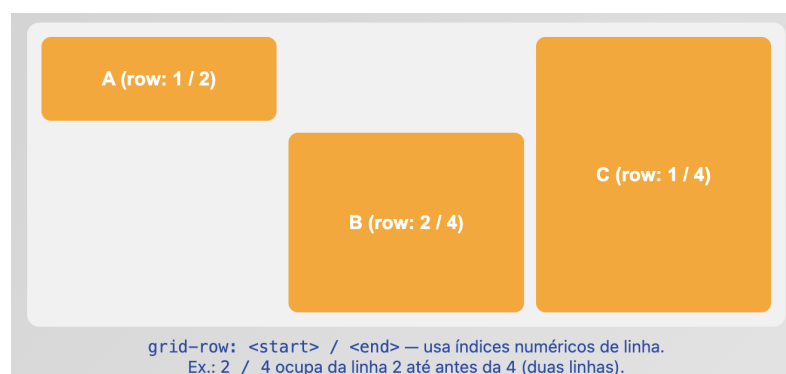
Fonte: própria autora.

2.6.4.3 Propriedade *grid-row*

A propriedade *grid-row* é responsável por mapear o posicionamento vertical de um item, definindo o intervalo entre as linhas de grade (*grid lines*) de origem e término. As diretrizes visuais na imagem fornecida ilustram duas sintaxes principais para controlar essa abrangência:

- *Definição por Índices de Linha (start / end)*: Esta abordagem utiliza índices numéricos para delimitar o início e o fim do item no eixo vertical. Conforme ilustrado na [Figura 2.27](#), o exemplo *B (row: 2/4)* posiciona o item para começar na linha horizontal 2 e terminar na linha 4. Esse comando faz com que o elemento ocupe o espaço compreendido entre essas duas linhas (duas linhas de grade), ilustrando um método preciso de delimitação espacial.

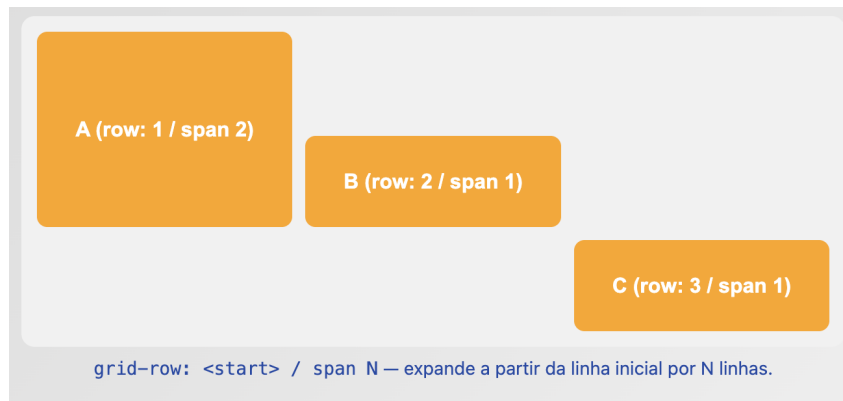
Figura 2.27 – Posicionamento vertical baseado em índices de *grid lines* (row: 2/4).



Fonte: própria autora.

- *Expansão por Quantidade (start / span N)*: A cláusula *span* oferece uma alternativa mais flexível, onde é possível especificar a linha inicial e a quantidade de faixas (*tracks*) que o item deve ocupar. A Figura 2.28 demonstra essa funcionalidade: o exemplo A (*row: 1 / span 2*) define que o item inicia na linha 1 e expande-se por duas linhas adjacentes. Similarmente, B (*row: 2 / span 1*) ocupa apenas uma linha a partir da linha 2, evidenciando a versatilidade dessa sintaxe.

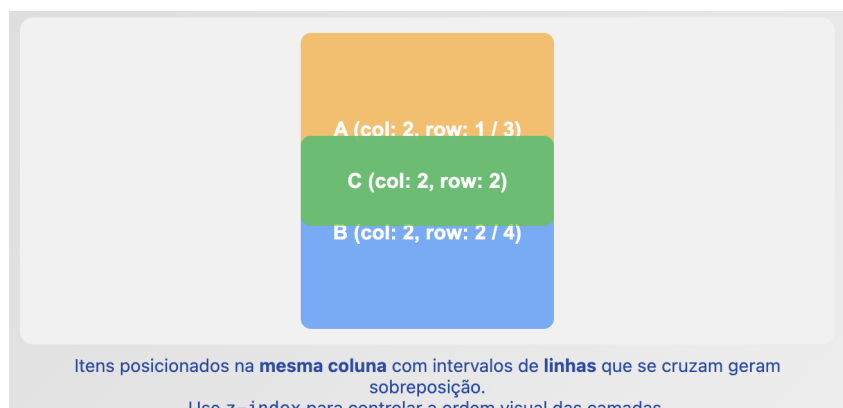
Figura 2.28 – Uso da cláusula *span* para expansão vertical de linhas (*1/span 2*).



Fonte: própria autora.

- *Gestão de Sobreposição Vertical*: A flexibilidade do sistema permite que múltiplos elementos ocupem o mesmo intervalo de linhas. É demonstrado abaixo visualmente que "Itens posicionados na mesma coluna com intervalos de linhas que se cruzam geram sobreposição". Nesses casos, a hierarquia visual é gerenciada pela ordem de declaração no código ou pela propriedade *z-index*, que define a profundidade das camadas.

Figura 2.29 – Sobreposição de itens em eixos cruzados e controle de hierarquia com *z-index*.



Fonte: própria autora.

2.6.4.4 Propriedade *justify-self*

A propriedade *justify-self* regula o comportamento de alinhamento horizontal de um item específico no eixo inline, permitindo que o elemento ignore a diretriz global estabelecida pelo

contêiner via *justify-items*. Esse recurso confere autonomia ao componente para se posicionar em relação às fronteiras laterais de sua célula. Conforme detalhado pela [Mozilla \(2024\)](#), as variações de alinhamento observadas na [Figura 2.30](#) operam sob as seguintes lógicas:

- **start**: Desloca o item para a extremidade esquerda da célula. Com essa escolha, o elemento fica encostado no início da sua área, deixando todo o espaço livre acumulado do lado direito.
- **end**: Posiciona o componente no lado direito da célula. O objeto é movido para o final do espaço disponível, fazendo com que o vazio ou a área restante se concentre no lado esquerdo.
- **center**: Mantém o elemento centralizado horizontalmente. O sistema divide o espaço que sobra de maneira igual entre os dois lados, garantindo que o item fique equilibrado no meio da célula.
- **stretch**: Estica o componente para que ele ocupe a largura total disponível. Este é o comportamento padrão, no qual o item cresce até tocar as bordas laterais, eliminando qualquer espaço vazio dentro da célula.

Figura 2.30 – Impacto visual das opções de alinhamento horizontal individual no item.



Fonte: própria autora.

2.6.4.5 Propriedade *align-self*

O recurso *align-self* possibilita que um item individual adote uma lógica de posicionamento vertical distinta daquela aplicada ao restante da grade. Em vez de seguir o padrão global do contêiner, o elemento ganha liberdade para transitar entre o topo e a base do seu espaço reservado. Segundo as orientações da [Mozilla \(2025b\)](#), as variações demonstradas na [Figura 2.31](#) revelam como o item se distribui no eixo vertical:

- **start**: Fixa o elemento na parte superior da célula. Com isso, o item fica encostado no topo, enquanto todo o espaço restante é deslocado para a porção inferior da área.

- *end*: Move o componente para a parte inferior da célula. O objeto ocupa a base do espaço disponível, fazendo com que o vazio ou a área livre fique concentrada na parte superior.
- *center*: Esta configuração garante a centralização vertical do conteúdo. O espaço vago é distribuído em partes iguais nas extremidades superior e inferior, resultando em um item posicionado à mesma distância em relação ao teto e à base da célula.
- *stretch*: Faz com que o componente se prolongue para atingir a altura máxima disponível. Por ser o comportamento padrão, o item se expande até preencher o vão entre as bordas superior e inferior, ocupando a célula por completo (requer *height: auto*).

Figura 2.31 – Diferentes modos de distribuição vertical individual de um item na grade.



Fonte: própria autora.

2.7 React: biblioteca da camada de visualização

Segundo Kumar (2024), o surgimento do React está associado a um problema recorrente no desenvolvimento de interfaces: aplicações construídas sobre arquiteturas *Model-View-Controller* do lado do cliente, com *binding* bidirecional de dados, isto é, um mecanismo em que qualquer alteração na interface atualiza automaticamente os dados subjacentes e vice-versa, tendiam a se tornar frágeis, fortemente interligadas e difíceis de manter à medida que cresciam. Para lidar com essa limitação, a biblioteca organiza a interface em torno de um conjunto restrito de conceitos centrais, entre os quais o autor destaca os componentes, o *Virtual DOM*, o *JSX*, as propriedades (*props*) e o estado.

Nesse contexto, Banks e Porcello (2020) apresentam o *React* como uma biblioteca *JavaScript* destinada à construção de interfaces de usuário. No capítulo inicial da obra, os autores destacam que, quando lançado, o *React* foi descrito como “the V in MVC”, isto é, como a camada de visualização do padrão *Model-View-Controller*. Tal caracterização evidencia que o *React* não se configura como um *framework* completo, mas como uma biblioteca especializada na

construção da interface, podendo ser combinada com outras soluções para compor aplicações mais abrangentes.

Segundo [Banks e Porcello \(2020\)](#), a proposta central do *React* consiste em permitir a construção de interfaces dinâmicas por meio de um modelo declarativo. Nesse paradigma, o desenvolvedor descreve o que deve ser exibido na interface, enquanto a biblioteca se encarrega de atualizar a representação visual sempre que os dados se modificam. Diferentemente da abordagem imperativa, na qual cada etapa da manipulação do **DOM** precisa ser explicitamente programada, o modelo declarativo prioriza a descrição do resultado desejado, abstraindo os detalhes operacionais da implementação.

Conforme explicam [Banks e Porcello \(2020\)](#), ao definir componentes que retornam estruturas em *JSX*, uma extensão de sintaxe que permite descrever a interface utilizando uma notação semelhante a HTML dentro do próprio código *JavaScript*, o desenvolvedor declara a interface esperada para determinado estado da aplicação. Essa tradução entre a descrição declarativa e o **DOM** real do navegador ocorre por meio do *Virtual DOM*, uma representação da interface mantida em memória e utilizada para calcular o conjunto mínimo de alterações necessárias antes de atualizar o documento efetivamente exibido ([Kumar, 2024](#)). Essa abstração reduz a complexidade associada à manipulação manual da árvore de elementos do navegador e favorece maior previsibilidade no comportamento da aplicação.

[Banks e Porcello \(2020\)](#) também destacam que o funcionamento do *React* está associado ao fluxo unidirecional de dados. Nesse modelo, as informações percorrem a árvore de componentes por meio das propriedades (*props*), responsáveis por transportar dados de um componente pai para seus componentes filhos, e alterações no estado (*state*), que representa informações internas de um componente capazes de se modificar ao longo do tempo, provocam re-renderizações que refletem automaticamente as mudanças na interface. Assim, o estado atua como a referência central da aplicação, garantindo que a interface seja sempre determinada pelos dados vigentes.

Dessa forma, conforme sintetizam [Banks e Porcello \(2020\)](#), trabalhar com *React* implica estruturar a aplicação de maneira declarativa, orientada a dados e fundamentada na atualização automática da interface a partir de mudanças de estado. Esse modelo reduz a complexidade acidental do desenvolvimento, favorece clareza estrutural e sustenta a construção de interfaces interativas modernas.

2.7.1 Arquitetura baseada em componentes

A partir dos conceitos apresentados na seção anterior, a arquitetura baseada em componentes define a forma como os elementos de uma aplicação *React* são organizados estruturalmente. Conforme descreve a documentação oficial do *React* [React \(2025\)](#), o processo de implementação de uma interface inicia-se pela decomposição do *design* em componentes independentes, que posteriormente são organizados em uma hierarquia. Componentes que aparecem visualmente con-

tidos em outros tornam-se seus filhos na árvore resultante. Essa estrutura possui um componente raiz, do qual derivam componentes filhos que, por sua vez, podem conter outros componentes, formando uma organização hierárquica e aninhada.

Quanto à natureza dos componentes que compõem essa árvore, [Wieruch \(2025\)](#) explica que, com a adoção dos *Hooks*, a distinção entre componentes com e sem estado deixou de estar associada à sua forma de declaração, *class* ou *function*, e passou a depender exclusivamente do uso ou não de *hooks* como o *useState*. Dessa forma, um componente funcional pode tanto se limitar a receber dados e convertê-los em elementos visuais quanto armazenar e atualizar informações internas ao longo do tempo, bastando incorporar os *hooks* apropriados. O autor dedica atenção especial à relação entre componentes pai e filho, demonstrando como componentes de níveis superiores podem encapsular partes significativas da aplicação e coordenar o comportamento dos elementos subordinados.

A arquitetura baseada em componentes está diretamente associada ao conceito de composição. Em vez de utilizar estruturas complexas de herança, o *React* privilegia a combinação de componentes menores e especializados para a construção de interfaces mais elaboradas. A documentação oficial da biblioteca [React \(2025\)](#) exemplifica esse padrão por meio da propriedade especial *children*, que permite a um componente receber e renderizar qualquer conteúdo *JSX* aninhado sem a necessidade de conhecer previamente sua estrutura interna. Dessa forma, componentes podem funcionar como contêineres genéricos que acomodam diferentes conteúdos, favorecendo a reutilização de elementos como cabeçalhos, menus, painéis e rodapés.

Outro aspecto central dessa arquitetura é o encapsulamento dos componentes e o baixo acoplamento entre suas responsabilidades. Conforme destacado pela documentação oficial do *React* [React \(2025\)](#), as *props* permitem que componentes pais transmitam informações aos seus filhos sem que estes precisem conhecer a origem dos dados recebidos. Da mesma forma, alterações na lógica interna de um componente filho podem ser realizadas sem impactar seus componentes ancestrais, desde que sua interface de comunicação seja preservada. Esse desacoplamento contribui para a escalabilidade da aplicação e facilita a manutenção e reutilização dos componentes em diferentes contextos.

Do ponto de vista da flexibilidade estrutural, [Wieruch \(2025\)](#) observa que a arquitetura do *React* pode ser complementada por padrões de gerenciamento de estado, à medida que a aplicação cresce em escala e complexidade. Essa característica evidencia que a árvore de componentes constitui uma estrutura extensível, capaz de acomodar novas camadas arquiteturais sem comprometer sua organização fundamental, aspecto retomado com maior profundidade na seção seguinte.

Por fim, a documentação oficial [React \(2025\)](#) destaca que as *props* são tratadas como valores imutáveis durante cada renderização. Consequentemente, quando uma interação do usuário exige alterações na interface, o componente deve solicitar a atualização dos dados ao seu componente pai, geralmente por meio de funções recebidas como propriedades. Essa restrição

contribui para a previsibilidade do fluxo de dados e reforça a integração entre a arquitetura baseada em componentes e o paradigma declarativo que caracteriza o funcionamento do *React*.

2.7.2 Gerenciamento de estado em aplicações *React*: *Context API* e *useReducer*

Segundo Boduch e Derks (2020), o gerenciamento de estado constitui um elemento central na construção de aplicações com *React*, pois é o *state* que confere dinamismo à interface. O estado representa dados passíveis de alteração ao longo do tempo e, sempre que sofre modificações, provoca a atualização automática da interface. Nesse modelo, a interface é determinada pelos dados vigentes, e as mudanças no *state* propagam novas propriedades pela árvore de componentes, resultando na re-renderização das partes afetadas.

À medida que a aplicação cresce e os componentes passam a se aninhar em níveis mais profundos, Boduch e Derks (2020) observam que se torna necessário utilizar componentes intermediários apenas para repassar dados adiante, mesmo quando esses componentes não os utilizam diretamente. Segundo Ayyarrappan (2021), esse fenômeno é conhecido como *prop drilling* e tende a comprometer a legibilidade e a manutenção do código à medida que a aplicação se torna mais complexa, constituindo um dos principais fatores que motivam a adoção de mecanismos específicos para compartilhamento de estado.

Para dados que precisam estar acessíveis em diferentes partes da aplicação, Boduch e Derks (2020) apresentam o *Context* como uma alternativa nativa a essa limitação. Esse mecanismo é composto por dois elementos principais: o *Provider*, responsável por disponibilizar os dados para os componentes descendentes, e o *Consumer*, responsável por acessar essas informações dentro do contexto estabelecido. Tecnicamente, o contexto é criado por meio da função *createContext*, que fornece os componentes necessários para compartilhamento de dados ao longo da árvore. Conforme destacam os autores, essa abordagem reduz a necessidade de repasse manual de propriedades entre múltiplos níveis hierárquicos, embora não substitua os mecanismos responsáveis pelo gerenciamento das mudanças de estado em si.

No que se refere à atualização de estados mais complexos, Boduch e Derks (2020) ressaltam que o uso exclusivo do *useState* pode tornar-se difícil de administrar quando diferentes partes do estado apresentam dependências entre si. Nesses casos, múltiplas funções de atualização precisam ser coordenadas de forma consistente para garantir o comportamento esperado da aplicação. Como alternativa, os autores apresentam o *useReducer*, um *hook*, isto é, uma função disponibilizada pelo *React* que permite a componentes funcionais utilizar recursos como gerenciamento de estado sem a necessidade de recorrer à sintaxe baseada em classes. O *useReducer* centraliza as alterações de estado por meio de uma única função *dispatch*, responsável por encaminhar ações de atualização para uma função redutora (*reducer*), concentrando em um único ponto a lógica de transição do estado da aplicação.

Uma função *reducer*, segundo Boduch e Derks (2020), recebe o estado atual da aplicação e uma ação que descreve a modificação desejada, retornando um novo estado calculado a partir dessas informações. Essa abordagem concentra as regras de transição em um único local, tornando explícita a lógica de atualização e favorecendo maior previsibilidade, organização e manutenção do código.

Dessa forma, o gerenciamento de estado em aplicações *React* envolve a adoção de mecanismos adequados ao nível de complexidade do sistema. O *useState* atende satisfatoriamente a cenários simples e localizados, enquanto o *useReducer* contribui para a organização de transições mais elaboradas, especialmente quando diferentes partes do estado estão relacionadas. Complementarmente, a *Context API* possibilita o compartilhamento eficiente de dados entre componentes distribuídos pela árvore da aplicação, eliminando a necessidade de repasse manual de propriedades em cada nível hierárquico. Em todos esses casos, preserva-se o fluxo unidirecional de dados e a coerência com o paradigma declarativo que caracteriza o *React*.

3 Trabalhos Relacionados

Este capítulo busca analisar e evidenciar a relevância das plataformas Web interativas como mediadoras do processo educacional, ressaltando que a interatividade, a organização consistente das interfaces e a presença de *feedback* em tempo real configuram fatores centrais para o engajamento e a participação ativa dos usuários. Desse modo, a efetividade dessas soluções não se restringe à inserção de elementos lúdicos, mas resulta da integração coerente entre tais recursos e uma experiência digital estruturada e orientada à ação do estudante.

Nesse cenário, [Moreira \(2022\)](#) desenvolveu a plataforma CS Arcade como uma proposta gamificada voltada à formação de hábitos de leitura entre estudantes de Ciência da Computação. O projeto foi fundamentado em referenciais clássicos da gamificação, como o *Framework 6D*, o *Gamification Design Framework* e o modelo *Octalysis*, articulando metodologias como *Design Participativo*, *Design Sprint* e *Design Baseado em Cenários* no processo de concepção e validação.

Sob o ponto de vista técnico, a aplicação CS Arcade ([Moreira, 2022](#)) foi estruturada segundo a arquitetura *Model–View–Controller (MVC)*, utilizando banco de dados *Amazon RDS*, testes automatizados com *PHPUnit* e prototipação realizada no *Figma*, além da coleta de dados por meio de formulários no *Google Forms*. Os resultados apontaram funcionamento adequado do sistema, ausência de falhas críticas e coerência nas regras de negócio. Ainda assim, o estudo reconhece como limitação a inexistência de acompanhamento a médio e longo prazo que permita comprovar empiricamente a consolidação do hábito de leitura ao longo do tempo.

