Chapter 2

Desenvolvimento de Experiência Digitais Imersivas através do Mapeapento de Vídeo em Tempo Real com Redes Neurais

Héder Pereira Rodrigues Silva, Iallen Gábio de Sousa Santos

Abstract

Gesture-based interactivity, powered by technologies such as Kinect and computer vision systems, allows users to control digital content without touch, making it useful for applications like interactive storefronts and public games. To enable this, computer vision algorithms and neural networks detect and interpret body movements in real time, often accelerated by WebGL. Tools like ml5.js, Matter.js, and P5.js facilitate the development of these web applications, enabling the creation of immersive and accessible experiences with gesture recognition, physics simulation, and integrated visual rendering.

Resumo

A interatividade com gestos permite que usuários controlem conteúdos digitais sem toque, tornando-se útil em aplicações como vitrines interativas e jogos públicos. Para viabilizar isso, algoritmos de visão computacional e redes neurais detectam e interpretam movimentos corporais em tempo real, geralmente com aceleração via WebGL. Ferramentas como ml5.js, Matter.js e P5.js facilitam o desenvolvimento web dessas aplicações, permitindo a criação de experiências imersivas acessíveis, com reconhecimento de gestos, simulação física e visualização gráfica integrada. Neste capítulo é apresentado um panorama geral sobre a aplicação deste tipo de tecnologia na construção de experiências digitais imersivas.

2.1. Introdução

Desde a década de 1980, a utilização de *Digital Signage* - ou Sinalização Digital - vem sendo priorizada em relação aos avisos tradicionais para a divulgação de marcas empresariais, pois ela os supera ao facilitar atualizações frequentes e oportunas, aumentando a precisão e permitindo o fornecimento de informações altamente dinâmicas que, de outra forma, não seriam disponibilizadas [Davies et al. 2022].

Estudos indicam que a sinalização digital pode aumentar significativamente o engajamento do público e influenciar positivamente as decisões de compra, destacando-se como uma ferramenta poderosa no arsenal do marketing moderno [Asif et al. 2024]. Na atualidade, essa tecnologia evoluiu para uma abordagem que tem ganhado destaque: o uso de sinalização digital interativa, como monitores, totens e painéis de LED, amplamente utilizados em ambientes como shoppings, ruas comerciais e supermercados (Figura 2.1). Essa interatividade vem sendo implementada por meio de telas *touchscreen* e/ou QR Codes, porém já existem formas de interação mais imersivas, como a utilização de rastreamento ocular [Chung 2017] e leitura de gestos de mão [Chen et al. 2009].

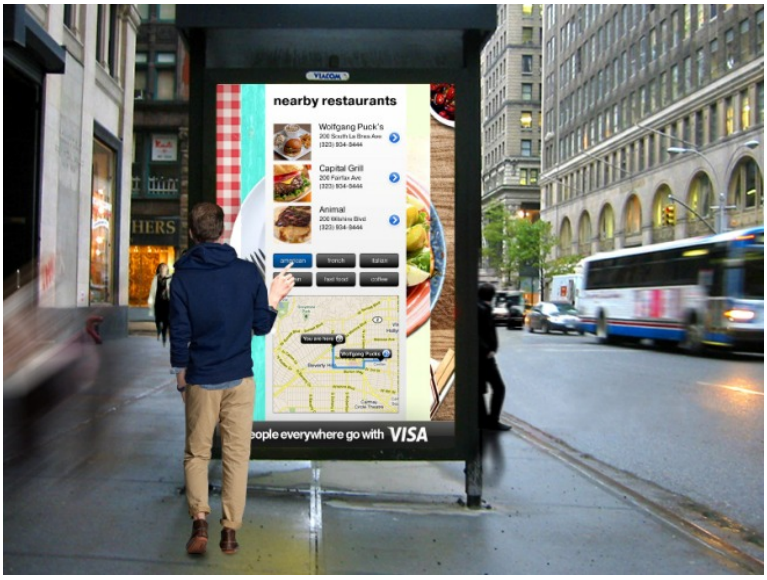

Figure 2.1. Representação de um Totem Digital com uma aplicação interativa. [Digital 2025]

O desenvolvimento dessas aplicações imersivas de sinalização digital atualmente é possível devido à utilização de algoritmos de aprendizado de máquina - para que o sistema de reconhecimento de gestos atenda aos requisitos em termos de precisão, desempenho em tempo real e robustez [Chen et al. 2009] - juntamente com ferramentas como o WebGL [Parisi 2012] que permitem a renderização de gráficos 3D interativos diretamente nos navegadores, sem a necessidade de plugins adicionais. Um exemplo de plataforma que utiliza o WebGL para o desenvolvimento de softwares de *digital signage* é a BrightSign [BrightSign LLC 2025].

Além disso, o desenvolvimento de aplicações interativas se tornou mais acessível para desenvolvedores menos experientes por meio de bibliotecas de JavaScript que facilitam a criação de gráficos, animações, objetos com propriedades físicas e uso de ferramentas de IA.

Diante desse panorama, este capítulo tem como objetivo apresentar as diferentes maneiras de se utilizar as aplicações interativas, introduzir ao leitor conceitos de *Machine Learning*, Visão Computacional e *design* de aplicações comerciais. Neste contexto, introduzir as bibliotecas **Ml5.js**, **Matter.js** e **P5.js** que serão utilizadas em um estudo de caso de criação de uma Experiência Imersiva utilizando o Mapeamento de posições de mão

para a interação com o usuário.

As próximas seções estão organizadas da seguinte forma:

Seção 2 Apresenta a evolução e os fundamentos da interação gestual em sinalização digital, abordando desde tecnologias pioneiras até exemplos práticos de aplicação.

Seção 3 Discute a arquitetura e os requisitos de software para sistemas de digital signage interativo, incluindo criação de conteúdo, agendamento, display e dispositivos móveis.

Seção 4 Introduz o conjunto de bibliotecas ml5.js, Matter.js e P5.js, mostrando como elas facilitam o reconhecimento de gestos, a simulação física e a renderização gráfica em aplicações web.

Seção 5 Demonstra, na forma de um tutorial prático, a integração das três bibliotecas para criar um jogo interativo baseado em gestos, desde a configuração do ambiente até a lógica de vitória.

2.2. Aplicações Interativas no contexto da sinalização digital

A interação com o público tem sido explorada no contexto de telas públicas há muitos anos. De fato, o trabalho com telas sensíveis ao toque começou na década de 1960 e tem continuado em ritmo acelerado desde então [Davies et al. 2022].

Uma explicação para a eficiência dessa estratégia pode ser explicada pelo fenômeno chamado *Honeypot Effect*(Efeito Pote de Mel) [Brignull and Rogers 2003], que descreve o efeito de atração que as pessoas sentem ao ver outras interagindo com um dispositivo de sinalização interativo, como pode ser observado na Figura 2.2.

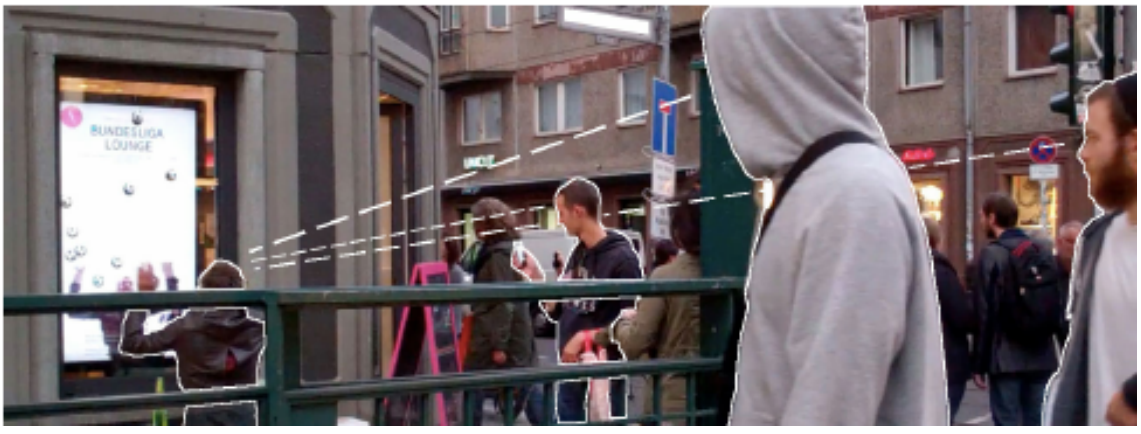

Figure 2.2. "O Efeito Pote de Mel: Quando as pessoas percebem alguém fazendo gestos incomuns, elas se posicionam de forma que tanto a tela quanto a pessoa que está interagindo sejam vistas. Muitas vezes, elas também se posicionam de forma que não sejam representadas na tela". [Müller et al. 2012]

Este fenômeno ocorre principalmente em locais públicos: um indivíduo, ao observar outra pessoa interagindo com uma tela publicitária, tem mais chances de ter a sua

atenção desviada para o local e se dirigir para mais perto da sinalização digital. Esse efeito continua e aumenta quanto mais pessoas estiverem próximas do dispositivo, tornando-o eficiente para chamar a atenção do público para um anúncio.

Um exemplo claro do efeito "pote de mel" pode ser observado no *CityWall* [HIIT 2025], um painel multitouch instalado no centro de Helsinque, capital da Finlândia. Peltonen et al. verificaram que, quando alguém já está interagindo com a tela, cerca de 19% dos passantes a percebem; em contrapartida, na ausência de um usuário ativo, alguns pedestres não identificam sua capacidade interativa [Peltonen et al. 2008].

2.2.1. Interatividade com Toque

Por muito tempo a interatividade nas aplicações interativas foi implementada por meio de toque. Um exemplo recente disso são os Quiosques de autoatendimento: são painéis digitais sensíveis ao toque instalados em pontos de venda ou restaurantes, permitindo que o cliente faça pedidos, escolha opções e pague sem auxílio de um atendente. Por exemplo, o McDonald's introduziu quiosques touchscreen em suas lojas, onde o cliente visualiza o cardápio, personaliza itens e confirma o pedido na própria tela, reduzindo erros e filas [McDonald's Corporation 2020].

Outra aplicação dessa forma de interatividade são estações interativas de personalização de produtos em *showrooms*. Por exemplo, numa concessionária Seat [SEAT S.A. 2021], foram criados pedestais com telas touchscreen ao lado de cada carro, onde o visitante pode "brincar" de configurador: escolher cores, acessórios e até registrar um depósito no próprio painel.