A análise do CS Arcade, portanto, contribui para situar o presente trabalho no campo das aplicações Web interativas voltadas à educação. Enquanto o CS Arcade direciona-se à promoção de práticas de leitura, o *Tezza Space* consolida-se como uma proposta voltada especificamente ao ensino do *CSS Grid*.

Em uma perspectiva complementar, [Magalhães \(2018\)](#) apresentou o desenvolvimento da aplicação Web educacional *QuizApp*, cujo objetivo consistiu em integrar dispositivos móveis ao ensino formal. O projeto adotou uma abordagem de desenvolvimento incremental, estruturada em ciclos de levantamento de requisitos, implementação e testes em sala de aula. A solução foi construída com tecnologias do ecossistema *Java* e documentada por meio de *Unified Modeling Language (UML)*, sendo avaliada a partir de observação, estatísticas do sistema e questionários baseados na escala de *Likert*.

Os resultados obtidos por [Magalhães \(2018\)](#) indicam uma percepção positiva quanto à usabilidade, à clareza da interface e à contribuição para a aprendizagem, além de um maior dinamismo nas aulas. [Magalhães \(2018\)](#) afirma que os objetivos propostos foram alcançados, embora reconheça limitações, como testes restritos à rede local, problemas pontuais de login e o número reduzido de participantes, fatores que restringem a generalização dos achados.

A partir dessa análise comparativa, observa-se que ambas as propostas utilizam aplicações Web como instrumentos de mediação pedagógica e incorporam o *feedback* imediato como estratégia de apoio à aprendizagem. Contudo, diferentemente do *QuizApp*, voltado ao suporte à avaliação em múltiplas disciplinas, o *Tezza Space* direciona-se especificamente ao ensino do *CSS Grid*, com ênfase na compreensão progressiva do modelo bidimensional de *layout*.

Por outro lado, Filho, Neves e Júnior (2023) descreve o desenvolvimento de um jogo digital interativo voltado ao ensino de lógica proposicional na Web, estruturado no formato de quebra-cabeças. A proposta buscou possibilitar ao usuário a interpretação de enunciados, a construção de sentenças com conectivos lógicos e a geração de tabelas-verdade, articulando raciocínio formal e aplicação prática. O projeto envolveu etapas de prototipação, implementação e avaliação, sendo desenvolvido com tecnologias Web modernas, como *HTML*, *CSS*, *JavaScript* e *layout* no *front-end*, além de *Node.js* com arquitetura *Representational State Transfer* (REST) no *back-end* e banco de dados *MongoDB*.

A avaliação do sistema baseou-se nos critérios de eficiência, eficácia e satisfação da norma *ISO 9241-11*, contando com a participação de 24 estudantes do ensino médio técnico. Os resultados evidenciaram elevada aceitação quanto à usabilidade, clareza do *feedback* e contribuição para a aprendizagem, sem registro de falhas críticas nos testes técnicos. O estudo conclui que os objetivos foram alcançados, embora aponte possibilidades de expansão de funcionalidades.

A partir desse panorama, percebe-se que, embora compartilhem a proposta de utilizar ambientes digitais interativos como mediadores do aprendizado, as iniciativas analisadas diferenciam-se quanto aos conteúdos abordados e às competências mobilizadas. Enquanto o jogo de lógica proposicional concentra-se na abstração e formalização de sentenças lógicas, o *Tezza Space* afirma-se como uma proposta direcionada ao ensino do *CSS Grid*, enfatizando a organização declarativa de *layouts* no contexto do desenvolvimento Web.

De modo geral, os estudos analisados indicam que a utilização de plataformas Web interativas no contexto educacional ultrapassa a simples incorporação de estratégias de gamificação. Observa-se que a combinação entre interfaces estruturadas de forma consistente, mecanismos de *feedback* imediato e elementos lúdicos favorece a construção de experiências de aprendizagem mais ativas e participativas. Nesse sentido, tais fatores mostram-se fundamentais para sustentar o engajamento dos estudantes/usuários e favorecer a assimilação de conhecimentos em diferentes áreas.

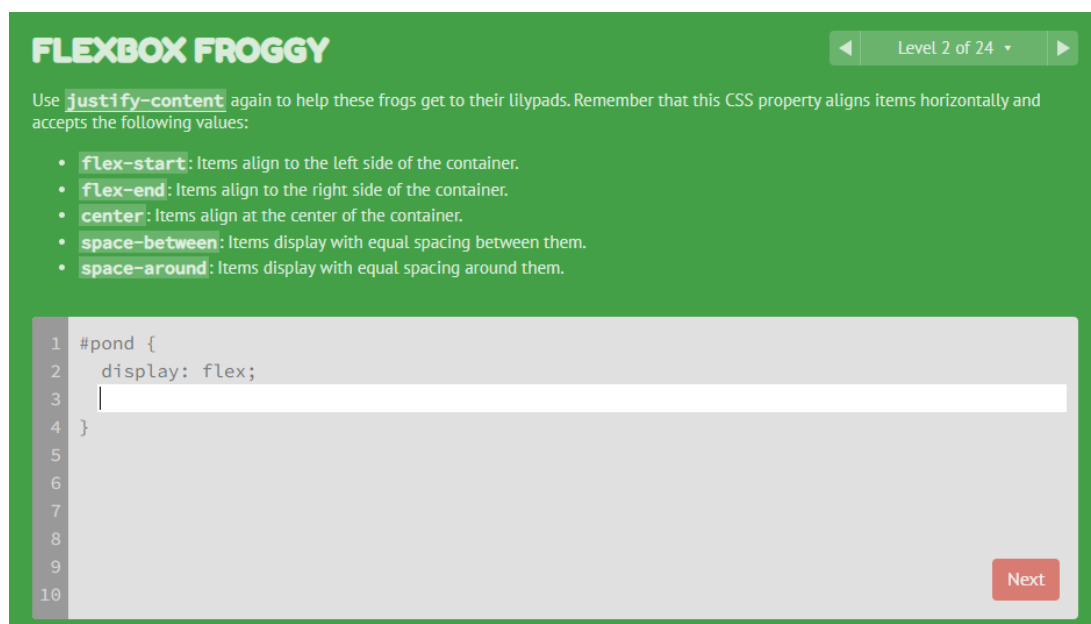
Apesar de os trabalhos citados terem um cunho pedagógico no estudo de programação, este trabalho tem duas aplicações com uma proposta semelhante: **Flexbox Froggy** e **CSS Grid Garden**.

3.1 Plataforma Interativa *Flexbox Froggy* como Abordagem Didática para o Ensino de *Flexbox*

A plataforma *Flexbox Froggy* (Flexbox Froggy, 2025) consiste em um ambiente interativo estruturado em formato de jogo, cujo objetivo é introduzir os conceitos fundamentais do modelo de *layout Flexbox*. A proposta adota uma metáfora visual simples, baseada em um lago no qual os sapos devem ser posicionados sobre vitórias-régias. Nessa representação, os sapos correspondem aos itens flexíveis, enquanto as vitórias-régias indicam as posições que devem ser alcançadas ao longo do desafio.

Conforme apresentado na Figura 3.1, a interface organiza-se em dois espaços principais: um painel de instruções textuais, responsável por descrever o objetivo da fase, e um editor de código no qual o usuário deve escrever regras em *CSS*. Essa disposição evidencia a articulação entre orientação conceitual e prática de codificação, conduzindo o usuário à aplicação direta das propriedades do *Flexbox* no contêiner flexível.

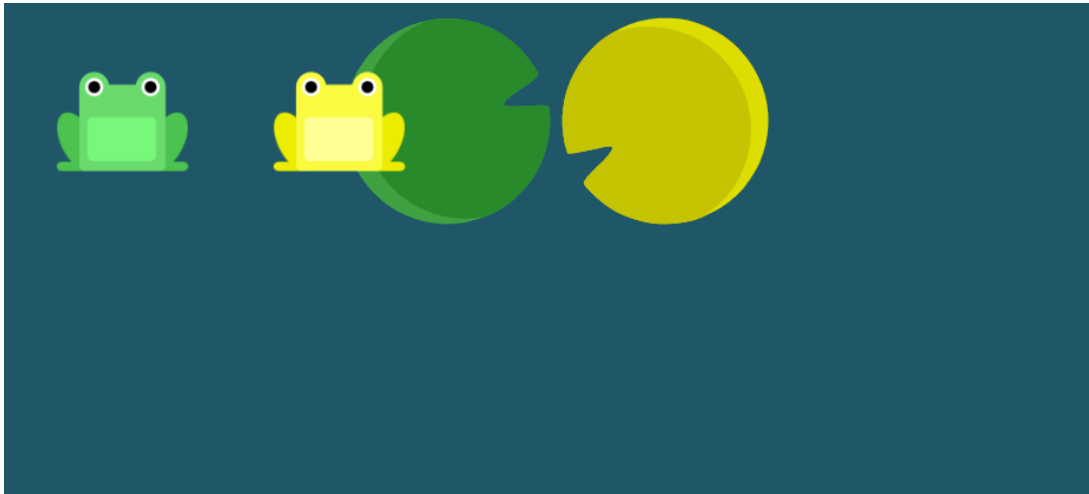
Figura 3.1 – Interface do jogo *Flexbox Froggy*, destacando o painel de instruções e o editor de código *CSS*.



Fonte: Plataforma Flexbox Froggy (2025).

A dinâmica do jogo fundamenta-se na atualização imediata do resultado visual. À medida que propriedades como *justify-content* são inseridas no editor, o posicionamento dos elementos é modificado em tempo real. A Figura 3.2 ilustra esse comportamento ao evidenciar o momento em que os sapos precisam ser alinhados corretamente às vitórias-régias por meio do controle do eixo principal. Esse retorno visual instantâneo favorece a aprendizagem por experimentação, permitindo que o usuário formule hipóteses, observe os efeitos produzidos e refine o código até alcançar a solução esperada.

Figura 3.2 – Área visual do jogo *Flexbox Froggy*, na qual os sapos devem ser alinhados às vitórias-régias por meio da aplicação de propriedades do *Flexbox*.



Fonte: [Flexbox Froggy \(2025\)](#).

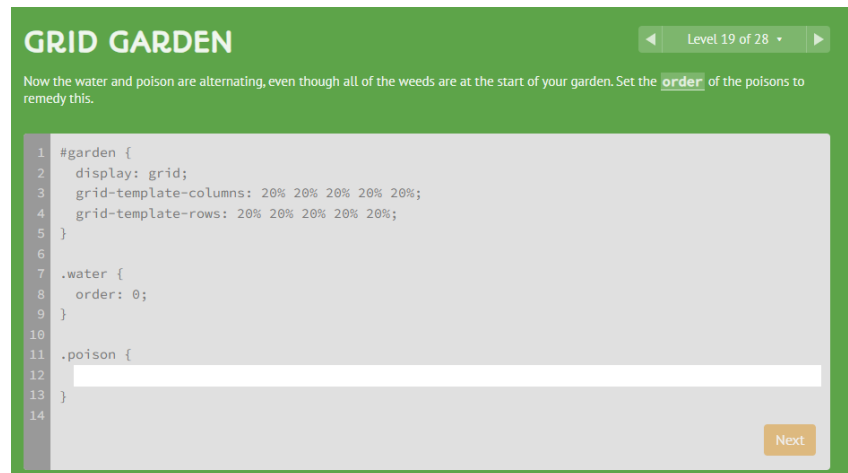
Do ponto de vista pedagógico, o *Flexbox Froggy* mostra-se eficaz como ferramenta introdutória para o ensino do modelo unidimensional do *Flexbox*. A clareza da interface e a associação direta entre comando e resultado contribuem para a construção de um modelo mental consistente acerca do alinhamento e da distribuição de elementos ao longo de um único eixo.

Enquanto o *Flexbox Froggy* é direcionado ao modelo unidimensional do *Flexbox*, concentrando-se na organização e no alinhamento de elementos ao longo de um único eixo por vez, o *Tezza Space* volta-se ao ensino do *CSS Grid*, cuja lógica envolve a estrutura bidimensional do *layout*, articulando simultaneamente linhas e colunas.

3.2 Plataforma Interativa *CSS Grid Garden* como Referência Didática

A plataforma *CSS Grid Garden* ([CSS Grid Garden, 2025](#)) consiste em um ambiente interativo de aprendizagem voltado para a compreensão do modelo bidimensional de *layout* do *CSS Grid*, estruturado em 28 fases. A ferramenta adota a metáfora visual de um jardim, no qual elementos como plantas, áreas irrigadas e ervas representam regiões do *layout* a serem organizadas. Cada fase propõe um problema de distribuição espacial que exige do usuário a definição explícita de linhas, colunas e áreas da *grid*. Conforme ilustrado na [Figura 3.3](#), a interface enfatiza a configuração estrutural da *grid* por meio do código *CSS*, evidenciando a relação direta entre a declaração das propriedades e a organização do espaço disponível.

Figura 3.3 – Interface do jogo *CSS Grid Garden*, destacando o painel de instruções e o editor de código *CSS* para configuração do *grid*.



Fonte: *CSS Grid Garden* (2025).

A dinâmica da plataforma concentra-se na estruturação do espaço bidimensional. Ao aplicar propriedades como *grid-template-rows* e *grid-template-columns*, o usuário constrói explicitamente a *grid* que organiza os elementos do jardim, compreendendo como linhas e colunas se combinam para formar áreas específicas. Essa organização pode ser observada na Figura 3.4, que apresenta o tabuleiro visual subdividido em células regulares, reforçando a compreensão do *CSS Grid* como um modelo orientado à definição prévia da estrutura do *layout*.

Figura 3.4 – Área visual do jogo *CSS Grid Garden*, evidenciando a organização bidimensional do tabuleiro em linhas e colunas.



Fonte: *CSS Grid Garden* (2025).

A principal diferença entre o *CSS Grid Garden* e o *Flexbox Froggy* reside no modelo de

layout abordado. Enquanto o *Flexbox Froggy* é direcionado ao ensino do modelo unidimensional do *Flexbox*, centrado na distribuição de elementos ao longo de um único eixo por vez, o *CSS Grid Garden* dedica-se ao modelo bidimensional do *CSS Grid*, no qual linhas e colunas são articuladas simultaneamente.

Em relação ao *Tezza Space*, embora ambas as propostas compartilhem o mesmo objeto de ensino, a diferença manifesta-se na ambientação temática, na abrangência conceitual e na estrutura arquitetural da aplicação. O *CSS Grid Garden* adota a metáfora de um jardim como elemento central da experiência, mantendo um tabuleiro estruturalmente estático ao longo das fases. No *Tezza Space*, por sua vez, a ambientação é espacial, organizada em torno de um tabuleiro orbital composto por módulos com identidade visual própria, alinhados a uma narrativa temática distinta.

Do ponto de vista conceitual, o *CSS Grid Garden* concentra-se predominantemente em propriedades relacionadas à definição estrutural da *grid* e ao posicionamento de seus elementos filhos. O *Tezza Space* busca integrar tanto propriedades aplicáveis aos itens quanto propriedades diretamente associadas ao contêiner, articulando esses dois níveis de configuração de maneira complementar ao longo das fases.

Além disso, o *Tezza Space* é fundamentado na componentização do sistema. O tabuleiro orbital não permanece fixo, mas assume diferentes configurações de linhas e colunas conforme os objetivos de cada fase, permitindo variações estruturais planejadas para apoiar a progressão didática. Diferentemente do jardim, cujo tabuleiro mantém-se essencialmente estático, o *Tezza Space* utiliza a própria estrutura da *grid* como elemento variável de aprendizagem, explorando mudanças na configuração espacial como parte integrante da experiência pedagógica.

4 CSS Tezza Space

Neste capítulo, será apresentada uma visão geral do sistema *Tezza Space*, descrevendo sua proposta, seus principais componentes e a lógica de funcionamento que sustenta a aplicação. A Seção 4.1 expõe a estrutura geral do sistema, destacando os elementos que compõem a interface e suas respectivas funções no processo de interação do usuário. A Seção 4.2, por sua vez, discute a organização das vinte fases em dois conjuntos didáticos distintos, responsáveis por orientar a progressão entre as propriedades dos itens filhos e as propriedades do contêiner, bem como o modelo de fase utilizado como referência durante essa construção. Na Seção 4.3, é detalhado o processo de produção dos *assets* visuais da aplicação com apoio de Inteligência Artificial Generativa, incluindo a estrutura do *prompt*-base adotado e a evolução iterativa dos elementos gráficos.

Em seguida, a Seção 4.4 apresenta a modelagem conceitual adotada para organizar fases, regras e propriedades exigidas, enquanto a Seção 4.5 aborda os aspectos arquiteturais da aplicação, com destaque para o gerenciamento centralizado de estado, discutido na Subseção 4.5.1, e para a camada de apresentação, detalhada na Subseção 4.5.2. Na sequência, a Seção 4.6 descreve o fluxo de edição e processamento das regras inseridas pelo usuário, a Seção 4.7 apresenta o conjunto de funções auxiliares que sustentam uma *grid* unificada, e a Seção 4.8 detalha o processo de renderização no tabuleiro orbital. Por fim, a Seção 4.9 descreve os testes automatizados implementados para validar os mecanismos discutidos ao longo do capítulo, incluindo a cobertura de código obtida.

Dessa forma, este capítulo busca apresentar, de maneira integrada, a estrutura interna do sistema e o fluxo completo que conecta a interação textual à materialização visual na interface.

4.1 Visão geral do sistema

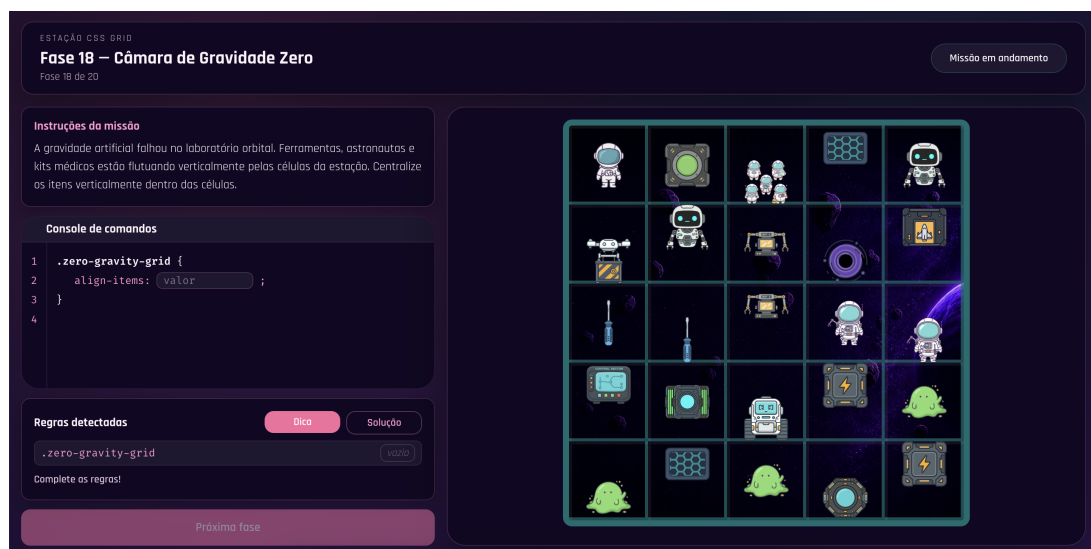
O sistema *Tezza Space* foi desenvolvido como uma aplicação Web interativa voltada ao ensino de *CSS Grid*, partindo da premissa de que o processo de aprendizagem é favorecido quando o usuário pode experimentar, testar e observar diretamente os resultados das suas ações. Nesse contexto, o usuário interage de forma direta com regras de *CSS*, alterando propriedades e acompanhando, em tempo real, as transformações visuais que ocorrem no *layout*, o que contribui para a compreensão da relação entre a sintaxe utilizada e o comportamento visual resultante.