2.2.2. Interatividade com Gestos

Com o surgimento de tecnologias de visão computacional como o *Kinect* [Microsoft 2010], a utilização de interação com gestos começou a ser utilizada como alternativa ao toque, em situações em que a liberdade de movimento é mais conveniente ou permite interações que não são possíveis apenas por meio de *touchscreens*.

Um exemplo de aplicação utilizada publicamente é o projeto *StrikeAPose* [Walter et al. 2013], que criou um jogo baseado em simulação física para motivar os transeuntes a interagir. Os passantes viam sua imagem espelhada na tela e podiam interagir com ela para manipular cubos virtuais. Ao posicionar os cubos em áreas-alvo específicas, eles acumulavam pontos. Este trabalho foi utilizado para explorar formas de ensinar novos gestos de interação aos usuários.

Outras formas de explorar a imersividade da interatividade com gestos são apontadas pela OnSign TV [TV 2023]: uma vitrine pode exibir um banner "chegue mais perto" e, ao detectar uma pessoa dentro de um raio definido (por sensor de movimento), mudar automaticamente o layout para mostrar informações detalhadas ou opções interativas. Por exemplo, ao detectar um cliente perto da tela, o sistema pode exibir especificações do produto em destaque ou ativar um menu de toque.

Após explorar as aplicações e o potencial do *Digital Signage* interativo, a seção seguinte se dedicará a detalhar os processos de implementação dessas soluções, bem como os principais recursos de software que viabilizam a criação de experiências imersivas.

2.3. Como é desenvolvida uma sinalização imersiva?

Um erro frequente é subestimar a complexidade dos sistemas de digital signage interativos [Davies et al. 2022]. Esses ambientes exigem um conjunto robusto de funcionalidades, que vão desde a criação e publicação de conteúdo até o gerenciamento remoto de dispositivos, passando pelo agendamento automatizado de exibições e pelo monitoramento em tempo real da interação dos usuários.

[Davies et al. 2022] representam a arquitetura de softwares de sinalização em 4 segmentos: Criação de conteúdo, Agendamento, Display e Dispositivos móveis - representados na Figura 2.3. Dado o contexto e o foco nas aplicações interativas, todo o conteúdo deste capítulo estará incluído nos segmentos de **Criação de conteúdo** e de **Display**. Estes segmentos são, respectivamente, responsáveis por fornecer as ferramentas necessárias para a criação do conteúdo que irá ser exibido e pelas operações relacionadas à interatividade.

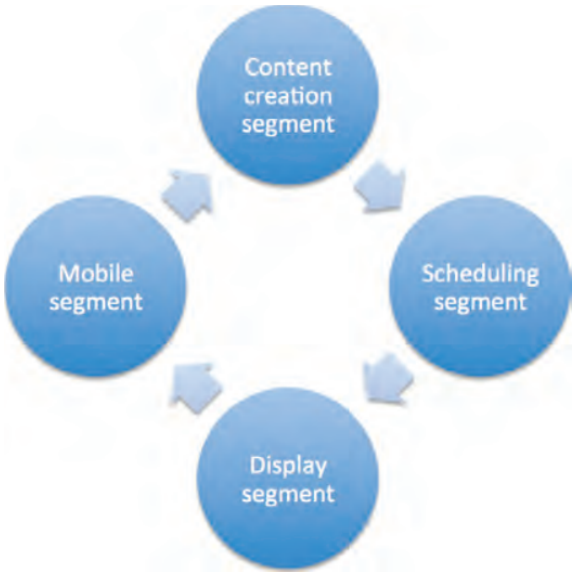

Figure 2.3. Entendendo a arquitetura de sistemas de sinalização. [Davies et al. 2022]

2.3.1. Criação de Conteúdo

Tradicionalmente, os sistemas consistiam em um software dedicado instalado em computadores conectados ao display ou diretamente incorporados neles. Além disso, também precisava-se de um software associado para a criação e gerenciamento do conteúdo a ser exibido [Davies et al. 2022].

Com o avanço da Web, especialmente em relação à reprodução de vídeos em alta qualidade e à capacidade de renderizar aplicações complexas diretamente no navegador, os sistemas de sinalização digital passaram a adotar um novo modelo: os displays agora executam navegadores que renderizam a aplicação localmente ou se conectam a um servidor dedicado para obter o conteúdo e a lógica do sistema.

Dessa forma, na atualidade, a produção de conteúdo para sinalização digital assemelha-se à de aplicações web. Por isso, não são necessárias ferramentas específicas para esse fim: os desenvolvedores costumam empregar soluções genéricas para criar

imagens, sons, vídeos, páginas e interações. As ferramentas que serão demonstradas neste capítulo serão aprofundadas na Seção 2.4.

2.3.2. Display

O conteúdo que é produzido precisa ser devidamente exibido na tela de sinalização; além disso, as interações que a aplicação irá utilizar devem ser gerenciadas, para que elas ocorram de forma precisa. Essa é a função do segmento de display, ele se encarrega de como funcionará a exibição do conteúdo.

No caso abordado na seção anterior, em que o display se conecta a um servidor dedicado, essa exibição será simplesmente a execução de um navegador, porém, em sistemas onde os nós de exibição têm mais autonomia local, algumas decisões de agendamento e exibição podem ser realizadas localmente, por exemplo, respondendo a eventos de sensores locais, e o conteúdo pode ser obtido de um cache local [Davies et al. 2022].

Esse segmento também oferece suporte à interatividade com o usuário; como visto na Seção 2.2, ela ocorre principalmente de duas formas: via toque em telas touch — cujo input é capturado como clique de mouse e repassado à aplicação — ou via gestos, em que poses corporais, manuais ou faciais são mapeadas e convertidas em comandos específicos para o sistema. Com o fim de se aprofundar nas experiências imersivas, trataremos com mais detalhes de como é feito esse processo de mapeamento e leitura de gestos.

A interação por gestos em sistemas de sinalização digital depende fundamentalmente de visão computacional. Esse campo da inteligência artificial surgiu nos anos 1960 (com trabalhos pioneiros como o de Lawrence Roberts no MIT em 1963 [de Oliveira and de Melo 2022]) e visa ensinar máquinas a interpretar imagens e vídeos do mesmo modo que o olho humano. Em termos simples, define-se visão computacional como um sistema computacional treinado para captar e analisar imagens do mundo real por meio do reconhecimento de padrões.

Na prática, algoritmos de visão computacional extraem características visuais de quadros de vídeo ou fotos (como contornos, texturas ou regiões de interesse) e as usam para tarefas de reconhecimento. Por exemplo, a estimativa de pose identifica pontos-chave do corpo humano (juntas, mãos, etc.) em imagens para entender a postura e o movimento.

No contexto de gestos, uma câmera captura sequências de quadros com o usuário realizando movimentos; algoritmos de detecção (baseados em segmentação, cor de pele ou filtros especializados) processam cada quadro, identificam as regiões correspondentes às mãos ou partes do corpo e calculam suas coordenadas espaciais (x,y em cada frame), como pode ser observado na Figura 2.4. Assim, gestos visuais são convertidos em pontos rastreáveis ou vetores de posição que podem ser usados em aplicações interativas. Essa transformação de vídeo em coordenadas de cursor ou em variáveis de controle é viabilizada justamente pelos métodos de visão computacional implantados no software de sinalização.

Para interpretar corretamente esses padrões visuais, o uso de aprendizado de máquina torna-se essencial. Redes neurais profundas são treinadas para reconhecer gestos a partir de grandes conjuntos de imagens etiquetadas, dispensando regras manuais de extração de características. Durante o treinamento, essas redes aprendem hierarquias de recursos (de bordas simples até formas complexas) que distinguem diferentes gestos. Como

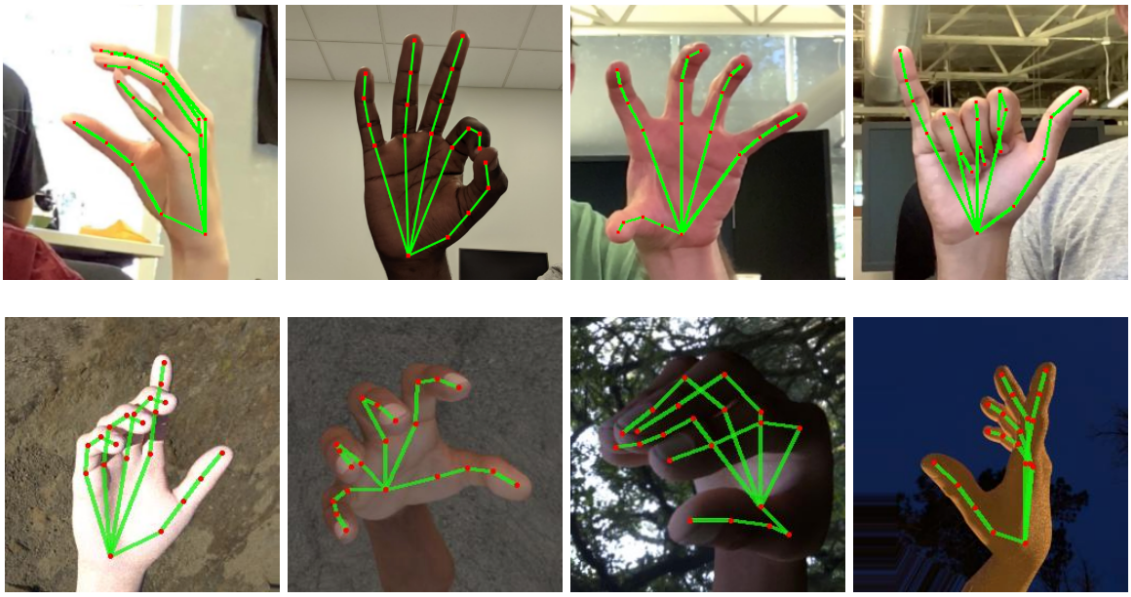

Figure 2.4. Mapeamento de gestos de mão realizado por um algoritmo de visão computacional. [Coldewey 2019]

observado por IBM, redes neurais convolucionais são computacionalmente intensivas e normalmente exigem GPUs para treinar modelos de visão computacional eficientemente [IBM Brasil 2023]. Mesmo para inferência em tempo real, deve-se contar com capacidade de processamento considerável, pois a rede precisa processar dezenas de frames por segundo. Em resumo, o aprendizado de máquina fornece os modelos que detectam e classificam gestos, mas demanda alto poder computacional (memória e processamento) tanto no treinamento quanto, em certa medida, na execução desses modelos.