Para sustentar essa proposta, o sistema foi estruturado de modo que cada modificação realizada no editor seja refletida imediatamente na visualização da *grid*, sem a necessidade de qualquer ação adicional por parte do usuário. Esse comportamento estabelece um ciclo contínuo de experimentação, no qual o usuário formula hipóteses, observa os efeitos das alterações realizadas

e, quando necessário, reescreve as regras aplicadas, promovendo um aprendizado progressivo e exploratório.

A interface do *Tezza Space* foi estruturada de modo a integrar orientação, ação e resposta visual. A [Figura 4.1](#) apresenta a interface geral do sistema, evidenciando a composição dos seus componentes na tela. No topo da página, o cabeçalho exibe o nome da estação, o título e o número da fase atual, além de um indicador de *status* que informa se a missão encontra-se em andamento ou concluída. Abaixo, a interface organiza-se em quatro componentes principais: o painel de instruções da missão, o console de comandos, o painel de regras detectadas e o tabuleiro *Tezza Space*. Essa disposição busca conduzir o usuário ao longo do processo de resolução dos desafios, tornando explícita a relação entre o objetivo proposto, a edição das regras e o resultado visual obtido.

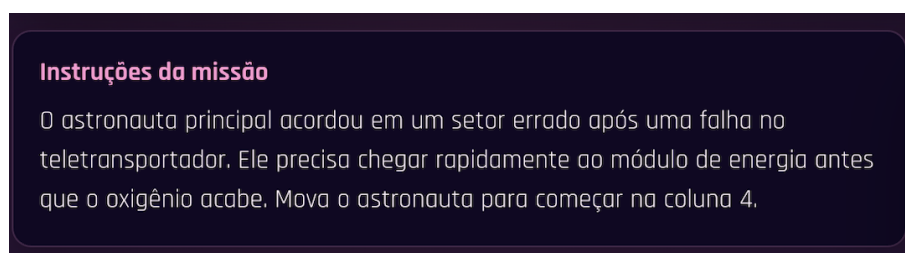
Figura 4.1 – Interface geral do sistema *Tezza Space*.



Fonte: própria autora.

O painel de instruções da missão apresenta, de forma textual, o objetivo específico de cada fase, contextualizando o desafio e delimitando as condições esperadas para sua resolução sem antecipar a solução. A [Figura 4.2](#) ilustra um exemplo desse painel, no qual a tarefa é descrita de maneira objetiva e contextualizada.

Figura 4.2 – Painel de instruções da missão.



Fonte: própria autora.

O console de comandos constitui o principal ponto de interação ativa do usuário com o sistema, no qual são escritas diretamente as regras de **CSS** associadas aos elementos do jogo, simulando um ambiente de codificação semelhante ao utilizado em contextos reais de desenvolvimento *front-end*. A [Figura 4.3\(a\)](#) apresenta o console em seu estado inicial, quando ainda não há regras definidas, enquanto a [Figura 4.3\(b\)](#) ilustra o mesmo componente após a inserção das propriedades de posicionamento.

Figura 4.3 – Comportamento do painel de regras



(a) Console no estado inicial.

(b) Console após a inserção das propriedades.

Fonte: própria autora.

O painel de regras detectadas apresenta, em tempo real, as declarações **CSS** identificadas pelo sistema durante a digitação no console. Para cada seletor reconhecido, é exibido um indicador visual de estado que auxilia o usuário no acompanhamento da construção das regras. Quando um seletor é declarado, mas ainda não possui propriedades associadas, ele é sinalizado como uma regra vazia, conforme ilustrado na [Figura 4.4\(a\)](#). À medida que propriedades válidas são adicionadas, o indicador é atualizado para representar uma regra completa, como mostrado na [Figura 4.4\(b\)](#).

Figura 4.4 – Comportamento do painel de regras



(a) Regras vazias.

(b) Regras completas.

Fonte: própria autora.

Além do acompanhamento das regras, o painel disponibiliza dois recursos de apoio à resolução dos desafios. O botão “Dica” projeta sobre o tabuleiro orbital uma configuração visual de referência abaixo do tabuleiro orbital, indicando graficamente o posicionamento esperado sem revelar os comandos necessários para alcançá-lo; aspecto que será detalhado na seção referente ao tabuleiro *Tezza Space*. O botão “Solução”, por sua vez, exibe diretamente no console, abaixo das linhas já digitadas pelo usuário, conforme exemplificado na [Figura 4.5](#), as propriedades **CSS** que produzem o resultado correto para a fase em execução, permitindo que o estudante consulte a resposta e compreenda a relação entre a declaração e o comportamento visual obtido.

Figura 4.5 – Console de comandos após o acionamento do botão “Solução”.

```

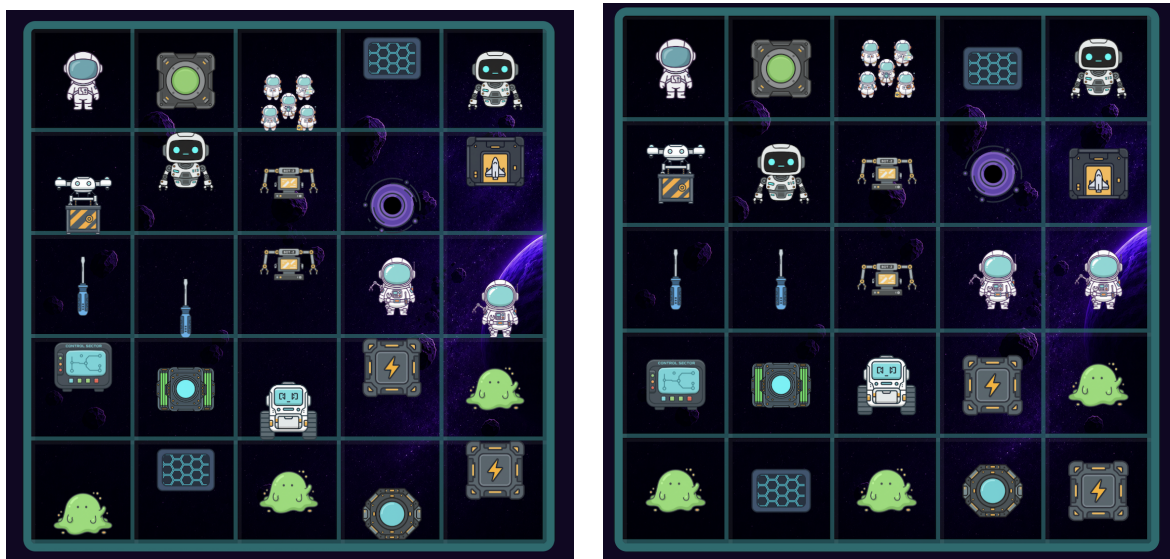
Console de comandos
1 .zero-gravity-grid {
2   align-items:  ;
3   /* digite: center */
4 }

```

Fonte: própria autora.

O tabuleiro orbital, por sua vez, corresponde à representação visual da *grid* configurado, espaço no qual se manifestam graficamente os efeitos das regras definidas no console. A Figura 4.6(a) apresenta o tabuleiro em seu estado padrão, antes da aplicação de qualquer regra de posicionamento, ao passo que a Figura 4.6(b) demonstra o comportamento do ambiente após a execução dos comandos, evidenciando as alterações provocadas pelas propriedades CSS aplicadas.

Figura 4.6 – Comportamento do tabuleiro orbital



(a) Tabuleiro no estado padrão.

(b) Tabuleiro após regras.

Fonte: própria autora.

Quando o usuário aciona o botão “Dica”, o tabuleiro exibe uma sobreposição visual sobre o estado atual dos elementos, indicando graficamente o posicionamento esperado para a conclusão da fase, conforme ilustrado na Figura 4.7. Essa sobreposição não altera o estado interno do sistema nem interfere nas regras definidas pelo usuário no console, funcionando exclusivamente como referência visual para orientar a resolução do desafio sem revelar os comandos necessários para alcançá-lo.

Figura 4.7 – Tabuleiro com a sobreposição visual ativada.



Fonte: própria autora.

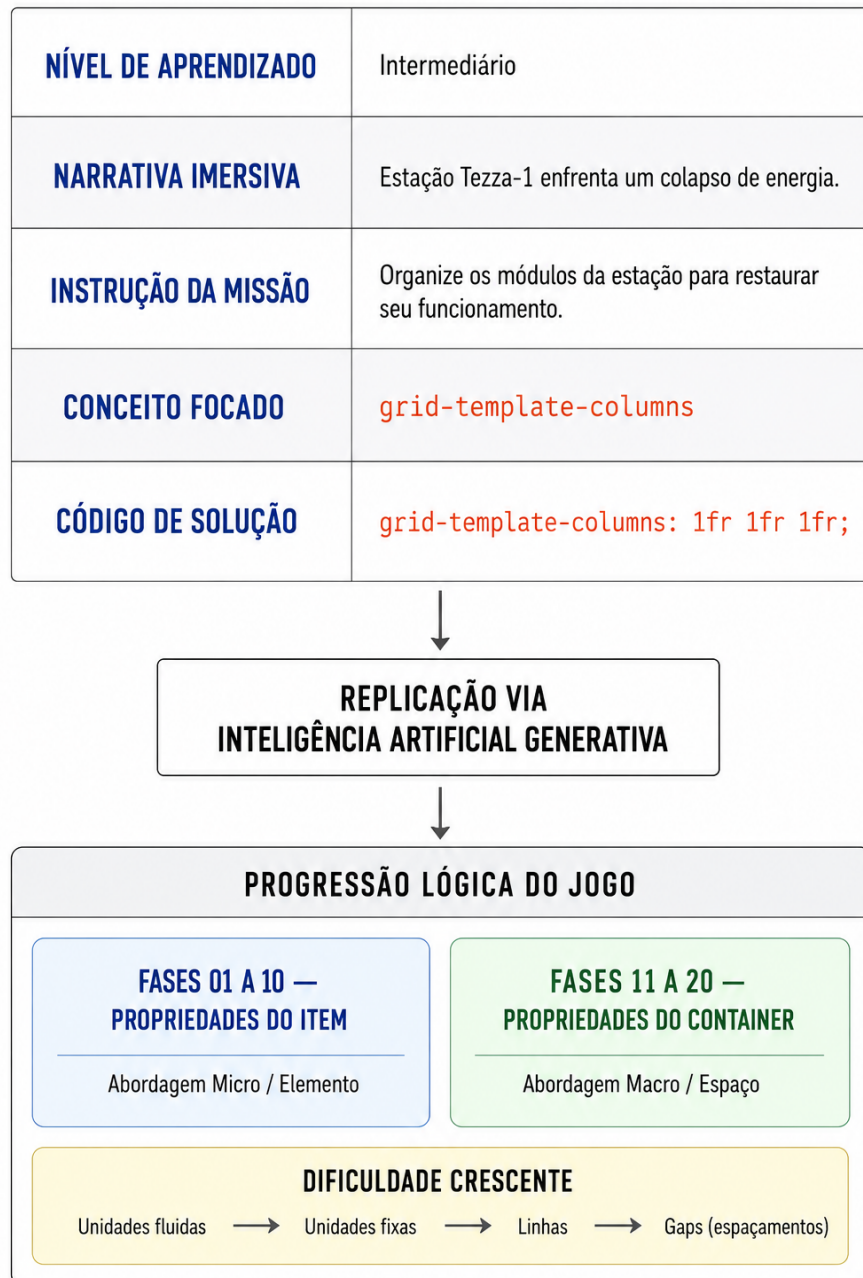
4.2 Estrutura de fases e progressão didática

A definição das vinte fases do *Tezza Space* foi realizada a partir de uma estrutura padronizada concebida para garantir coerência pedagógica, progressão de dificuldade e alinhamento entre os objetivos de aprendizagem e os desafios apresentados ao estudante. Para apoiar esse processo, foi utilizada a Inteligência Artificial Generativa [Chat Generative Pre-trained Transformer \(ChatGPT\)](#), versão *GPT-4*, na elaboração de um modelo de fase capaz de organizar, de forma sistemática, os elementos necessários para a construção de cada missão. A partir de um *prompt* descrevendo os requisitos pedagógicos desejados, incluindo a progressão gradual dos conteúdos, a contextualização narrativa, a definição explícita do conceito abordado e o critério objetivo de validação, a ferramenta gerou uma estrutura base que passou a ser utilizada como referência durante o desenvolvimento de toda a sequência de fases da aplicação.

Conforme apresentado na [Figura 4.8](#), cada fase foi organizada em cinco componentes principais: (1) o nível de aprendizado, que posiciona o usuário dentro da sequência de progressão; (2) a narrativa imersiva, que contextualiza o desafio no cenário da estação *Tezza Space*; (3) a instrução objetiva, que descreve diretamente a ação esperada; (4) o conceito *CSS* associado, definido como foco exclusivo da etapa; e (5) o código de solução previamente definido, que orienta o mecanismo de validação automática do sistema. Dessa forma, cada missão trabalha

apenas uma propriedade por vez, evitando a introdução simultânea de múltiplos conceitos e favorecendo uma construção progressiva do conhecimento.

Figura 4.8 – Plano mental utilizado como referência durante o desenvolvimento da sequência de missões do *Tezza Space*.



Fonte: própria autora.

A organização das fases materializa, na prática, a distinção conceitual entre propriedades do contêiner e propriedades dos itens filhos discutida na [Seção 2.6.2](#). As vinte fases foram divididas em dois conjuntos distintos: o primeiro reúne dez fases dedicadas exclusivamente às propriedades dos *Grid Items* e o segundo agrupa outras dez fases voltadas às propriedades do

Grid Container. Essa separação evita que o usuário precise transitar simultaneamente entre dois níveis de controle distintos antes de consolidar a compreensão de cada um deles isoladamente.

O conjunto voltado aos itens filhos organiza-se segundo complexidade crescente, partindo do controle básico de posição em direção a formas progressivamente mais expressivas de manipulação. As primeiras fases introduzem o posicionamento elementar por meio de `grid-column-start` e `grid-row-start`, conduzindo o usuário a deslocar um único elemento ao longo dos eixos horizontal e vertical. Na sequência, explora-se a delimitação de término com `grid-column-end` e a expansão de elementos por meio da cláusula `span` aplicada a `grid-column` e a `grid-row`, bem como o uso do *shorthand* de linhas para definir simultaneamente início e fim. As fases finais desse conjunto avançam para o alinhamento individual com `justify-self` e `align-self` e culminam no emprego de `grid-area`, primeiro por meio de nome de área e, por último, com as abreviações de linhas, o que evidencia ao usuário a equivalência entre as diferentes sintaxes de posicionamento.

O conjunto dedicado ao contêiner pai segue uma lógica análoga, iniciando pela definição da estrutura da *grid* e avançando rumo às propriedades de distribuição e organização semântica do espaço. As fases iniciais tratam da configuração das trilhas por meio de `grid-template-columns` e `grid-template-rows`, contemplando tanto colunas iguais quanto frações desiguais, e prosseguem com o controle de espaçamento mediante `gap`, `row-gap` e `column-gap`. Na etapa seguinte, aborda-se o alinhamento dos itens em suas células com `justify-items` e `align-items` e a distribuição do conjunto na *grid* com `justify-content`. O conjunto encerra-se com `grid-template-areas`, na qual o usuário nomeia regiões da estação para compor um mapa estrutural completo, reunindo, nessa última fase, os conceitos de definição de trilhas e de organização declarativa do *layout*.

A progressão técnica é sustentada por uma ambientação narrativa coesa, situada a bordo da estação orbital Tezza-1, cuja denominação remete ao sobrenome da autora. Cada fase apresenta uma instrução contextualizada por meio de personagens recorrentes, entre eles o astronauta principal, o robô BOT-Z, os drones de carga e os robôs de transporte, envolvidos em missões de reparo e situações de urgência, como explosões solares, vazamentos criogênicos e o avanço de gosma alienígena. Essa narrativa de urgência confere sentido ao objetivo técnico de cada desafio, conduzindo o usuário a aplicar determinada propriedade do [CSS Grid](#) como resposta a um problema concreto, sem que a ambientação interfira na lógica de aprendizado ou na avaliação da solução.

4.3 Produção dos *assets* visuais com Inteligência Artificial Generativa

O desenvolvimento dos elementos gráficos do *Tezza Space* constituiu uma etapa fundamental para a construção da identidade visual da aplicação. Esses recursos, organizados em

uma biblioteca de *assets*, abrangem personagens, módulos, equipamentos, estruturas e objetos distribuídos ao longo das vinte fases do sistema. A construção dessa coleção exigiu a produção de aproximadamente quarenta ativos gráficos distintos, planejados de modo a representar os elementos manipulados pelo usuário durante as atividades e a manter uma linguagem visual uniforme entre os diferentes tipos de módulos.

Diante da escala de produção necessária e da exigência de coerência estética entre elementos de natureza variada, optou-se pela utilização de Inteligência Artificial Generativa como ferramenta de apoio ao processo de criação visual. A produção da biblioteca foi realizada no *Figma* em conjunto com o modelo *Nano Banana 2.0*. Diferentemente de um fluxo tradicional de ilustração digital, no qual cada elemento é desenhado manualmente desde sua concepção, a abordagem generativa permitiu acelerar a etapa de ideação e explorar rapidamente diferentes possibilidades visuais para um mesmo componente, sem abrir mão do controle sobre o resultado final.

Para garantir uniformidade em toda a coleção, foi desenvolvido um *prompt*-base responsável por estabelecer características visuais comuns a todas as gerações. Conforme apresentado na [Tabela 4.1](#), este *prompt* segue uma estrutura modular composta por sete blocos fixos, sendo apenas o último variável e adaptado conforme o ativo desejado. O bloco de estilo determina a utilização de ilustrações vetoriais bidimensionais planas, compostas por formas geométricas simplificadas e sem gradientes, texturas, sombras, iluminação ou volume. O bloco de traço estabelece contornos consistentes de espessura média, com detalhes internos mínimos e mesma espessura de traço em todo o elemento. O bloco de cor restringe a composição a estruturas em cinza, azul, laranja e roxo. O bloco de grade define que todos os elementos devem ser centralizados em um bloco quadrado, com margens internas uniformes, alinhados a uma grade interna implícita e sem nenhum elemento cortado nas bordas. O bloco de vista determina perspectiva frontal ortográfica estrita, sem isometria. O bloco de fundo exige cor plana única, sem textura ou gradiente. O bloco de objeto, único variável da estrutura, recebe a descrição específica do ativo a ser gerado.

Tabela 4.1 – Estrutura em blocos do *prompt*-base utilizado na geração de *assets*.

Bloco	Descrição
[estilo]	Ilustração vetorial 2D plana, formas geométricas simplificadas, sem gradientes, texturas, sombras, iluminação ou volume.
[traço]	Contornos consistentes de espessura média com traços levemente marcados; detalhes internos mínimos e usar mesma espessura do traço.
[cor]	Estruturas em cinza, azul, laranja e roxo.
[grade]	Bloco quadrado, margens internas uniformes, centrado e simétrico, alinhado em grade interna implícita, nada cortado nas bordas.
[vista]	Vista frontal ortográfica estrita, sem perspectiva ou isometria.
[fundo]	Fundo de cor plana única, sem textura ou gradiente.
[objeto]	<descrição específica do ativo> — único bloco variável do <i>prompt</i> .

Fonte: própria autora.

A Figura 4.9 apresenta uma linha do tempo representativa do processo de evolução visual adotado na construção da biblioteca, exibindo iterações sucessivas de três componentes do ecossistema de módulos: o alienígena, o painel solar e a gosma, desde as versões inicialmente geradas até os ativos incorporados à aplicação. Essa sequência evidencia de que forma a ferramenta generativa apoiou a exploração de alternativas visuais e o ajuste gradual dos elementos que compõem a identidade gráfica do sistema.