Uma solução para prover capacidade computacional no ambiente web é o uso de WebGL [Parisi 2012]. WebGL é uma API JavaScript que expõe a GPU do dispositivo via contexto de desenho no navegador. Inicialmente criado para gráficos 3D, o WebGL vem sendo usado por bibliotecas de Machine Learning (ML) em JavaScript para computação numérica pesada - alguns exemplos são o **Tensorflow.js** [TensorFlow.js Team 2023], o **ml5.js** [ml5.js contributors 2018] e o **ONNX.js** [Microsoft 2019]. Na prática, tensores (arranjos multidimensionais de dados) são armazenados como texturas na GPU e operações matemáticas (como multiplicação de matrizes) são implementadas como shaders em WebGL.

Dessa forma, o processamento de redes neurais que rodariam originalmente na CPU passa a ser acelerado pela GPU, que é otimizada para cálculos paralelos. Segundo a documentação do TensorFlow.js, o backend WebGL pode ser até cem vezes mais rápido que a implementação em CPU, uma vez que delega os cálculos aos recursos gráficos.

Um exemplo prático desse conceito é o **TensorFlow.js** [TensorFlow.js Team 2023]. Essa biblioteca de código aberto do Google permite treinar e executar modelos de aprendizado de máquina inteiramente em JavaScript, tanto no navegador quanto em Node.js. Em seu modo de uso no browser, o TensorFlow.js seleciona automaticamente o backend WebGL: ele armazena tensores como texturas de GPU e compila operações tensorais em

shaders acelerados. Isso acelera significativamente tarefas de inferência de redes neurais (como reconhecimento de gestos ou classificação de imagens) sem depender de servidores externos. Em suma, essa ferramenta demonstra como técnicas de aprendizado de máquina podem ser integradas em aplicações web interativas, aproveitando os recursos gráficos do dispositivo cliente para obter desempenho.

Essa abordagem baseada em navegador traz vantagens específicas para sinalização digital interativa. Como navegadores web são universais e baseados em padrões (HTML, CSS, JavaScript), não há necessidade de software proprietário ou hardware dedicado, o que simplifica a escalabilidade e a manutenção do sistema. Conteúdo e lógica podem ser atualizados remotamente: por exemplo, um servidor central pode distribuir scripts e modelos de ML para múltiplos players. Além disso, atualmente, a sinalização digital, por meio da web, integra-se naturalmente aos sistemas existentes (APIs, bancos de dados, serviços em nuvem), permitindo criar conteúdo dinâmico nas páginas de sinalização.

Em termos práticos, essa arquitetura reduz custos operacionais (evita deslocamentos para atualizar cada dispositivo fisicamente), facilita a manutenção (por usar frameworks web comuns) e garante que o sistema seja escalável e compatível com o ecossistema web do ambiente corporativo.

2.4. Desenvolvimento de Aplicações Interativas Simplificado

Na seção anterior, foi discutido como aplicações com reconhecimento de gestos podem ser desenvolvidas para a web e utilizadas em contextos de sinalização digital. No entanto, esse tipo de desenvolvimento pode exigir um conhecimento mais aprofundado, especialmente em áreas como aprendizado de máquina e visão computacional. Considerando essa complexidade, esta seção apresentará três bibliotecas que se complementam e cuja simplicidade possibilita que desenvolvedores iniciantes — ou com pouca experiência em softwares de sinalização digital — consigam criar conteúdos interativos e imersivos de forma acessível.

Essas bibliotecas são: a **ml5.js**, a *Matter.js* e a *P5.js* — aqui referidas como o conjunto MMP — compartilham o propósito de oferecer, em suas áreas de especialização, ferramentas acessíveis e de fácil compreensão para desenvolvedores iniciantes. Com um conhecimento introdutório sobre o MMP, torna-se viável construir aplicações interativas simples, mas eficazes. Em poucos passos, é possível implementar reconhecimento de gestos, simulações físicas com objetos manipuláveis por movimentos corporais e a renderização visual desses elementos com o design definido pelo desenvolvedor. Esses três componentes formam uma base completa para a criação de experiências digitais imersivas e responsivas.

2.4.1. Ml5.js

Na Seção 2.3.2 o **TensorFlow.js** foi apresentado como uma ferramenta capaz de implementar modelos de machine learning para o ambiente web. No entanto, seu uso costuma exigir conhecimentos em álgebra linear, estatística e arquiteturas de redes neurais [Shiffman et al. 2018]. Assim, o **Ml5.js** surge como uma camada de abstração amigável sobre o TensorFlow.js para simplificar o uso de aprendizado de máquina em aplicações web. Ele foi projetado para tornar o ML acessível a artistas, estudantes e desenvolvedores não espe-

cialistas, oferecendo uma interface de alto nível com poucas linhas de código. Assim, em vez de manipular diretamente tensores e operações matemáticas, o programador carrega os modelos pré-treinados por meio de chamadas simples, permitindo focar no aspecto criativo do projeto, ainda com a possibilidade de realizar novos treinamentos para os modelos.

Entre os modelos disponibilizados pelo ml5.js, destacam-se especialmente aqueles voltados à visão computacional interativa. A biblioteca integra classes como HandPose, Pose/BodyPose (às vezes chamada de PoseDetection) e FaceMesh, correspondendo a modelos do MediaPipe/TensorFlow para rastreamento de mãos, detecção de pose corporal e de pontos faciais, respectivamente. Por exemplo, o modelo PoseNet (usado no BodyPose) estima a pose humana identificando as posições de pontos-chave do corpo (como ombros, cotovelos, quadris) em uma imagem, como pode ser visto na Figura 2.5.

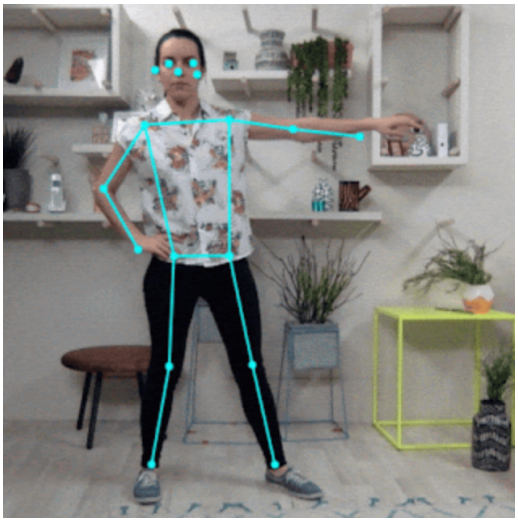

Figure 2.5. Mapeamento de pose de corpo por meio de pontos feito pelo modelo PoseNet, base para o BodyPose [TensorFlow.js Team 2023]

O HandPose realiza, em tempo real, a detecção de uma mão inteira: primeiro, um detector de palma localiza a mão, depois um modelo de landmarks calcula 21 pontos-chave nos dedos e na palma da mão (Figura 2.6).

E O FaceMesh estima 468 pontos de referência faciais em 3D, mapeando detalhadamente a geometria completa do rosto (olhos, nariz, boca e contornos). Cada uma dessas classes carrega internamente o modelo pré-treinado correspondente e expõe funções simples (como .predict() ou callbacks) que retornam as coordenadas dos landmarks detectados a cada frame de vídeo. Na prática, os modelos de visão do ml5.js operam a partir de um fluxo de vídeo (por exemplo, da webcam). A cada quadro, o modelo processa a imagem e devolve listas de coordenadas (normalmente normalizadas) dos pontos detectados. Essas coordenadas podem ser usadas para reconhecer gestos e posturas: por exemplo, identificando quando a mão de um usuário cruza determinada região da imagem ou se um usuário levantou um braço. Esse processamento em tempo real torna possível criar sinais digitais interativos. Usando FaceMesh, por exemplo, é possível inferir expressões faciais ou a direção do olhar de um espectador, enquanto HandPose facilita reconhecer gestos (como acenos ou pinças no ar).

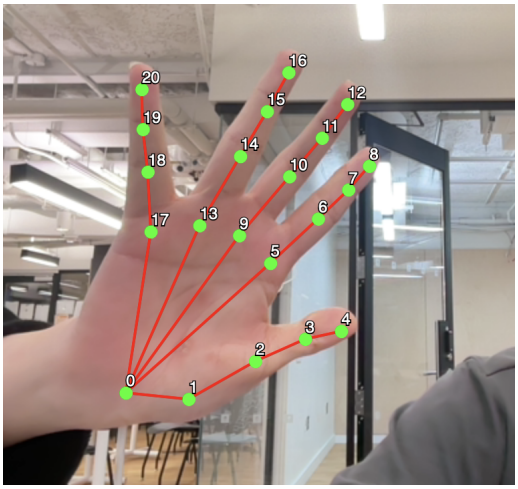

Figure 2.6. Mapeamento de pose de mão por meio de pontos feito pelo modelo Handpose [ml5.js contributors 2018].

A detecção de pose corporal permite determinar quando alguém está presente ou se movimentando em frente ao display, também é possível colocar uma condição para iniciar uma atividade, por exemplo, o usuário ter que fazer um sinal de 'Ok' para iniciar um jogo. Com essas informações, displays digitais podem adaptar o conteúdo exibido de acordo com o comportamento do público, ou criar aplicações em que o usuário ativamente interage com seus movimentos.

2.4.2. Matter.js

Uma vez estabelecido o mapeamento dos gestos do usuário, torna-se essencial disponibilizar elementos com os quais ele possa interagir. É nesse momento que se desenvolve a lógica funcional da aplicação, definindo quais interações desencadeiam eventos específicos e orientam o fluxo de uso. Para conferir maior dinamismo e fluidez à experiência interativa, é comum que os desenvolvedores incorporem elementos dotados de propriedades físicas, como colisão, velocidade e inércia. Nesse contexto, a biblioteca **Matter.js** [Brummit 2023] mostra-se uma ferramenta adequada, pois permite a criação de objetos e simulações que reproduzem, com precisão, comportamentos físicos diversos. Seu objetivo é permitir que desenvolvedores criem simulações interativas realistas, atribuindo a objetos gráficos propriedades físicas (massa, densidade, atrito etc.) e comportamentos dinâmicos.

Em particular, a Matter.js pode implementar detecção e resposta a colisões, gravidade uniforme e restrições mecânicas (como molas e juntas), recursos essenciais para adicionar realismo físico (queda, quicadas, empurrões) em jogos e aplicações de sinalização digital interativa.