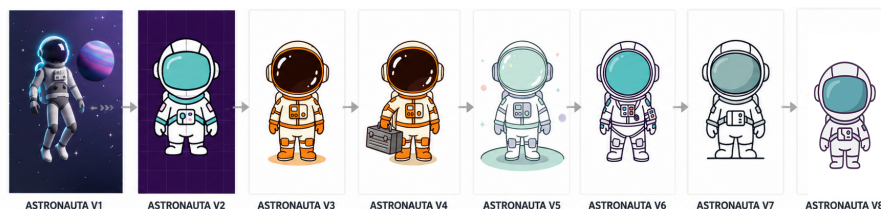
Figura 4.9 – Linha do tempo do processo de refinamento visual dos *assets*: alienígena, painel solar e gosma.



Fonte: própria autora.

Entre todos os *assets* produzidos, o astronauta recebeu atenção especial por representar o personagem central da narrativa e estar presente em diversas fases da aplicação. A Figura 4.10 apresenta a evolução visual desse personagem desde as primeiras gerações até a versão final incorporada à biblioteca. As propostas iniciais apresentavam variações nas proporções, no formato do capacete e no nível de detalhamento. Com a consolidação do *prompt*-base e dos critérios visuais estabelecidos para a biblioteca, as versões posteriores passaram a apresentar maior uniformidade, resultando no personagem utilizado na versão final da aplicação.

Figura 4.10 – Evolução visual do astronauta *asset* astronauta.



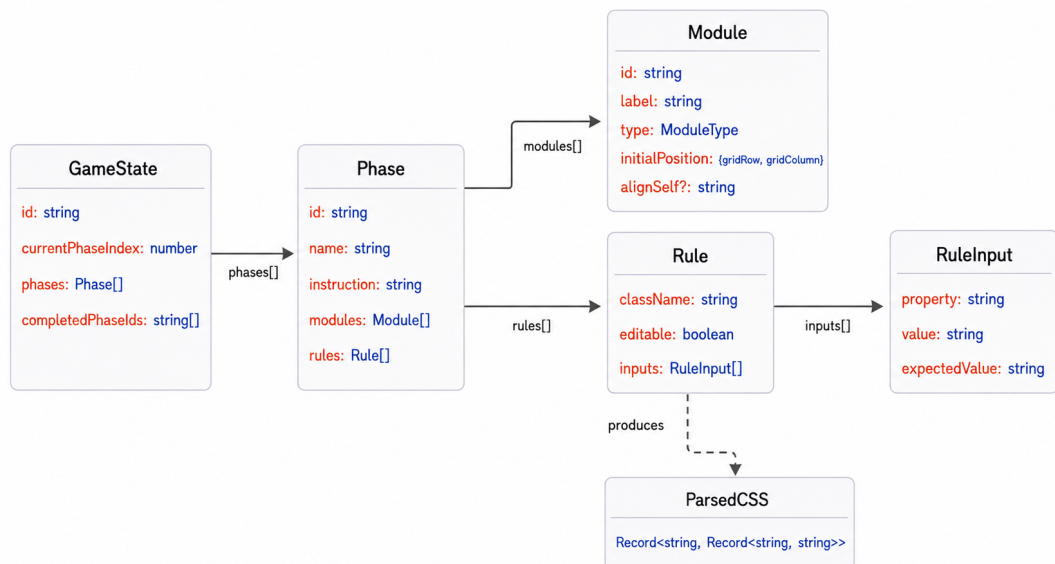
Fonte: própria autora.

4.4 Modelagem conceitual do sistema

A modelagem conceitual do *Tezza Space* organiza-se a partir de um conjunto de interfaces definidas em *TypeScript*, que estabelecem um padrão estrutural único para todas as fases do jogo. Essa formalização assegura que cada desafio siga a mesma organização de dados, facilitando

o controle do fluxo da aplicação e contribuindo para sua manutenção e evolução. As entidades centrais compreendem os módulos posicionados no tabuleiro, as propriedades [CSS](#) editáveis, as regras que as agrupam, a fase e o estado global da sessão, conforme ilustrado na [Figura 4.11](#), que apresenta o conjunto de entidades e seus relacionamentos.

Figura 4.11 – Diagrama de entidades da modelagem conceitual.



Fonte: própria autora.

A interface *Module* representa cada unidade visual disposta no tabuleiro orbital. Além de um identificador único, registrado no campo *id*, e de um rótulo descritivo definido em *label*, o módulo possui um campo *type*, do tipo *ModuleType*, que determina sua categoria visual, como o astronauta, o hangar ou o drone. A posição inicial é descrita pelo campo *initialPosition*, composto pelos atributos *gridRow* e *gridColumn*, ambos expressos como *strings* capazes de representar tanto coordenadas simples quanto intervalos e expansões. Há ainda o campo opcional *alignSelf*, que define um alinhamento vertical inicial aplicado antes de qualquer regra inserida pelo usuário.

A interface *RuleInput* representa uma propriedade [CSS](#) editável dentro de uma fase, reunindo o nome da propriedade no campo *property*, o valor informado pelo usuário em *value* e o valor esperado como solução em *expectedValue*. Caso o usuário informe um valor diferente que produza a mesma representação visual, esse resultado também é considerado válido pelo sistema. A interface *Rule*, por sua vez, articula um conjunto de propriedades a um alvo específico por meio do campo *className*, que pode referenciar tanto o identificador de um módulo individual quanto o identificador de uma classe de contêiner ou o tipo de um módulo. O campo *editable* determina se o usuário pode alterar os valores daquela regra, permitindo que algumas sejam exibidas apenas como contexto. O campo *inputs* agrupa uma ou mais instâncias de *RuleInput* sob o mesmo seletor, o que possibilita que um único mecanismo de regras atenda simultaneamente às propriedades do contêiner e às dos itens filhos.

A interface *Phase* reúne os elementos necessários à execução de cada desafio, contem-

plando um identificador, um nome descritivo exibido no painel de instruções, o texto narrativo da missão, o conjunto de módulos posicionados no tabuleiro e o conjunto de regras que compõem o console de edição. Dessa forma, cada fase reúne todas as informações necessárias para sua execução de maneira independente. O estado global da sessão é representado pela interface *GameState*, que mantém o identificador da sessão, o array *phases* com todas as fases disponíveis e o array *completedPhaseIds*, no qual se acumulam os identificadores das fases já concluídas pelo usuário. A fase ativa é referenciada pelo campo *currentPhaseIndex*, um inteiro que indica a posição no array de fases. A [Código 4.1](#) ilustra um exemplo prático dessa modelagem.

Código 4.1: Exemplo prático da estrutura de dados adotada para a representação de uma fase no *Tezza Space*.

```

1 {
2   "id": "phase-grid-column-start-lost-astronaut",
3   "name": "Fase 01 - O Astronauta Perdido",
4   "instruction": "O astronauta principal acordou em um setor errado apos uma falha no
   ⇨ teletransportador. Mova o astronauta para começar na coluna 4.",
5   "modules": [
6     {
7       "id": "lost-astronaut",
8       "label": "Astronauta Perdido",
9       "type": "ASTRONAUT",
10      "initialPosition": {
11        "gridRow": "4",
12        "gridColumn": "1"
13      }
14    },
15    {
16      "id": "energy-module",
17      "label": "Modulo de Energia",
18      "type": "POWER_CORE",
19      "initialPosition": {
20        "gridRow": "4",
21        "gridColumn": "5"
22      }
23    }
24  ],
25  "rules": [
26    {
27      "className": "lost-astronaut",
28      "editable": true,
29      "inputs": [
30        {
31          "property": "grid-column-start",
32          "value": "",
33          "expectedValue": "4"
34        }
35      ]
36    }
37  ]
38 }

```

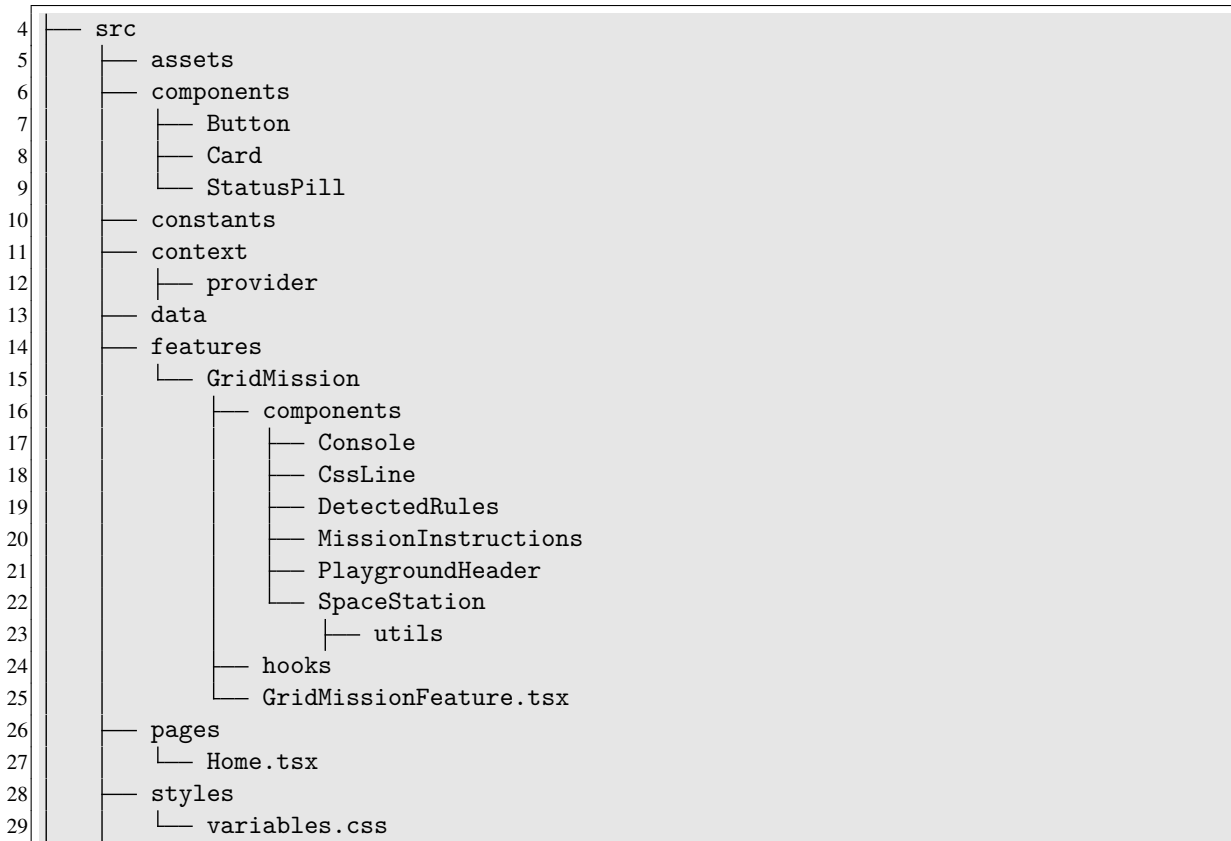
O modelo adota ainda tipos auxiliares que apoiam o processamento e a renderização

dos dados. O campo *expectedValue* coexiste diretamente no *RuleInput* junto à regra manipulada pelo usuário, de modo que a propriedade editável e o critério pelo qual ela é avaliada integram o mesmo objeto, conferindo coesão ao modelo e simplificando a lógica de validação. O tipo *ParsedCSS* mapeia cada *className* a um objeto de pares propriedade-valor, atuando como estrutura intermediária derivada das regras e posteriormente consumida pelos construtores de estilo. O tipo *ModuleType*, por sua vez, define as categorias visuais disponíveis para os módulos do jogo, funcionando como referência para a associação entre os elementos gráficos e as regras de estilo processadas pelo sistema.

4.5 Arquitetura da aplicação

A aplicação *Tezza Space* foi desenvolvida como um sistema *front-end* em *React* e *TypeScript*, organizado em camadas que separam os dados, a lógica de estado, a lógica da *feature* e a apresentação. Essa estrutura distribui as responsabilidades entre os diretórios *data*, *context*, *hooks* e *components*, de modo que os dados narrativos, o controle de estado e a interface permaneçam desacoplados. A [Código 4.2](#) apresenta a árvore de diretórios do projeto, evidenciando essa organização. A *feature* *GridMission* atua como unidade de composição central, agregando componentes, *hooks* e utilitários em torno de um único contexto funcional, o que confere coesão ao sistema sem comprometer a independência de suas partes.

Código 4.2: Árvore de diretórios do projeto.



```

30 | — test
31 | — types
32 | — utils
33 | — App.css
34 | — App.tsx
35 | — index.css
36 | — main.tsx

```

4.5.1 Gerenciamento de estado

O gerenciamento de estado da aplicação é realizado de forma centralizada por meio da *Context API*. O `GameContext` concentra as informações do estado global e as disponibiliza para toda a árvore de componentes, enquanto o `GameProvider` envolve a aplicação e inicializa os dados da sessão. O acesso a essas informações ocorre por meio de um *hook* dedicado, que retorna o estado corrente acompanhado dos mecanismos necessários para sua atualização. As regras de transição entre estados são concentradas em um único ponto da aplicação, de modo que toda alteração na sessão de jogo seja processada de maneira previsível e rastreável.

As alterações de estado são organizadas em três tipos de ação. O primeiro corresponde à navegação entre fases, atualizando a fase atualmente ativa. O segundo trata da edição das propriedades `CSS` pelo usuário, localizando a regra correspondente e atualizando seu conteúdo de forma a preservar a consistência dos demais dados. O terceiro registra a conclusão de uma fase, adicionando ou removendo seu identificador do histórico de desafios concluídos; quando o estado já corresponde ao esperado, a operação não produz alterações, evitando processamentos desnecessários.

Acima dessa camada de estado, um *hook* dedicado à *feature* transforma os dados brutos nas informações efetivamente utilizadas pela interface, expondo o nome e a instrução da fase, o número da fase atual, o total de fases, as regras editáveis, os módulos do tabuleiro e indicadores de progresso e conclusão. Esse *hook* também disponibiliza as funções responsáveis por atualizar as regras inseridas pelo usuário, registrar a conclusão da fase e avançar para o desafio seguinte. Durante esse processo, cada módulo recebe o componente visual correspondente ao seu tipo, de modo que a representação gráfica já esteja associada ao dado antes de chegar à interface.

4.5.2 Camada de apresentação

A camada de apresentação é composta por componentes especializados, cada um com responsabilidade bem delimitada. O `MissionInstructions` exibe o nome da fase e a instrução narrativa da missão, enquanto o `PlaygroundHeader` apresenta o número da fase atual, o total de fases e o indicador de conclusão, além de disponibilizar o controle de navegação para o desafio seguinte. O `Console` organiza as regras por seletor, exibindo cada bloco no formato convencional do `CSS`, com suas linhas editáveis dispostas de forma estruturada. O `CssLine` renderiza cada linha individual de código, permitindo a alteração da propriedade e do valor

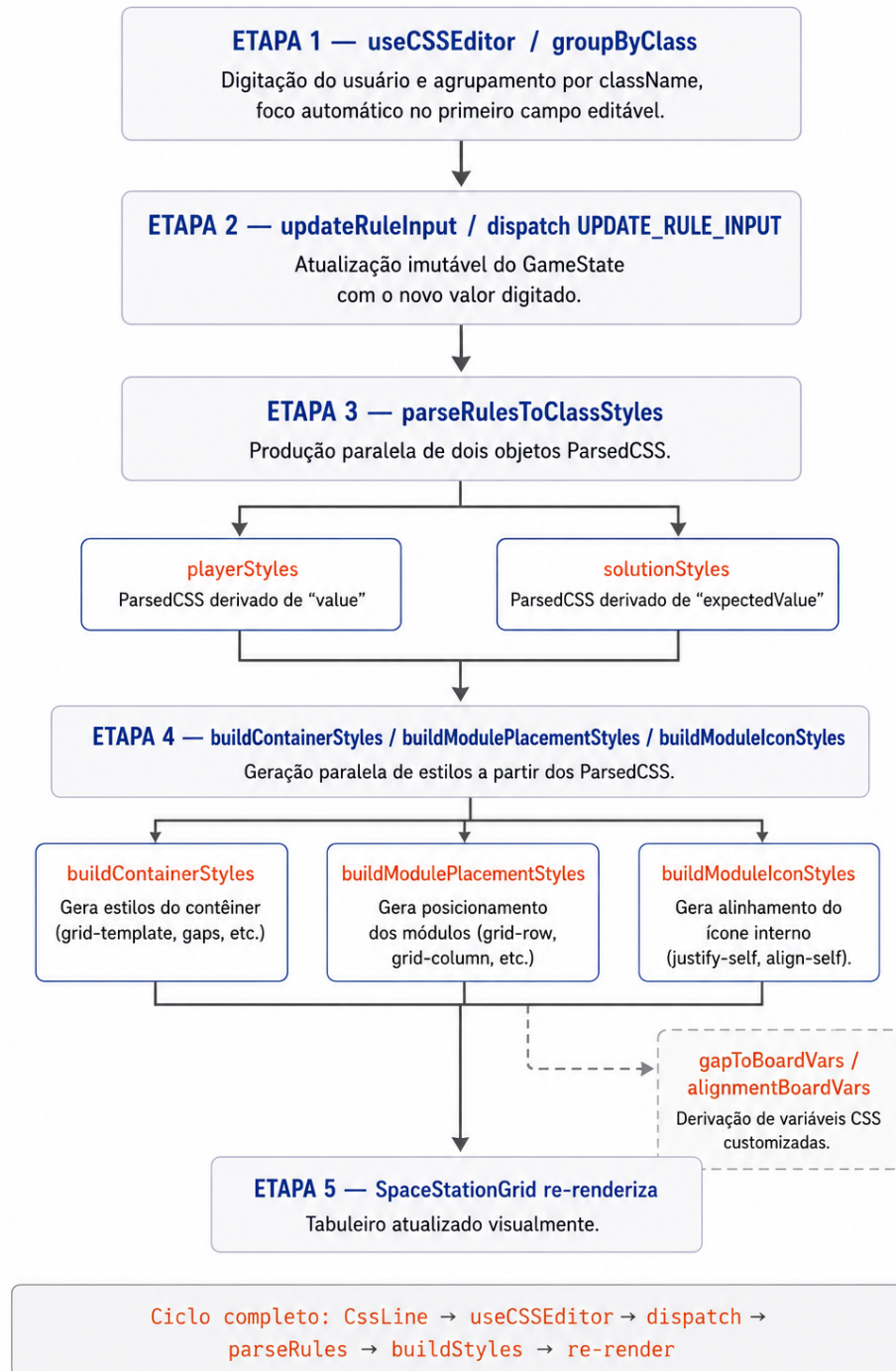
quando a regra correspondente admite edição. O `DetectedRules` apresenta as propriedades reconhecidas pelo sistema e indica quais delas foram corretamente definidas pelo usuário. O `SpaceStation` constitui o componente de maior complexidade dessa camada, sendo responsável por receber as regras processadas e os módulos com seus respectivos elementos visuais e produzir a representação gráfica da *grid* configurado.

Como princípio estruturante, nenhum desses componentes mantém lógica de controle própria, atuando como elementos reativos às informações fornecidas pelas camadas anteriores. Componentes de uso geral, como `Button`, `Card` e `StatusPill`, são utilizados pela *feature*, mas permanecem independentes dela, o que favorece sua reutilização em outros contextos da aplicação. Essa organização contribui para um comportamento mais previsível do sistema e facilita tanto sua manutenção quanto a incorporação de novos desafios e variações de configuração.

4.6 Fluxo de edição e processamento de regras

A edição e o processamento de regras no *Tezza Space* constituem um ciclo único e contínuo, no qual a digitação realizada pelo usuário percorre uma sequência de transformações até refletir-se na visualização do tabuleiro. Esse ciclo inicia-se na alteração de uma linha de código, prossegue com a atualização do estado global, passa pela conversão das regras em estruturas computáveis e culmina na construção dos estilos aplicados a *grid*. A articulação entre essas etapas garante a consistência entre a entrada fornecida e a resposta apresentada pelo sistema, de modo que cada modificação seja imediatamente interpretada e materializada sem ação adicional por parte do usuário. A Figura 4.12 apresenta uma visão geral desse fluxo de processamento.

Figura 4.12 – Visão geral do fluxo de edição e processamento de regras.



Fonte: própria autora.

A primeira etapa ocorre no Console, onde o usuário insere as declarações CSS relacionadas à fase em execução. O conjunto de regras é organizado por seletor, de modo que cada bloco reúna todas as linhas pertencentes ao mesmo *className*. O *hook* também posiciona automaticamente o cursor no primeiro campo editável ao carregar a fase. A cada edição realizada pelo usuário, a alteração é propagada ao estado global por meio do mecanismo centralizado de

gerenciamento, registrando imutavelmente o novo valor e mantendo a interface sincronizada com as ações realizadas.

A partir do estado atualizado, as regras são convertidas em objetos de estilo por uma função que percorre os *inputs* de cada seletor e agrega, para cada par de propriedade e valor não vazios, o resultado ao objeto correspondente. Esse processo é executado duas vezes de forma independente: uma sobre os valores digitados pelo usuário, produzindo o objeto de estilos do jogador, e outra sobre os valores esperados, produzindo o objeto de estilos da solução. A manutenção desses dois objetos de forma separada é determinante para o funcionamento do sistema, pois ambos precisam coexistir para viabilizar a renderização em duas camadas e fundamentar a validação visual da atividade.

Com os dois conjuntos de estilos disponíveis, três funções especializadas produzem estilos independentes a partir deles. A primeira aplica somente as propriedades reconhecidas como pertencentes ao contêiner, entre elas *grid-template-columns*, *grid-template-rows*, *grid-template-areas*, *gap*, *justify-content* e *align-items*, gerando o estilo do elemento pai da *grid*. A segunda inicializa cada módulo com sua posição definida e sobrepõe as propriedades de posicionamento de item, como *grid-column*, *grid-row* e suas variantes, produzindo o estilo de localização de cada elemento. A terceira aplica exclusivamente as propriedades de alinhamento individual, como *align-self* e *justify-self*, ao *asset* interno do módulo. Essa separação entre estilo do contêiner, posicionamento do item e alinhamento do módulo permite que o tabuleiro suporte qualquer propriedade do **CSS Grid** sem necessidade de lógica condicional na camada de visualização.

A comunicação entre as regras definidas e a aparência do tabuleiro é complementada por funções que derivam variáveis de estilo customizadas. Uma delas extrai dos estilos do contêiner as propriedades de espaçamento e gera as variáveis responsáveis pelo distanciamento visual entre as células. Outra analisa os estilos da solução e, quando neles constam propriedades de alinhamento de conteúdo, deriva variáveis que reduzem o tamanho visual das células, evidenciando o efeito de *justify-content* ou de *align-content* sobre a distribuição da grade. Esse mecanismo permite que a visualização se adapte sempre que o usuário manipula propriedades de espaçamento ou de alinhamento, tornando perceptíveis comportamentos que, de outra forma, permaneceriam pouco evidentes na *grid*. Quando a configuração produzida pelo usuário corresponde à solução de referência da fase, o desafio é considerado concluído e o progresso é atualizado automaticamente.

4.7 Funções auxiliares de suporte à *Grid* unificada

A capacidade de suportar qualquer propriedade do **CSS Grid** em uma única instância de tabuleiro exigiu a criação de uma camada de funções puras, executadas antes da renderização, responsáveis por classificar, transformar e resolver as regras inseridas pelo usuário. Essas funções estão organizadas em dez arquivos reunidos no diretório de utilitários do *SpaceStation* e

expostos de maneira centralizada, a partir da qual todos os *imports* do componente são realizados. Os arquivos distribuem-se em quatro grupos segundo suas responsabilidades: o primeiro trata da classificação de propriedades **CSS**; o segundo dedica-se à geometria e à resolução de posicionamento; o terceiro abrange a construção dos estilos e as regras de domínio do jogo; e o quarto concentra a validação visual. A modularização em funções independentes de estado favorece a manutenção, uma vez que cada arquivo concentra uma responsabilidade única e pode ser ajustado ou expandido sem afetar os demais.

O primeiro grupo trata da classificação de propriedades **CSS**. Um dos arquivos centraliza as dimensões fixas do tabuleiro, definidas como cinco colunas, cinco linhas e vinte e cinco células, de modo que qualquer função que necessite dessas medidas as importe de um único ponto. O outro arquivo concentra funções que operam exclusivamente sobre *strings*, sem dependência de tipos de domínio. Entre elas estão as responsáveis por identificar se uma propriedade pertence ao contêiner ou se corresponde a um auto-alinhamento, por indicar quais propriedades devem suprimir a sombra visual das células, por converter nomes de propriedades do formato **CSS** para o formato utilizado pelo *React* e por contabilizar o número de trilhas representadas por um valor de `grid-template-columns` ou `grid-template-rows`, expandindo expressões com `repeat` para calcular a quantidade de células do *overlay*.

O segundo grupo dedica-se à geometria e à resolução de posicionamento. Um arquivo define os tipos que representam intervalos de linhas e retângulos na grade, além de funções que convertem valores de posicionamento como "2 / 4", "1 / span 2" ou "3" nos intervalos correspondentes, detectam sobreposições entre elementos e calculam o retângulo que envolve uma coleção de módulos. Outro arquivo realiza o processamento, a comparação e o embaralhamento de valores de `grid-template-areas`, extraindo sua estrutura, verificando a equivalência semântica da resposta do usuário e gerando um *layout* embaralhado que compõe o estado inicial das fases de áreas nomeadas antes de qualquer inserção. A partir desses recursos, um terceiro arquivo resolve o retângulo de cada módulo considerando áreas nomeadas e abreviações de eixo, agrupando os módulos cujos retângulos se sobrepõem em um único *slot* cujo posicionamento corresponde à união dos retângulos envolvidos.

O terceiro grupo abrange a construção dos estilos e as regras de domínio do jogo. Uma função atua como ponto de entrada do processamento, mapeando cada seletor a pares de propriedade e valor para produzir o objeto de estilos do jogador ou o da solução, conforme a chave de leitura adotada. O mesmo arquivo concentra funções que relacionam regras e módulos, definindo quais elementos entram na validação e identificando aqueles sujeitos a regras de auto-alinhamento. Outro arquivo reúne os construtores de estilos responsáveis pela separação entre estilo do contêiner, posicionamento do item e alinhamento do módulo, ao passo que um terceiro compõe a lista de classes **CSS** aplicada a cada *asset* renderizado, combinando a classe base, a classe condicional de edição e o identificador do módulo.

O quarto grupo corresponde à validação visual. As funções desse arquivo fundamentam

a verificação da solução na comparação das dimensões e posições dos elementos renderizados. Uma delas obtém a caixa delimitadora arredondada de um elemento e outra compara a posição e o tamanho de dois elementos, indicando se são idênticos. Uma terceira estende essa lógica ao conjunto de módulos sob validação, verificando se cada elemento construído pelo usuário coincide com o correspondente da solução; quando há regra de auto-alinhamento, a comparação recai sobre o ícone interno do módulo e não sobre o módulo inteiro, o que permite avaliar o alinhamento individual com precisão. O resultado positivo dessa verificação constitui o sinal que aciona a conclusão da fase, integrando a validação ao fluxo de progressão do jogo.

A organização desses dez arquivos em torno de responsabilidades isoladas estabelece a base sobre a qual o tabuleiro unificado se sustenta. Cada função recebe entradas explícitas e produz saídas determinísticas, sem manter estado próprio, de maneira que a interpretação das regras permaneça desacoplada da camada de visualização. Essa separação permite a incorporação de novas propriedades ou comportamentos com alterações pontuais, preservando a coesão interna do componente de tabuleiro e a previsibilidade de seu funcionamento.

4.8 Renderização no tabuleiro orbital

A materialização visual das regras processadas nas etapas anteriores ocorre no componente *SpaceStation*, responsável por aplicar os estilos gerados a partir das entradas do usuário ao tabuleiro orbital. Durante esse processo, as propriedades relacionadas à configuração da *grid*, como *grid-template-columns*, *grid-template-rows* e *gap*, são aplicadas ao contêiner principal, enquanto as propriedades de posicionamento e alinhamento são aplicadas individualmente a cada módulo presente no cenário.

Para possibilitar a validação automática das atividades, o sistema mantém simultaneamente duas representações da mesma *grid*. A primeira corresponde à *grid* do jogador, construída a partir das regras digitadas durante a resolução da fase. A segunda corresponde à *grid* de solução, gerada a partir dos valores esperados definidos na estrutura da atividade. Enquanto a *grid* do jogador permanece visível e interativa, a *grid* de solução é utilizada apenas internamente como referência para comparação. Essa abordagem garante que ambas as representações sejam produzidas sob as mesmas condições, permitindo que a comparação entre elas seja direta e consistente.

O tabuleiro adota dois comportamentos distintos conforme o tipo de propriedade trabalhado na fase. Nas fases que não empregam áreas nomeadas, é exibido um guia visual de células vazias que reflete em tempo real as colunas e linhas estabelecidas pelo usuário. Nas fases que utilizam *grid-template-areas*, esse guia é substituído por blocos de áreas nomeadas; quando o usuário ainda não inseriu um valor válido, o tabuleiro apresenta um *layout* embaralhado que serve de ponto de partida visual para a resolução do desafio.

A validação ocorre pela comparação das posições ocupadas pelos módulos nas duas

grids. Para cada elemento participante da atividade, o sistema verifica se sua posição e dimensões correspondem ao resultado esperado. Nas fases que envolvem propriedades de alinhamento individual, como *justify-self* e *align-self*, a comparação recai sobre o conteúdo interno do módulo, respeitando o comportamento real dessas propriedades. Nas fases com áreas nomeadas, a fase só é considerada concluída quando tanto a comparação visual das posições quanto a verificação dos valores definidos resultam corretas. Nem todos os elementos presentes no cenário participam desse processo; apenas os módulos diretamente relacionados ao objetivo da fase são avaliados, enquanto os elementos utilizados exclusivamente para compor a narrativa visual são desconsiderados. Quando todas as condições são atendidas, o sistema registra automaticamente a conclusão da fase e atualiza o progresso do usuário, fornecendo retorno imediato sobre o resultado obtido.

A aparência do tabuleiro também se adapta conforme as propriedades manipuladas pelo usuário. O espaçamento entre as células é ajustado de acordo com os valores de *gap* definidos, e o tamanho visual das células é reduzido quando a solução envolve propriedades de alinhamento de conteúdo, tornando perceptíveis comportamentos que, de outra forma, permaneceriam pouco evidentes na interface.

4.9 Testes automatizados do Tezza Space

Com o objetivo de assegurar a correção dos mecanismos descritos ao longo deste capítulo, foi implementado um conjunto de testes automatizados utilizando o *Jest* como executor de testes e a *React Testing Library* como ferramenta de verificação dos componentes a partir da perspectiva do usuário, complementada pela biblioteca *user-event* para a simulação de interações como digitação, cliques e navegação por teclado. Os testes foram organizados em quatro frentes, alinhadas às etapas descritas ao longo do capítulo: os componentes de interface, o gerenciamento de estado, os utilitários de conversão e posicionamento das regras na *grid*, e a integração completa do fluxo de uma missão.

Na camada de interface, componentes básicos como *Button*, *Card* e *StatusPill* tiveram seu comportamento verificado quanto à acessibilidade e à resposta a interações, enquanto os componentes do *playground*, como *Console*, *CssLine*, *DetectedRules*, *MissionInstructions* e *PlaygroundHeader*, foram testados quanto à edição de regras, à navegação entre linhas de comando e à exibição do progresso da missão, incluindo o bloqueio de edição em linhas não editáveis e a exibição de dicas.

Na camada de gerenciamento de estado, descrita nas Seções 4.5.1 e 4.4, os testes do *gameReducer* verificam as transições de estado, como a atualização da fase corrente, a alteração de valores de regras específicas sem interferência sobre fases não relacionadas e a marcação de fases como concluídas, preservando a imutabilidade do estado quando não há alteração efetiva. As suítes de *GameContext*, *useGame* e *GameProvider* complementam essa verificação ao validar

a propagação desse estado para os componentes consumidores, tanto no valor padrão exposto quanto na integração com um provedor real.

Relacionado à conversão das regras em estilos aplicáveis e à renderização do tabuleiro, discutidas nas Seções 4.6 a 4.8, um conjunto mais extenso de utilitários foi testado de forma isolada da camada visual. Os testes de *cssUtils* e *cssProperties* cobrem a interpretação de linhas de código digitadas no console e a classificação de propriedades CSS em categorias relevantes, como propriedades de contêiner, de espaçamento e de autoalinhamento. Os testes de *gridGeometry* e *gridAreas* validam o cálculo de posições e sobreposições na *grid* e a interpretação de declarações de *grid-template-areas*, incluindo o embaralhamento de áreas nomeadas para a geração de novos desafios. Os testes de *moduleLayout*, *moduleRules* e *styles* verificam o agrupamento de módulos que compartilham a mesma célula, a correspondência entre regras e módulos, e a construção dos estilos finais de posicionamento e alinhamento. Complementando esse grupo, os testes de *collision* validam a comparação visual entre a posição alcançada pelo jogador e a posição esperada pela solução, a partir das coordenadas retornadas pelo navegador para cada elemento renderizado.

Por fim, *hooks* como *useCssEditor*, *useGridMission* e *useGridValidation* foram testados de forma a validar a orquestração entre a edição de código, a validação das regras e a progressão das fases, enquanto os testes de *GridMissionFeature* reproduzem o fluxo completo de uma missão do início à conclusão, simulando a camada de renderização do tabuleiro por meio de um componente substituto. Essa suíte confirma a atualização do *feedback* visual conforme o valor digitado, a habilitação do avanço de fase mediante a conclusão da missão e a reinicialização de recursos como dica e solução ao avançar para a fase seguinte, evidenciando a coerência entre as etapas descritas ao longo do Capítulo 4.

A Tabela 4.2 sintetiza as suítes implementadas por camada.

Tabela 4.2 – Suítes de teste automatizado implementadas no sistema *Tezza Space*

Camada	Suítes	Testes
Componentes de interface	<i>Button, Card, StatusPill</i>	6
Componentes do <i>playground</i>	<i>Console, CssLine, DetectedRules, MissionInstructions, PlaygroundHeader</i>	22
Gerenciamento de estado	<i>gameActions, gameReducer, GameContext, useGame, GameProvider</i>	18
Conversão de regras e geometria da <i>grid</i>	<i>cssUtils, cssProperties, normalizeRules, gridGeometry, gridAreas, moduleLayout, moduleRules, moduleClassList, styles, constants/index</i>	67
Validação visual e renderização do tabuleiro	<i>collision, useGridValidation, SpaceStationGrid</i>	16
Orquestração e fluxo completo	<i>useCssEditor, useGridMission, gameUtils, GridMissionFeature</i>	22

Fonte: própria autora.

No total, foram implementados 151 testes, distribuídos entre as camadas de interface, de

gerenciamento de estado, de utilitários de conversão e geometria da *grid*, e de orquestração das fases, o que reforça a consistência do fluxo entre a edição de regras, o processamento do estado e a materialização visual no *playground*. Em termos de cobertura geral do sistema, o relatório gerado pelo *Jest* indicou 97,8% de instruções (*statements*), 94,31% de ramos condicionais (*branches*), 95,76% de funções (*functions*) e 97,91% de linhas (*lines*) cobertas pelas suítes implementadas.

A Figura 4.13 apresenta o relatório de cobertura gerado pelo *Jest* ao final da execução da suíte completa, detalhando os percentuais de instruções (*statements*), ramos condicionais (*branches*), funções e linhas cobertas por arquivo. Os valores confirmam uma cobertura consistentemente alta nos módulos relacionados à lógica de negócio e aos utilitários de conversão e geometria da *grid*, com poucas exceções concentradas em arquivos de configuração e entrada da aplicação, como *App.tsx* e *Home.tsx*, que não possuem lógica testável relevante e, por esse motivo, permanecem fora do escopo direto das suítes implementadas.

Figura 4.13 – Relatório de cobertura de testes gerado pelo *Jest*

Coverage Report					
File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	97.8	94.31	95.76	97.91	
src	0	100	0	0	
App.tsx	0	100	0	0	1-14
src/assets	100	100	81.08	100	
index.ts	100	100	81.08	100	
src/components/Button	100	100	100	100	
index.tsx	100	100	100	100	
src/components/Card	100	100	100	100	
index.tsx	100	100	100	100	
src/components/StatusPill	100	100	100	100	
index.tsx	100	100	100	100	
src/constants	100	100	100	100	
moduleTypes.ts	100	100	100	100	
src/context	97.05	87.5	100	96.66	
GameContext.tsx	100	100	100	100	
gameReducer.ts	100	92.85	100	100	25
useGame.ts	87.5	50	100	85.71	8
src/context/provider	100	100	100	100	
GameProvider.tsx	100	100	100	100	
src/data	88.23	100	85.71	90	
childNarrative.ts	100	100	100	100	
gameInitialState.ts	100	100	100	100	
index.ts	0	100	0	0	1-3
parentNarrative.ts	100	100	100	100	
src/features/GridMission	100	75	100	100	
GridMissionFeature.tsx	100	75	100	100	44
src/features/GridMission/components/Console	100	100	100	100	
Console.tsx	100	100	100	100	
src/features/GridMission/components/CssLine	100	100	100	100	
CssLine.tsx	100	100	100	100	
src/features/GridMission/components/DetectedRules	100	100	100	100	
DetectedRules.tsx	100	100	100	100	
src/features/GridMission/components/MissionInstructions	100	100	100	100	
MissionInstructions.tsx	100	100	100	100	
src/features/GridMission/components/PlaygroundHeader	100	100	100	100	
PlaygroundHeader.tsx	100	100	100	100	
src/features/GridMission/components/SpaceStation	100	100	100	100	
SpaceStation.tsx	100	100	100	100	
src/features/GridMission/components/SpaceStation/Utils	99.45	94.55	100	100	
collision.ts	100	89.47	100	100	65-66
constants.ts	100	100	100	100	
cssProperties.ts	100	87.5	100	100	85
gridAreas.ts	100	100	100	100	
gridGeometry.ts	100	100	100	100	
index.ts	100	100	100	100	
moduleClassList.ts	100	100	100	100	
moduleLayout.ts	98.68	91.84	100	100	77-85,91-93,121
moduleRules.ts	98.03	94.73	100	100	24
styles.ts	100	98.07	100	100	119
src/features/GridMission/hooks	98.79	89.28	100	100	
useCastEditor.ts	100	100	100	100	
useGridMission.tsx	96.29	85.71	100	100	115-125
useGridValidation.ts	100	87.5	100	100	89
src/pages	0	100	0	0	
Home.tsx	0	100	0	0	1-7
src/test	100	100	100	100	
renderWithProviders.tsx	100	100	100	100	
src/Utils	100	82.35	100	100	
cssUtils.ts	100	85.71	100	100	10-14
gameUtils.ts	100	100	100	100	
normalizeRules.ts	100	0	100	100	3
Test Suites: 31 passed, 31 total					
Tests: 151 passed, 151 total					
Snapshots: 0 total					
Time: 17.928 s					
Ran all test suites.					

Fonte: própria autora.

5 As fases da missão *Tezza Space*

Este capítulo apresenta a análise dos resultados obtidos a partir do desenvolvimento e do funcionamento do *Tezza Space*, organizando a discussão em torno das vinte fases que compõem o percurso pedagógico da plataforma. As fases foram estruturadas em dois módulos complementares: o primeiro, composto pelas fases 01 a 10, dedica-se ao ensino das propriedades aplicadas aos *Grid Items*, concentrando-se no posicionamento, na extensão e no alinhamento individual dos elementos filhos dentro da *grid*; o segundo, composto pelas fases 11 a 20, aborda as propriedades do *Grid Container*, por meio das quais o estudante passa a controlar a definição estrutural da *grid*, seus espaçamentos e o alinhamento coletivo dos elementos. Essa organização reflete a distinção conceitual entre os dois níveis de configuração do *CSS Grid*, apresentada na fundamentação teórica, e orienta a progressão didática do sistema de modo que cada módulo construa sobre os conhecimentos adquiridos no anterior.