A biblioteca funciona a partir de seus principais componentes: que incluem o *Engine*, o *World*, o *Composite* e as entidades *Body* e *Bodies*. O módulo *Matter.Engine* representa o motor de simulação física, responsável por atualizar o estado do mundo a cada quadro. Ao se criar um engine, é gerada uma instância de Engine que contém o atributo *engine.world*: a raíz do sistema, abrigando todos os corpos e restrições da simulação. Por sua vez, *Matter.Bodies* oferece fábricas para criação de formas geométricas comuns (círculos, retângulos, polígonos), enquanto *Matter.Body* designa cada objeto físico

individual criado através dessas fábricas. Além disso, o módulo *Matter.Composite* fornece métodos para agregar corpos em estruturas complexas, permitindo a composição de objetos complexos a partir de partes simples.

Uma vez criados, esses corpos podem ser manipulados dinamicamente. Por exemplo, o método *Matter.Body.applyForce* aplica uma força linear a um corpo a partir de um ponto no espaço, gerando aceleração e torque correspondentes. Em contrapartida, *Matter.Body.setVelocity* define diretamente a velocidade linear do corpo, e *Matter.Body.setPosition* atualiza instantaneamente a posição do objeto no mundo. O atributo body.torque acumula a força de rotação aplicada a cada atualização do motor (zerando a cada passo), indicando a tendência de giro do corpo em cada frame.

No navegador web, a integração do Matter.js é feita tipicamente via renderização em um canvas ou contexto similar. Para visualização, o módulo Matter.Render provê um renderizador básico em um elemento canvas, no entanto, a renderização desses objetos também pode ser feita por ferramentas externas, como será mostrado no próximo tópico.

Em cenários de produção, pode-se usar o laço de execução próprio *Matter.Runner* ou um loop *requestAnimationFrame* personalizado, chamando repetidamente Engine.update e, então, desenhando os corpos. Para entrada de usuário, Matter.js inclui o plugin *Matter.MouseConstraint*, que permite interagir com os corpos através de cliques ou toques na tela. Essa abordagem de restrição do ponteiro pode ser estendida para sinais gestuais, em que a posição rastreada da mão é mapeada para ações físicas na simulação.

2.4.3. P5.js

Como discutido na subseção anterior, para que os elementos simulados pelo Matter.js sejam visíveis ao usuário, é necessário renderizá-los em um elemento gráfico da tela, como um canvas. Embora a própria biblioteca forneça um renderizador básico por meio de seu módulo *Matter.Render*, esse recurso limita-se à exibição das formas geométricas simples presentes no "mundo" da simulação. Tal abordagem pode ser insuficiente para aplicações que demandam maior complexidade visual, como, por exemplo, a exibição de textos, imagens/fotos ou interfaces que respondam a toques e arrastos. Para atender a essas necessidades, o Matter.js permite integração com outros contextos gráficos mais versáteis, como a biblioteca **P5.js** [McCarthy 2025], que será apresentada nesta seção do minicurso.

A **P5.js** é uma biblioteca JavaScript - feita com base no *Processing.js* - voltada a artistas e iniciantes, que simplifica a programação criativa e a manipulação de elementos gráficos no navegador. Com p5.js, basta chamar *createCanvas()* dentro de uma função *setup()* para gerar o elemento canvas automaticamente e usar a função *draw()* para redesenhar a cena a cada frame, e também, é aqui que os elementos do matter também podem ser desenhados. A biblioteca conta com uma vasta quantidade de funções destinadas a desenho, desde desenho 2D até 3D, além de formas geométricas e também elementos DOM. Além disso, p5.js conta com uma comunidade ativa e diversas bibliotecas complementares (*p5.sound*, *p5.dom*, *p5.accessibility*), tutoriais e exemplos oficiais que aceleram o protótipo e a experimentação de interfaces interativas para desenvolvedores iniciantes.

2.5. Estudo de caso: Criação de uma aplicação interativa utilizando o conjunto MMP

Tendo sido apresentadas as capacidades e vantagens das bibliotecas que compõem o conjunto MMP, esta seção demonstrará, na prática, o uso dessas ferramentas por meio de um tutorial para o desenvolvimento de um projeto simples de jogo interativo. O objetivo é evidenciar o potencial dessas bibliotecas na criação de experiências digitais imersivas, mesmo em cenários introdutórios.

Neste tutorial, será desenvolvido um jogo interativo que visa a detecção de gestos com ml5.js, a renderização gráfica com p5.js e a simulação física com Matter.js. Ao final da aplicação, o jogador utilizará um gesto de pinça diante da webcam para "agarrar" palavras flutuantes, arrastá-las e posicioná-las corretamente sobre plataformas fixas, formando uma sequência coerente. O tutorial abordará desde a preparação do ambiente e a captura de vídeo, passando pela configuração do motor físico, até a criação, exibição e manipulação das entidades textuais, com lógica de colisão e validação automática da vitória quando todas as palavras estiverem corretamente organizadas. Com esta atividade, será possível compreender como integrar modelos de aprendizado de máquina para rastreamento de mãos, manipular corpos rígidos em um espaço bidimensional e renderizar conteúdos gráficos interativos em um canvas.

O jogo é uma demonstração de uma aplicação que seria executada em um totem interativo com câmera, instalado em um ambiente físico da empresa fictícia "DYB" (*Develop Your Dreams*). A dinâmica consiste em desafiar o jogador a organizar corretamente palavras, posicionando-as nos espaços designados (slots) na tela. Ao completar a tarefa com sucesso, a aplicação exibirá uma mensagem de vitória, informando que o jogador concluiu o desafio. Para reiniciar o jogo, será solicitado ao usuário que una todos os dedos diante da câmera, gesto que acionará a reinicialização da interface interativa.