O código-fonte da aplicação está disponível publicamente no repositório *GitHub*, acessível por meio do endereço <<https://github.com/sthephanytezza/tezza-space>>. Além disso, uma versão funcional do sistema encontra-se publicada na Web e pode ser acessada em <<https://sthephanytezza.github.io/tezza-space/>>, permitindo a utilização da ferramenta sem a necessidade de instalação local.

5.1 Visualização do posicionamento dos *Grid Items*

O conjunto de dez fases voltadas ao ensino das propriedades aplicadas aos *Grid Items* revela um padrão comportamental consistente, cuja regularidade permite identificar dois estados recorrentes da interface: o estado inicial, caracterizado pelo desalinhamento narrativo e visual do elemento manipulado, e o estado final, marcado pela atualização imediata da *grid* em resposta à declaração *CSS* inserida pelo usuário. Em todas as fases analisadas, esse padrão se sustenta de forma análoga: o campo de valor da propriedade apresenta-se vazio no editor de código, o elemento encontra-se posicionado em uma célula incompatível com o objetivo descrito no painel de instruções e a área de visualização evidencia explicitamente o contraste entre a configuração atual e a configuração esperada. Na fase 01, por exemplo, o astronauta ocupa a primeira coluna da grade enquanto o módulo de energia permanece na quarta, e a propriedade `grid-column-start` aguarda um valor que ainda não foi definido; essa disposição inicial cria uma relação direta entre o erro espacial representado no tabuleiro e a tarefa técnica que o aprendiz deve executar.

Após a inserção do valor correto, a renderização do tabuleiro orbital é atualizada de maneira imediata, sem qualquer ação adicional por parte do usuário, e o elemento passa a ocupar a coordenada esperada pela missão. Esse comportamento mantém-se uniforme ao longo das fases que abordam propriedades distintas, como `grid-row-start`, `grid-column-end`, `grid-column`

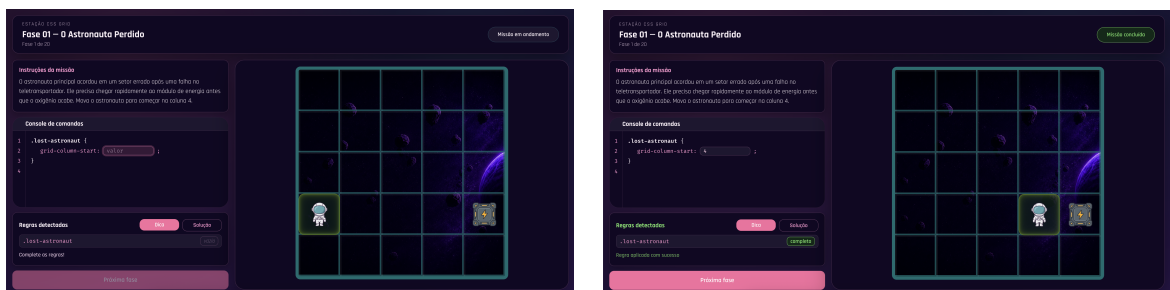
com operador `span`, `grid-row` com operador `span`, a forma abreviada de `grid-column`, `align-self`, `justify-self` e as duas variantes de `grid-area`.

Em cada caso, a alteração visual restringe-se exclusivamente ao elemento manipulado, preservando a disposição de todos os demais componentes da interface. Nas fases em que a propriedade atua sobre o posicionamento e a extensão do item, a exemplo das fases 03, 04, 05 e 06, o resultado observado consiste na ampliação da área ocupada pelo elemento em uma ou mais trilhas da *grid*, sem que ocorra o deslocamento dos itens adjacentes. As fases orientadas ao alinhamento interno, correspondentes às fases 07 e 08, mantêm inalterada a célula ocupada pelo elemento, alterando exclusivamente a sua posição dentro desse espaço delimitado. Esse comportamento evidencia a distinção conceitual entre o posicionamento estrutural e o alinhamento de conteúdo no modelo do *CSS Grid*. Complementarmente, a fase 10 exige a aplicação da sintaxe abreviada por meio de quatro coordenadas, consolidando os mecanismos introduzidos anteriormente ao demonstrar a capacidade da plataforma de sustentar operações de posicionamento simultâneo em ambos os eixos de maneira coerente e visualmente verificável.

5.1.1 Fase 01 — O Astronauta Perdido

A jornada de aprendizado se inicia com o desafio do astronauta perdido, cuja mecânica introduz o conceito de posicionamento horizontal por meio da propriedade `grid-column-start`. Na configuração inicial da interface, o personagem encontra-se isolado na primeira coluna, distante do módulo de energia fixado na quarta coluna. Ao preencher o campo vazio no seletor `.lost-astronaut` com o valor correto, a renderização atualiza-se imediatamente, conforme ilustrado na Figura 5.1.

Figura 5.1 – O Astronauta Perdido.



(a) Tabuleiro no estado padrão.

(b) Tabuleiro após regras.

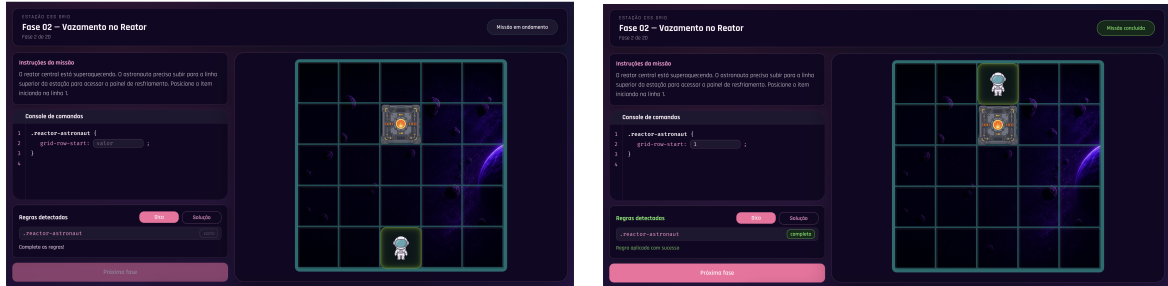
Fonte: própria autora.

5.1.2 Fase 02 — Vazamento no Reator

Na sequência, o desafio voltado ao vazamento no reator desloca o foco para o eixo vertical por meio da propriedade `grid-row-start`. O cenário apresenta o astronauta na extremidade inferior da *grid*, enquanto o objetivo pedagógico exige seu deslocamento até a área superaquecida localizada na parte superior da interface. A Figura 5.2 apresenta os dois estados do tabuleiro: antes

e após a aplicação do valor numérico adequado no seletor `.reactor-astronaut`, promovendo o movimento vertical esperado.

Figura 5.2 – Vazamento no Reator.



(a) Tabuleiro no estado padrão.

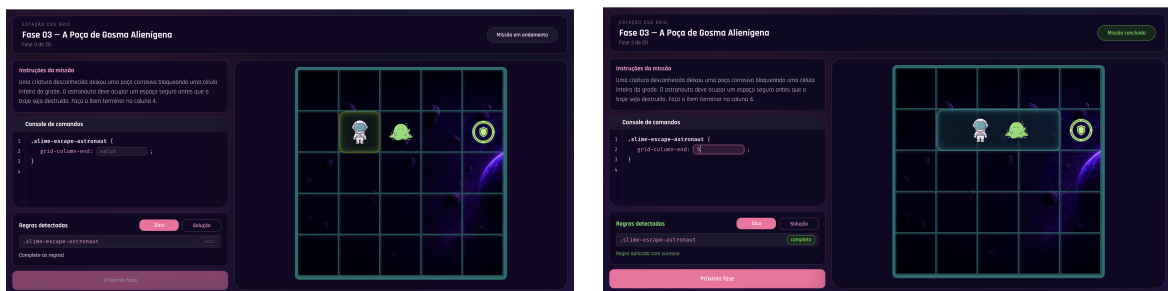
(b) Tabuleiro após regras.

Fonte: própria autora.

5.1.3 Fase 03 — A Poça de Gosma Alienígena

A expansão horizontal e o controle do limite final da ocupação de uma célula são trabalhados no terceiro cenário, denominado Poça de Gosma Alienígena, que faz uso da propriedade `grid-column-end`. Diante de um bloqueio no percurso à direita, torna-se necessário fazer com que o astronauta amplie sua área ocupada para contornar o obstáculo. Assim que o seletor `.slime-escape-astronaut` recebe a coordenada de término adequada, o item passa a estender-se por múltiplas colunas de forma contínua, comportamento evidenciado na Figura 5.3.

Figura 5.3 – A Poça de Gosma Alienígena.



(a) Tabuleiro no estado padrão.

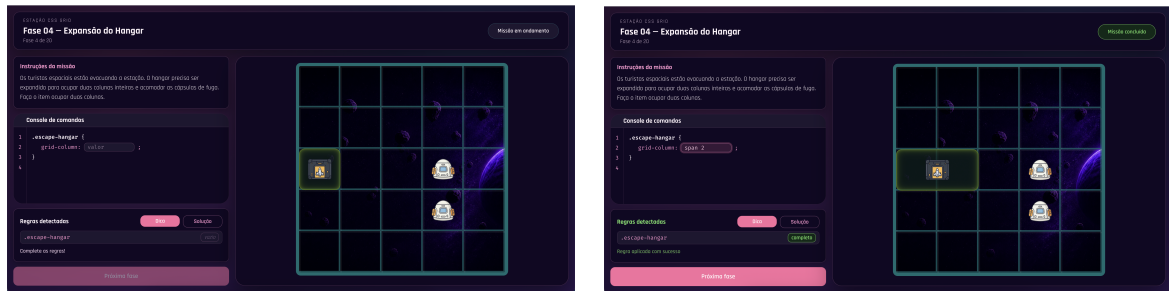
(b) Tabuleiro após regras.

Fonte: própria autora.

5.1.4 Fase 04 — Expansão do Hangar

Uma abordagem distinta para a ampliação de elementos ocorre no desafio do hangar espacial, em que se introduz o uso combinado da propriedade `grid-column` com o operador `span`. Em vez de definir linhas delimitadoras específicas, a atividade estimula o raciocínio baseado na contagem de trilhas consecutivas. A Figura 5.4 apresenta o estado inicial com o hangar restrito a um único quadrante e, após a inserção da instrução `span 2` no seletor `.escape-hangar`, a estrutura passando a ocupar o dobro do espaço horizontal original a partir de sua posição inicial.

Figura 5.4 – Expansão do Hangar.



(a) Tabuleiro no estado padrão.

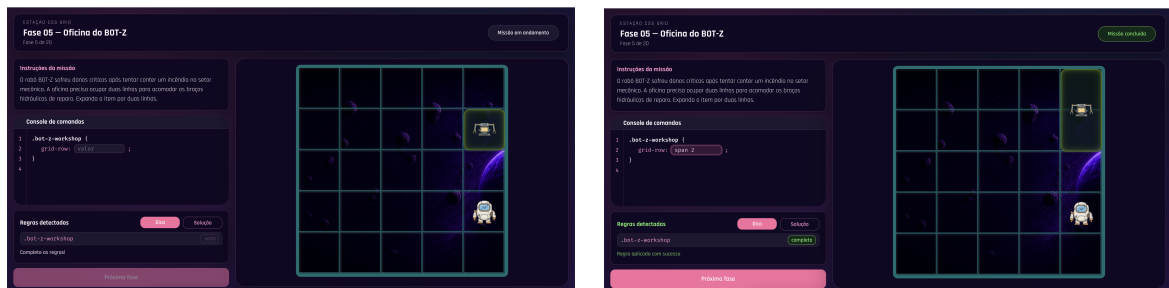
(b) Tabuleiro após regras.

Fonte: própria autora.

5.1.5 Fase 05 — Oficina do BOT-Z

A transposição do conceito de dimensionamento para o plano vertical é consolidada no desafio da oficina do BOT-Z, no qual a propriedade `grid-row` é utilizada em conjunto com o modificador `span`. O problema pedagógico consiste em ampliar a oficina, inicialmente limitada a uma única linha, para alcançar um robô danificado posicionado nas trilhas inferiores da *grid*. Ao inserir o comando `span 2` no seletor `.bot-z-workshop`, o elemento passa a expandir-se verticalmente pelas linhas adjacentes, ocupando uma área maior do layout. Esse comportamento pode ser observado na Figura 5.5, que apresenta a configuração inicial e o resultado obtido após a aplicação da instrução correta.

Figura 5.5 – Oficina do BOT-Z.



(a) Tabuleiro no estado padrão.

(b) Tabuleiro após regras.

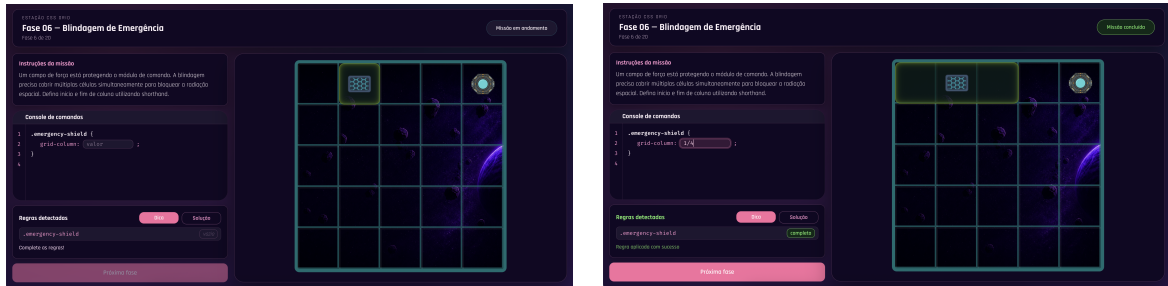
Fonte: própria autora.

5.1.6 Fase 06 — Blindagem de Emergência

A transição para abordagens mais avançadas tem início no desafio da blindagem de emergência, cenário projetado para apresentar a sintaxe abreviada da propriedade `grid-column`. Nessa atividade, os conceitos de posicionamento inicial e final, explorados separadamente nos desafios anteriores, são combinados em uma única declaração. A Figura 5.6 ilustra o comportamento da mecânica: inicialmente restrita a uma célula isolada, a estrutura de proteção passa a ocupar três colunas consecutivas após o preenchimento do seletor `.emergency-shield` com a instrução `1 / 4`. Dessa forma, o jogador compreende como a propriedade abreviada permite definir, em

uma única expressão, tanto o ponto de início quanto o de término da ocupação horizontal do elemento.

Figura 5.6 – Blindagem de Emergência.



(a) Tabuleiro no estado padrão.

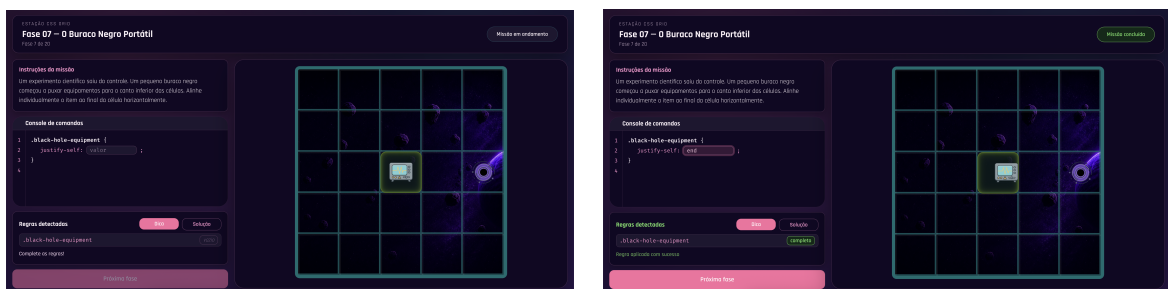
(b) Tabuleiro após regras.

Fonte: própria autora.

5.1.7 Fase 07 — O Buraco Negro Portátil

A dinâmica sofre uma alteração conceitual significativa a partir do desafio do buraco negro portátil, cujo objetivo é introduzir a propriedade `justify-self`. Diferentemente dos mecanismos de posicionamento estrutural explorados até então, essa atividade concentra-se no alinhamento interno do elemento dentro da célula que já ocupa na *grid*. A configuração inicial apresenta o equipamento centralizado em razão do alinhamento padrão definido pelo contêiner. Ao inserir o valor `end` no seletor `.black-hole-equipment`, o item é deslocado para o limite direito de sua própria célula, sem alterar sua posição na estrutura da grade, conforme ilustrado na Figura 5.7.

Figura 5.7 – O Buraco Negro Portátil.



(a) Tabuleiro no estado padrão.

(b) Tabuleiro após regras.

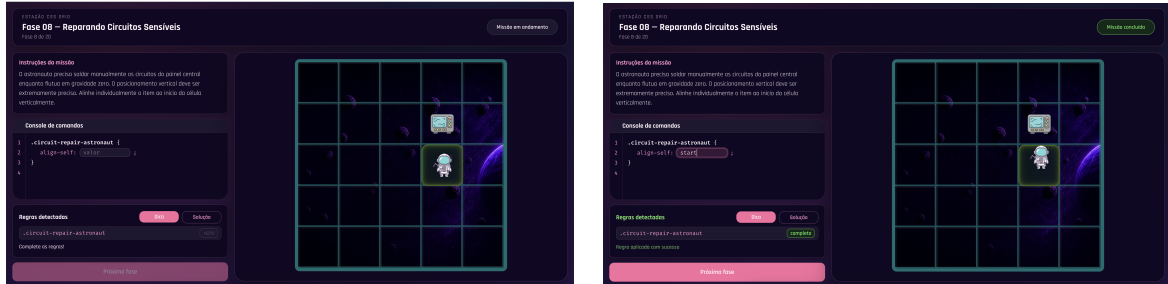
Fonte: própria autora.

5.1.8 Fase 08 — Reparando Circuitos Sensíveis

O mesmo princípio de controle individual do alinhamento é retomado no plano vertical durante o desafio de reparo de circuitos, fundamentado na propriedade `align-self`. Nesse cenário, o astronauta encontra-se afastado do ponto de contato elétrico localizado na parte superior da área ocupada. A Figura 5.8 apresenta a transformação resultante da atividade: ao preencher o

seletor `.circuit-repair-astronaut` com o valor `start`, o personagem é reposicionado junto à borda superior da célula, estabelecendo contato com o ponto de reparo.

Figura 5.8 – Reparando Circuitos Sensíveis.



(a) Tabuleiro no estado padrão.

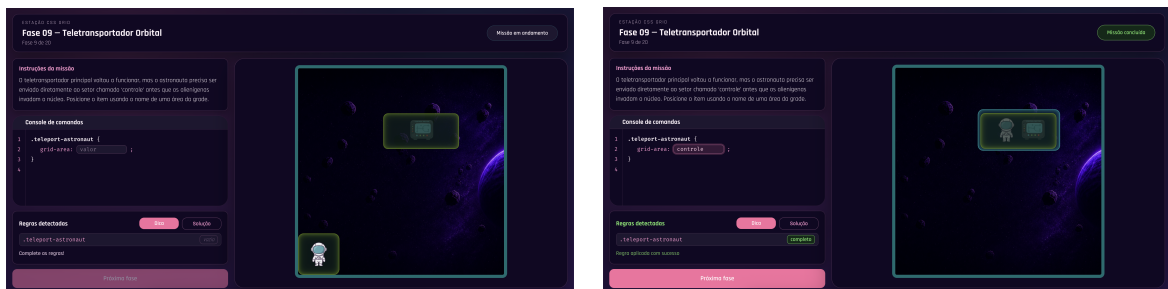
(b) Tabuleiro após regras.

Fonte: própria autora.

5.1.9 Fase 09 — Teletransportador Orbital

O nono desafio adota uma abordagem mais semântica ao introduzir o posicionamento por áreas nomeadas por meio da propriedade `grid-area`. Em vez de utilizar índices numéricos ou contagem de trilhas, o estudante passa a controlar o teletransportador orbital associando o elemento a um identificador textual definido previamente na estrutura da grade. Na Figura 5.9, o astronauta encontra-se inicialmente posicionado em uma região periférica e, após a inserção da `string` controle no seletor `.teleport-astronaut`, é instantaneamente transferido para a área central do painel.

Figura 5.9 – Teletransportador Orbital.



(a) Tabuleiro no estado padrão.

(b) Tabuleiro após regras.

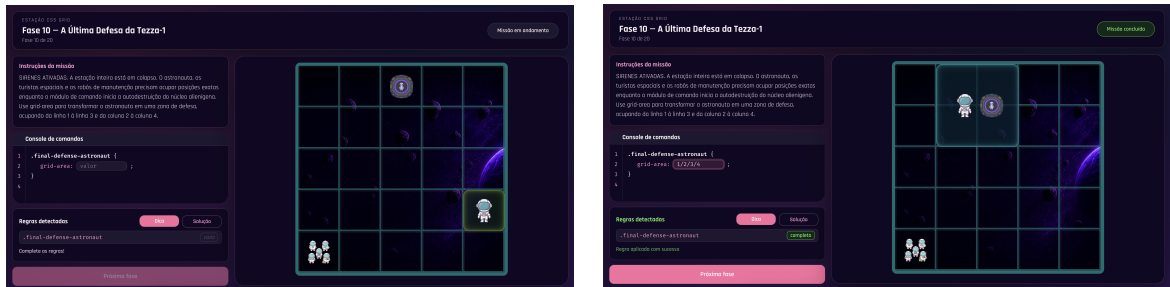
Fonte: própria autora.

5.1.10 Fase 10 — A Última Defesa da Tezza-1

O encerramento do módulo dedicado aos *Grid Items* ocorre no desafio final, que apresenta a sintaxe abreviada completa da propriedade `grid-area`. A atividade exige a definição simultânea das quatro coordenadas responsáveis por delimitar uma área da grade em uma única declaração. No estado inicial, o astronauta ocupa apenas uma célula isolada; após a inserção da sequência `1 / 2 / 3 / 4` no seletor `.final-defense-astronaut`, o elemento passa a ocupar toda a

região compreendida entre as coordenadas especificadas. Conforme demonstrado na Figura 5.10, essa mecânica consolida os conhecimentos adquiridos ao longo do módulo e conclui a etapa de aprendizagem voltada ao posicionamento e dimensionamento de elementos filhos em *CSS Grid*.

Figura 5.10 – A Última Defesa da Tezza-1.



(a) Tabuleiro no estado padrão.

(b) Tabuleiro após regras.

Fonte: própria autora.

5.2 Visualização da estrutura e organização do *Grid Container*

O segundo módulo, composto pelas fases 11 a 20, introduz uma natureza de interação qualitativamente distinta da observada no primeiro. Enquanto as fases anteriores concentravam a manipulação em um único elemento da *grid*, mantendo a estrutura do *container* estática, as fases do segundo módulo deslocam o foco de atuação para o próprio *container*, de modo que cada declaração *CSS* aplicada reconfigura a malha da *grid* e redistribui automaticamente todos os elementos nela contidos. Esse comportamento fica evidenciado já na fase 11, em que a inserção do valor `1fr 1fr` na propriedade `grid-template-columns` do seletor `.hangar-grid` transforma uma distribuição espacial inadequada em duas colunas simétricas, sem que qualquer propriedade seja aplicada diretamente aos itens. A alteração produzida não recai sobre um elemento isolado, mas sobre a estrutura que os organiza, o que exige do acadêmico uma compreensão distinta da relação entre código e *layout*.

Ao longo das dez fases deste módulo, os resultados obtidos demonstram que o sistema responde corretamente a propriedades de naturezas variadas. As fases 11, 12 e 13 verificam a capacidade da plataforma de reconfigurar as trilhas da *grid* por meio de `grid-template-columns` e `grid-template-rows`, com valores tanto simétricos quanto proporcionalmente assimétricos, como no caso da fase 13, em que a declaração `2fr 1fr 2fr` produz colunas laterais mais largas que a central. As fases 14, 15 e 16 testam as propriedades de espaçamento `gap`, `row-gap` e `column-gap`, evidenciando que o sistema interpreta corretamente tanto a configuração simultânea de ambos os eixos quanto a configuração independente de cada um deles, preservando inalterado o espaçamento não afetado pela propriedade declarada.

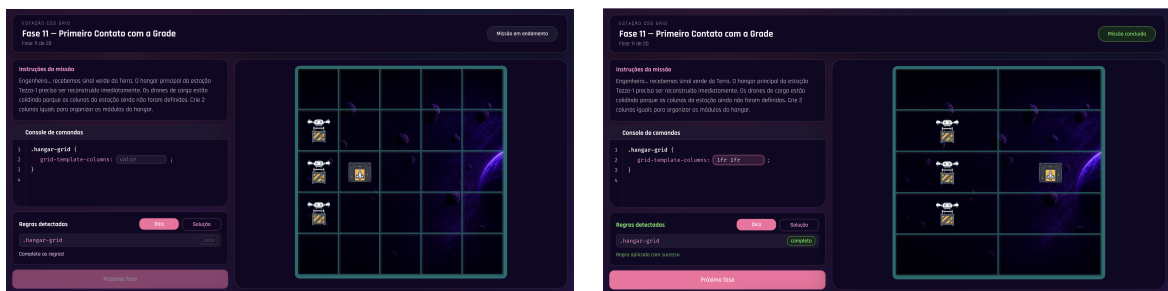
As fases 17 e 18, dedicadas a `justify-items` e `align-items`, demonstram que o mecanismo de renderização é capaz de aplicar alinhamento coletivo a todos os itens da *grid* a partir de uma única declaração no *container*, sem modificar as coordenadas individuais de cada

elemento na estrutura. A fase 19, com a propriedade `justify-content`, amplia esse escopo ao verificar o alinhamento da *grid* como totalidade dentro do espaço disponível do *container*, e a fase 20 encerra o módulo com `grid-template-areas`, confirmando que o sistema processa corretamente a definição de regiões nomeadas e reorganiza todos os módulos da estação segundo o mapeamento declarado.

5.2.1 Fase 11 — Primeiro Contato com a *Grid*

O início do módulo voltado aos elementos organizadores da estrutura ocorre na décima primeira fase, cuja proposta introduz a criação de colunas por meio da propriedade `grid-template-columns`, utilizando a unidade fracionária. Na configuração inicial, o selector `.hangar-grid` apresenta essa propriedade sem definição, o que resulta em uma distribuição inadequada dos elementos na tela. Ao inserir o valor `1fr 1fr`, o sistema passa a renderizar duas colunas de largura idêntica, conforme ilustrado na Figura 5.11.

Figura 5.11 – Primeiro Contato com a *Grid*.



(a) Tabuleiro no estado padrão.

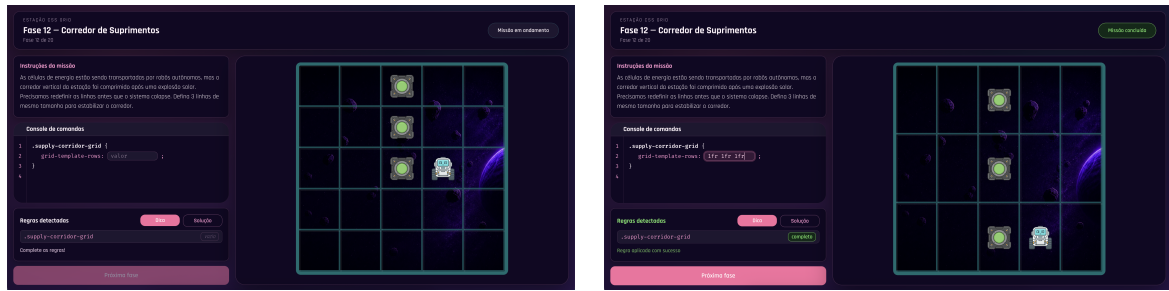
(b) Tabuleiro após regras.

Fonte: própria autora.

5.2.2 Fase 12 — Corredor de Suprimentos

O controle sobre o eixo oposto é explorado na fase seguinte, denominada corredor de suprimentos, cujo objetivo é introduzir a propriedade `grid-template-rows` para a definição explícita das linhas da grade. Conforme apresentado na Figura 5.12, o estado inicial exibe o selector `.supply-corridor-grid` sem parâmetros configurados, fazendo com que as células de energia se concentrem em uma única região do *layout*. Após a inserção da sequência `1fr 1fr 1fr`, são criadas três linhas com proporções equivalentes, promovendo uma distribuição adequada dos elementos na interface.

Figura 5.12 – Corredor de Suprimentos.



(a) Tabuleiro no estado padrão.

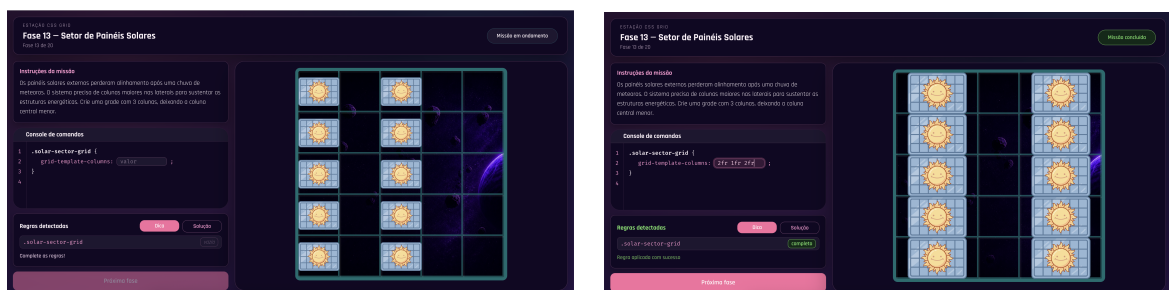
(b) Tabuleiro após regras.

Fonte: própria autora.

5.2.3 Fase 13 — Setor de Painéis Solares

A exploração de proporções assimétricas e do dimensionamento flexível ocorre no setor de painéis solares, ambiente que retoma a utilização da propriedade `grid-template-columns`. Diferentemente das fases anteriores, que empregam divisões uniformes, esta atividade desafia o estudante a configurar colunas com larguras distintas. Como ilustrado na Figura 5.13, a inserção do valor `2fr 1fr 2fr` no seletor `.solar-sector-grid` modifica a distribuição do espaço disponível no *container*, fazendo com que as colunas laterais ocupem o dobro da largura da coluna central.

Figura 5.13 – Setor de Painéis Solares.



(a) Tabuleiro no estado padrão.

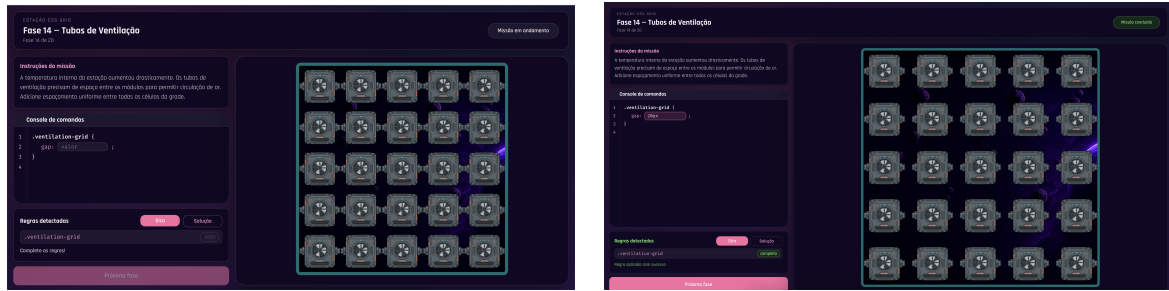
(b) Tabuleiro após regras.

Fonte: própria autora.

5.2.4 Fase 14 — Tubos de Ventilação

A introdução de espaçamentos entre os elementos da grade é realizada no desafio dos tubos de ventilação por meio da propriedade `gap`. Inicialmente, o seletor `.ventilation-grid` não apresenta nenhum valor configurado, fazendo com que os módulos sejam exibidos sem qualquer separação visual, conforme ilustrado na Figura 5.14. Ao definir o valor `20px`, o sistema passa a aplicar espaçamentos uniformes entre linhas e colunas, tornando a disposição dos elementos mais organizada e visualmente clara.

Figura 5.14 – Tubos de Ventilação.



(a) Tabuleiro no estado padrão.

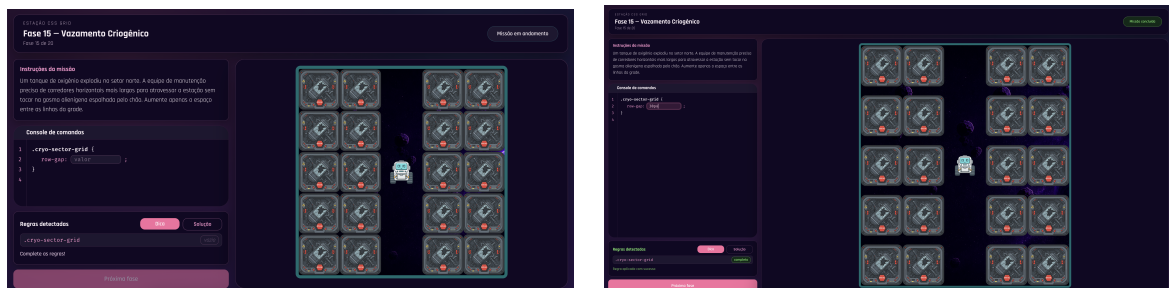
(b) Tabuleiro após regras.

Fonte: própria autora.

5.2.5 Fase 15 — Vazamento Criogênico

A propriedade `row-gap` é apresentada na atividade de vazamento criogênico com o objetivo de demonstrar o controle específico do espaçamento vertical entre as linhas da grade. Conforme mostrado na Figura 5.15, o seletor `.cryo-sector-grid` encontra-se inicialmente sem um valor definido para essa propriedade, fazendo com que os módulos sejam exibidos com pouca separação vertical. Após a inserção do valor 30px, o sistema amplia de forma uniforme a distância entre as linhas, mantendo inalterado o espaçamento horizontal existente entre as colunas.

Figura 5.15 – Vazamento Criogênico.



(a) Tabuleiro no estado padrão.

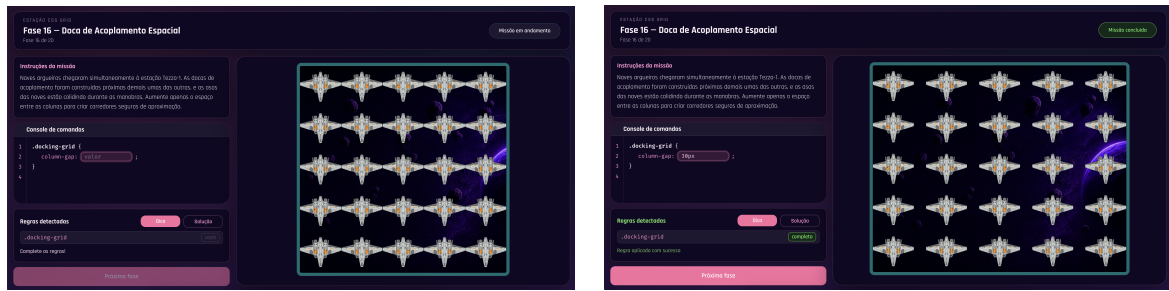
(b) Tabuleiro após regras.

Fonte: própria autora.

5.2.6 Fase 16 — Doca de Acoplamento Espacial

Na sequência, a propriedade `column-gap` é apresentada como mecanismo de controle do espaçamento horizontal entre colunas. No estado inicial, os elementos exibidos pelo seletor `.docking-grid` encontram-se excessivamente próximos lateralmente. Ao definir o valor 30px, a distância entre as colunas é ampliada de maneira homogênea, sem alterar o espaçamento entre linhas, conforme ilustrado na Figura 5.16.

Figura 5.16 – Doca de Acoplamento Espacial.



(a) Tabuleiro no estado padrão.

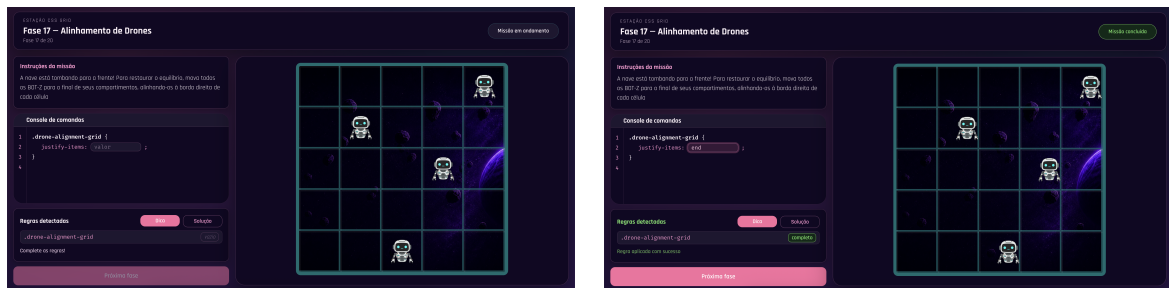
(b) Tabuleiro após regras.

Fonte: própria autora.

5.2.7 Fase 17 — Alinhamento de Drones

O foco passa então para o alinhamento coletivo dos elementos no eixo horizontal por meio do desafio de alinhamento de drones, que introduz a propriedade `justify-items`. A Figura 5.17 demonstra a transformação obtida: inicialmente, o seletor `.drone-alignment-grid` não apresenta configurações de alinhamento, mantendo os drones em suas posições padrão. Ao inserir o valor `end`, todos os elementos são reposicionados simultaneamente junto à borda direita de suas respectivas células.

Figura 5.17 – Alinhamento de Drones.



(a) Tabuleiro no estado padrão.

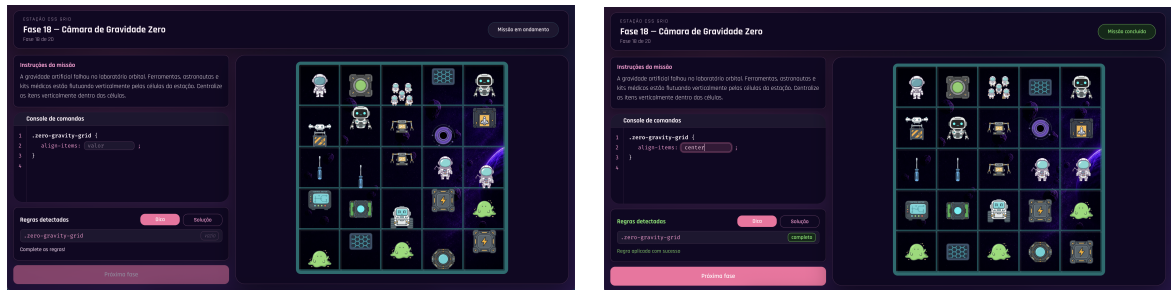
(b) Tabuleiro após regras.

Fonte: própria autora.

5.2.8 Fase 18 — Câmara de Gravidade Zero

O mesmo princípio de alinhamento coletivo é aplicado ao eixo vertical na câmara de gravidade zero, atividade desenvolvida com base na propriedade `align-items`. Inicialmente, os componentes encontram-se distribuídos em alturas distintas dentro das células da grade. Com a definição do valor `center` no seletor `.zero-gravity-grid`, todos os itens passam a ocupar o centro vertical de seus compartimentos, conforme observado na Figura 5.18.

Figura 5.18 – Câmara de Gravidade Zero.



(a) Tabuleiro no estado padrão.

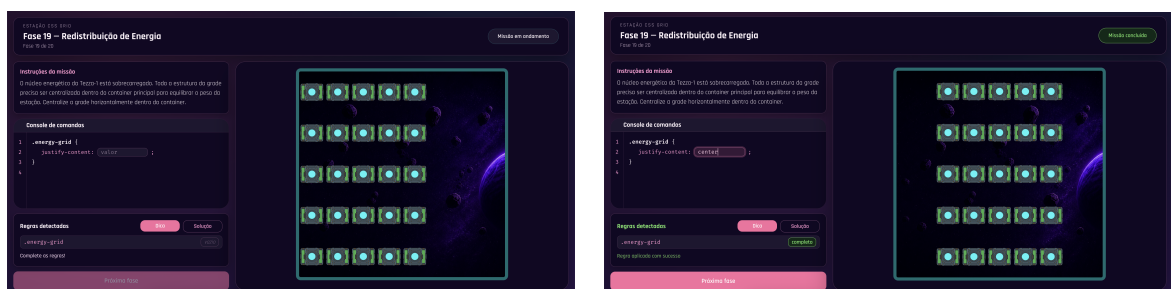
(b) Tabuleiro após regras.

Fonte: própria autora.

5.2.9 Fase 19 — Redistribuição de Energia

A penúltima etapa do módulo aborda a propriedade `justify-content`, responsável pelo alinhamento horizontal da estrutura completa da *grid* dentro do espaço disponível no *container*. Conforme mostrado na Figura 5.19, a *grid* encontra-se inicialmente posicionada à esquerda. Após a inserção do valor `center` no seletor `.energy-grid`, todo o conjunto é deslocado para a região central do contêiner.

Figura 5.19 – Redistribuição de Energia



(a) Tabuleiro no estado padrão.

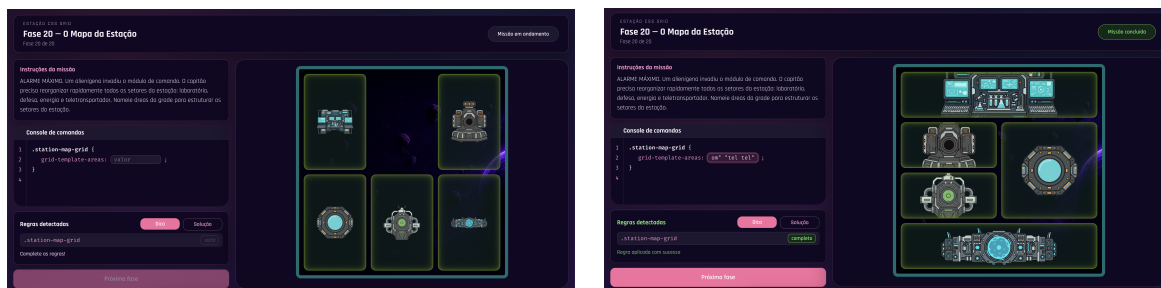
(b) Tabuleiro após regras.

Fonte: própria autora.

5.2.10 Fase 20 — O Mapa da Estação

O encerramento do módulo dedicado ao *Grid Container* ocorre com a introdução da propriedade `grid-template-areas`, utilizada para estruturar o *layout* por meio de regiões nomeadas. No estado inicial, os componentes ocupam células individuais sem uma organização funcional definida. Após a configuração correta das áreas nomeadas no seletor correspondente, os módulos são reorganizados de acordo com o mapeamento especificado, formando uma disposição coerente e concluindo o percurso pedagógico do sistema, conforme ilustrado na Figura 5.20.

Figura 5.20 – O Mapa da Estação.



(a) Tabuleiro no estado padrão.

(b) Tabuleiro após regras.

Fonte: própria autora.

6 Considerações Finais

Este trabalho apresentou o desenvolvimento do *Tezza Space*, plataforma interativa em formato de jogo voltada ao ensino do *CSS Grid*, partindo da premissa de que a aprendizagem de conceitos espaciais e bidimensionais é favorecida quando o usuário pode experimentar, testar hipóteses e observar, de forma imediata, os efeitos visuais das regras aplicadas. A arquitetura implementada sustenta um fluxo completo de interação, no qual as regras definidas no console são processadas, convertidas em estruturas computáveis e refletidas diretamente no tabuleiro orbital, evidenciando a viabilidade técnica da proposta, conforme demonstrado pelos testes automatizados apresentados no [Capítulo 4](#), que confirmaram a correção dos mecanismos de conversão, validação e renderização com ampla cobertura de código.

Retomando os objetivos específicos definidos na introdução deste trabalho, observa-se que os três foram contemplados ao longo do desenvolvimento. A sistematização progressiva dos conceitos fundamentais do *CSS Grid* materializou-se na organização das vinte fases em dois conjuntos didáticos distintos, o primeiro dedicado às propriedades dos itens filhos e o segundo às propriedades do contêiner, conduzindo o usuário do posicionamento elementar até a organização declarativa do *layout* por meio de áreas nomeadas, conforme detalhado individualmente no [Capítulo 5](#). A visualização direta e o *feedback* imediato foram assegurados pelo ciclo contínuo de edição e renderização descrito no [Capítulo 4](#), no qual cada alteração realizada no console reflete-se instantaneamente no tabuleiro orbital, sem exigir qualquer ação adicional do usuário. Por fim, a proposta de uma plataforma interativa inspirada em abordagens gamificadas foi concretizada por meio da ambientação narrativa da estação Tezza-1, na qual personagens recorrentes e situações de urgência conferem sentido prático ao objetivo técnico de cada desafio, enquanto o desenvolvimento dos elementos gráficos e do modelo estrutural das fases, realizado com apoio de Inteligência Artificial Generativa, mostrou-se um recurso eficaz para garantir consistência visual e pedagógica em escala.

Como principal limitação, o projeto ainda não foi submetido a uma validação empírica com usuários reais, o que impede verificar, na prática, se a sequência didática proposta efetivamente favorece a construção gradual do conhecimento sobre *CSS Grid* conforme esperado.

O *Tezza Space* estabelece, assim, uma base técnica e pedagógica consistente para o ensino interativo do *CSS Grid*, sustentada tanto pela cobertura de testes automatizados quanto pela documentação detalhada de cada fase, abrindo espaço para investigações futuras voltadas à validação de sua proposta junto a usuários reais.

6.1 Trabalhos Futuros

Os desdobramentos identificados ao longo do desenvolvimento apontam para diferentes possibilidades de evolução do sistema. Do ponto de vista pedagógico, a realização de estudos de usabilidade com diferentes perfis de usuários, incluindo estudantes iniciantes e desenvolvedores com experiência prévia em *front-end*, representa uma etapa importante para a continuidade do projeto, uma vez que os dados coletados poderiam subsidiar a revisão da progressão das fases, o ajuste do nível de dificuldade e a avaliação da efetividade dos recursos de apoio à resolução, como os botões de dica e solução. A coleta de métricas de interação, incluindo o número de tentativas por fase, o tempo médio de resolução e os padrões de erro mais recorrentes, também forneceria evidências concretas para orientar melhorias futuras.

No âmbito do conteúdo pedagógico, a ampliação do conjunto de fases existentes constitui um desdobramento natural da proposta. Novos desafios poderiam ser desenvolvidos para as propriedades já contempladas, explorando diferentes cenários, narrativas e configurações dos módulos, de modo a oferecer oportunidades adicionais de prática e consolidação dos conceitos. Além disso, a criação de fases avançadas, posicionadas ao final da sequência didática, permitiria desafiar o usuário a combinar simultaneamente diferentes propriedades do [CSS Grid](#) na resolução de um único problema de *layout*, aproximando a experiência das situações encontradas no desenvolvimento *front-end*.

Do ponto de vista técnico, a implementação de mecanismos de persistência de progresso possibilitaria ao usuário retomar a experiência a partir do ponto em que foi interrompida, tornando a plataforma mais adequada a contextos de aprendizagem distribuídos ao longo do tempo. Da mesma forma, a disponibilização de um modo livre, no qual fosse possível experimentar propriedades do [CSS Grid](#) sem as restrições das missões estruturadas, configura uma extensão compatível com os objetivos educacionais do projeto, ampliando as oportunidades de exploração e aprendizado.

Referências

AYYARRAPPAN, M. Creating modular and reusable web components with react. **International Journal Research of Leading Publication (IJLRP)**, v. 2, n. 5, 2021.

BANKS, A.; PORCELLO, E. **Learning React: Modern Patterns for Developing React Apps**. 2. ed. Sebastopol, CA: O'Reilly Media, 2020.

BLANC, J. **Composer avec les technologies du web**. Tese (Doutorado) — Université Paris 8, 2020. Capítulo 2. Disponível em: <<https://phd.julie-blanc.fr/chapitre-02.html>>. Acesso em: 05 jan. 2026.

BODUCH, A.; DERKS, R. **React and React Native: A complete hands-on guide to modern web and mobile development with React.js**. 3. ed. Birmingham: Packt Publishing, 2020.

BOS, B.; LIE, H. W. **Cascading Style Sheets, Designing for the Web**. [S.l.]: Addison-Wesley, 1997.

CODEPEN. **CSS Grid Layout Quick Start**. 2025. Acessado em: 13 jan. 2026. Disponível em: <<https://codepen.io/var-bin/post/css-grid-layout-quick-start>>.

COULOURIS, G. *et al.* **Distributed Systems: Concepts and Design**. 5. ed. Harlow: Addison-Wesley, 2012.

COYIER, C. **Fighting The Space Between Inline Block Elements**. 2016. CSS-Tricks. Disponível em: <<https://css-tricks.com/fighting-the-space-between-inline-block-elements/>>. Disponível em: <<https://css-tricks.com/fighting-the-space-between-inline-block-elements/>>.

CSS Grid Garden. **CSS Grid Garden**. 2025. Acesso em: jan. 2026. Disponível em: <<https://cssgridgarden.com/>>.

Devographics. **State of CSS 2025: Features**. 2025. Acesso em: jul. 2026. Disponível em: <<https://2025.stateofcss.com/en-US/features/>>.

DOAN, P. **Responsive Web Design Techniques and Challenges**. Trabalho de Conclusão de Curso, 2017. Acesso em: jan. 2026. Disponível em: <https://www.theseus.fi/bitstream/handle/10024/124419/Doan_Phuc.pdf>.

ECMA International. **ECMAScript Language Specification**. 2023. <<https://tc39.es/ecma262/>>. Acesso em: 11 fev. 2026.

FILHO, L. K. A.; NEVES, J. L. F.; JÚNIOR, W. P. Trabalho de Conclusão de Curso, **Lógica-Mente: Um jogo de lógica proposicional**. Inhumas: [s.n.], 2023. Orientador: Prof. Me. Ricardo Rodrigues Dias de Lima. Coorientadora: Profa. Me. Danila Fernandes Mendonça.

Flexbox Froggy. **Flexbox Froggy**. 2025. Acesso em: jan. 2026. Disponível em: <<https://flexboxfroggy.com/>>.

HENKLAIN, M. H. O.; CARVALHO, L. S. G. d.; FEITOSA, E. L. Dificuldades no aprendizado de programação: Um exame baseado em mapeamento sistemático da literatura e interpretação analítico-comportamental. **Revista Brasileira de Informática na Educação**, v. 33, p. 1484–1521, 2025.

JENSEN, S. H.; MØLLER, A.; THIEMANN, P. Type analysis for javascript. In: **Proc. European Conference on Object-Oriented Programming**. [S.l.]: Springer, 2009.

KOLB, D. A. **Experiential Learning: Experience as the Source of Learning and Development**. Englewood Cliffs, NJ: Prentice Hall, 1984.

KUMAR, T. **Fluent React: Build Fast, Performant, and Intuitive Web Applications**. Sebastopol, CA: O'Reilly Media, 2024.

LAURSEN, M. R.; XU, W.; MØLLER, A. Reducing static analysis unsoundness with approximate interpretation. **Proceedings of the ACM on Programming Languages**, v. 8, n. PLDI, p. 1165–1188, 2024.

MAGALHÃES, R. A. P. Trabalho de Conclusão de Curso, **Desenvolvimento de uma aplicação web educativa com gamificação: um estudo sobre mobile learning**. Castanhal, Pará: [s.n.], 2018. Orientador: Prof. Dr. Marcos César da Rocha Seruffo. Coorientador: Prof. Paulo Sérgio Brito Pinheiro.

MAYER, R. E. **Multimedia Learning**. 2. ed. Cambridge: Cambridge University Press, 2009.

MDN Web Docs. **Basic Concepts of CSS Grid Layout**. 2025. Acessado em: 13 jan. 2026. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/CSS/Guides/Grid_layout/Basic_concepts>.

MEYER, E. A.; WEYL, E. **Grid Layout in CSS: Interface Layout for the Web**. Sebastopol, CA: O'Reilly Media, 2017.

MICROSOFT. **TypeScript Handbook - From Scratch**. 2026. <<https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html>>. Acesso em: 19 jan. 2026.

MOREIRA, G. L. Monografia (Graduação), **CS Arcade: uma proposta de intervenção gamificada para incentivo à leitura no curso de Bacharelado em Ciência da Computação da Universidade Federal de Ouro Preto**. Ouro Preto, MG: [s.n.], 2022. Orientador: Reinaldo Silva Fortes. Coorientador: Elton José Silva.

Mozilla. **justify-self**. 2024. Acesso em: 13 jan. 2026. Disponível em: <<https://developer.mozilla.org/en-US/docs/Web/CSS/Reference/Properties/justify-self>>.

MOZILLA. **Block and inline layout in normal flow**. 2025. MDN Web Docs. Acesso em: 5 jan. 2026. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/CSS/Guides/Display/Block_and_inline_layout>.

MOZILLA. **Conceitos básicos de Grid Layout**. 2025. MDN Web Docs, disponível em <https://developer.mozilla.org/pt-BR/docs/Web/CSS/Guides/Grid_layout>. Acesso em: 13 jan. 2026.

MOZILLA. **CSS layout: Flexbox**. 2025. MDN Web Docs. Acesso em: 12 jan. 2026. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Learn_web_development/Core/CSS_layout/Flexbox>.

MOZILLA. **Elements participating in a block formatting context**. 2025. MDN Web Docs. Acesso em: 12 jan. 2026. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/CSS/Guides/Display/Block_formatting_context>.

Mozilla Developer Network. **Introduction to server-side programming**. 2025. Acesso em: fev. 2026. Disponível em: <https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Server-side/First_steps/Introduction>.

Mozilla Developer Network. **Viewport concepts**. 2025. <https://developer.mozilla.org/en-US/docs/Web/CSS/Viewport_concepts>. Acesso em: 11 fev. 2026.

Mozilla Developer Network. **HTML: Linguagem de Marcação de Hipertexto**. 2026. Acesso em: 12 fev. 2026. Disponível em: <<https://developer.mozilla.org/pt-BR/docs/Web/HTML>>.

NETWORK, M. D. **What is JavaScript?** 2026. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Learn_web_development/Core/Scripting/What_is_JavaScript>.

PHAM, Q. **The Evolution of CSS: From CSS2 to Flexbox**. 2025. Disponível em: <https://www.theseus.fi/bitstream/handle/10024/891511/Pham_Quang.pdf>. Disponível em: <https://www.theseus.fi/bitstream/handle/10024/891511/Pham_Quang.pdf>.

React. **React Documentation**. 2025. Acesso em: 10 jan. 2026. Disponível em: <<https://react.dev/learn>>.

RUVALCABA, Z.; BOEHM, A. **Murach's HTML and CSS**. 5. ed. Fresno: Mike Murach & Associates, 2021.

W3C. **Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification**. 1998. World Wide Web Consortium. Atualizado em 2018. Acesso em: 12 jan. 2026. Disponível em: <<https://www.w3.org/TR/CSS2/>>.

W3C. **CSS Flexible Box Layout Module Level 1**. 2018. World Wide Web Consortium. Acesso em: 12 jan. 2026. Disponível em: <<https://www.w3.org/TR/css-flexbox-1/#overview>>.

WHATWG. **DOM Standard**. 2026. Acesso em: jul. 2026. Disponível em: <<https://dom.spec.whatwg.org/>>.

WHATWG. **HTML Living Standard**. 2026. Acesso em: jul. 2026. Disponível em: <<https://html.spec.whatwg.org/>>.

WIERUCH, R. **The Road to React: The React.js 19 with Hooks in JavaScript Book**. 2025. ed. Leanpub, 2025. Acesso em: jul. 2026. Disponível em: <<https://www.roadtoreact.com/>>.

World Wide Web Consortium. **Cascading Style Sheets (CSS) Specification**. 2018. <<https://www.w3.org/TR/CSS/>>. Acesso em: 11 fev. 2026.

World Wide Web Consortium (W3C). **Cascading Style Sheets, level 1**. 1996. W3C Recommendation, 17 dez. 1996. Disponível em: <<https://www.w3.org/TR/CSS1/>>. Acesso em: 05 jan. 2026.

Declaração

Durante a preparação deste documento, eu, Sthéphany Karoline Soares de Araújo Tezza, estudante de graduação do curso de Bacharelado em Ciência da Computação, declaro o uso de IAGen neste trabalho. Após o uso desta ferramenta, revisei e editei integralmente o conteúdo em conformidade com os princípios éticos para uso de IAGen (Resolução CONPEP 144) e com os acordos estabelecidos com a pessoa orientadora da pesquisa. Dessa forma, assumo total responsabilidade pelo conteúdo da publicação.

A IAGen foi utilizada nas seguintes etapas:

- Etapa 1: **Reorganização textual e tradução.** Nesta etapa, a ferramenta de IAGen foi utilizada para reorganizar o texto do resumo e adequá-lo às normas acadêmicas, bem como para realizar a tradução do resumo para o idioma inglês. O modelo utilizado foi o ChatGPT (OpenAI), versão GPT-4. O *prompt* utilizado foi:

“Reescreva o resumo do trabalho de forma adequada às normas acadêmicas, mantendo um único parágrafo, e em seguida gere a versão em inglês.”

- Etapa 2: **Reescrita e refinamento de textos acadêmicos.** Nesta etapa, a ferramenta de IAGen foi utilizada para reescrever e aprimorar a redação dos textos referentes aos trabalhos relacionados *Flexbox Froggy* e *CSS Grid Garden*, com foco na melhoria da coesão textual, clareza conceitual e adequação à linguagem acadêmica, sem alteração do conteúdo técnico ou das referências originais. O modelo utilizado foi o ChatGPT (OpenAI), versão GPT-4. O *prompt* utilizado foi:

“Reescreva o texto de forma acadêmica, clara e coesa, mantendo o conteúdo técnico original.”

- Etapa 3: **Geração da biblioteca de *assets* gráficos.** Nesta etapa, a ferramenta de IAGen foi utilizada como apoio à criação visual dos elementos gráficos do sistema *Tezza Space*, incluindo personagens, módulos, equipamentos, estruturas e objetos distribuídos ao longo das fases, a partir de um *prompt*-base estruturado responsável por garantir uniformidade estética entre os ativos gerados. O modelo utilizado foi o Nano Banana 2.0, em conjunto com a plataforma Figma. O *prompt*-base utilizado seguiu a estrutura:

“[Bloco de estilo: ilustração vetorial bidimensional plana, sem gradientes, texturas ou sombras] + [Bloco de traço: contornos de espessura média e consistente] + [Bloco de cor: paleta restrita a cinza, azul, laranja e roxo] + [Bloco de grade: elemento centralizado em bloco quadrado, com margens uniformes] + [Bloco de

vista: perspectiva frontal ortográfica, sem isometria] + [Bloco de fundo: cor plana única] + [Bloco de objeto: descrição específica do ativo desejado].”

- Etapa 4: **Estruturação de fases e progressão didática.** Nesta etapa, a ferramenta de IAGen foi utilizada na elaboração de um modelo estrutural de fase, capaz de organizar de forma sistemática o nível de aprendizado, a narrativa imersiva, a instrução objetiva, o conceito de *Cascading Style Sheets (CSS)* associado e o código de solução de cada missão, servindo como referência para a construção das vinte fases do sistema. O modelo utilizado foi o ChatGPT (OpenAI), versão GPT-4. O *prompt* utilizado foi:

“Elabore um modelo estrutural de fase para um jogo educativo de CSS Grid, contemplando progressão gradual dos conteúdos, contextualização narrativa, definição explícita do conceito abordado e critério objetivo de validação.”