2.5.1. Estrurura do Projeto Inicial da Página HTML

Comumente, os projetos que utilizam as bibliotecas do conjunto MMP utilizam o editor online do P5.js [McCarthy 2025], porém, haja vista que esta aplicação possui o intuito de ser publicada na web, utilizaremos a arquitetura mostrada na Figura 2.7, que é uma estrutura básica para uma página HTML.

O arquivo *index.html* deve ter uma estrutura da página web, com um campo reservado para o canvas em que será exibido o jogo. E também deverá ter as importações das bibliotecas por meio de CDN, assim como no Código 2.1.

```
<!DOCTYPE html>
<html lang="pt-BR">
<head>
  <!-- Links CDN para as bibliotecas -->
  <script src="https://cdnjs.cloudflare.com/ajax/libs/p5.js/1.11.1/p5.
    js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/matter-js/0.20.0/
    matter.min.js"></script>
  <script src="https://unpkg.com/ml5@1/dist/ml5.min.js"></script>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0"
```

```
    >
</head>
<body>
  <main>
    <!-- local onde ficará o canvas da aplicação -->
    <div class="canvas-container"></div>
  </main>
  <script src="scripts/sketch.js"></script>
  <script src="scripts/boxes.js"></script>
  <script src="scripts/gameLogic.js"></script>
</body>
</html>
```

Código 2.1. Estrutura da página html

2.5.2. Setup do mapeamento de mão e do canvas do P5.js

Agora, para iniciar, faremos a configuração do modelo *handpose* do Ml5.js, que faz o mapeamento da mão do usuário. No arquivo *sketch.js* (Código 2.2), devem ser criadas as variáveis que serão utilizadas pelo ml5. Também são criadas as variáveis que estarão relacionadas às pontas dos dedos do usuário, pois cada ponto de mapeamento da mão recebe uma numeração própria, como pode ser observado na Figura 2.6. Após isso, dentro da função *preload* - específica do P5.js que é executada automaticamente - o modelo handpose é carregado com parâmetros que definem o número de mãos que serão mapeadas simultaneamente e com a captura flipada (por meio do atributo *Flipped*, que inverte a imagem horizontalmente), para que a imagem não fique invertida para o usuário. Após isso, na função *setup*, o canvas é criado e colocado na div reservada no HTML, e também é iniciada a captura de vídeo, que é inicialmente escondida pela função *video.hide()*, pois ela gera um elemento <video> no DOM, fora do canvas. Posteriormente, esse vídeo será exibido pelo próprio P5.js, na função *draw()*. A função *gotHands()* é uma callback que armazena results - que é um array de todos os pontos do mapeamento - na variável hands, para que possamos pegar as coordenadas dos pontos nos próximos passos.

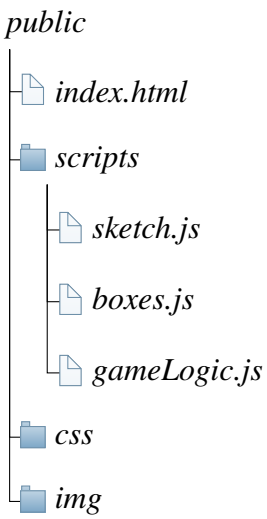

Figure 2.7. Estrutura de diretórios do projeto

```
let handPose, video, hands = [];

const INDEX_FINGER_TIP = 8; //Indice da ponta do indicador
const THUMB_TIP = 4; // Indice da ponta do polegar

function preload() {
  // Carrega o modelo handPose do Ml5.js
  handPose = ml5.handPose({ maxHands: 1, flipped: true });
}

function setup() {
  // Cria canvas e captura de vídeo (p5.js)
  let canvas = createCanvas(1280, 960);
  canvas.parent(document.querySelector(".canvas-container"));

  video = createCapture(VIDEO, { flipped: true });
  video.size(width, height);
  video.hide(); // Oculta o elemento <video> do DOM

  // Inicia detecção de mãos
  handPose.detectStart(video, gotHands);
}

function gotHands(results) {
  hands = results; // Atualiza keypoints das mãos
}
```

Código 2.2. Configuração do modelo Handpose e do Canvas.

2.5.3. Configurando o Motor de Física

Ainda no sketch, agora devemos realizar os preparos para a criação dos objetos do jogo (Código 2.3). Primeiro, são extraídos os componentes do Matter que serão utilizados neste projeto, eles são:

- **Engine** - Cria e controla o motor de física (tempo, atualização etc.)
- **Composite** - Está relacionado a agrupamento de corpos (compostos), neste projeto servirá principalmente adicionarmos elementos ao composto *Engine.world*, que abriga todos os elementos.
- **Bodies** - Fábrica de corpos: circle(), rectangle(), etc.
- **Body** - Manipula um corpo físico (mover, rotacionar, alterar massa...)

As aplicações desses componentes serão observadas no decorrer deste tutorial. Após isso, novamente na função *setup()*, o motor de simulação é criado por meio do *Engine*, e então, utilizando o *Bodies*, cria-se um chão para que os objetos não caiam para fora do canvas e dois círculos que servirão de ponteiros para os dedos; eles possuem a propriedade *isStatic*, pois não queremos que eles sejam afetados pela gravidade. Logo após isso, esses corpos são adicionados ao composto *Engine.world*, para que sejam incluídos no motor da simulação.

```
const { Engine, Bodies, Composite, Body } = Matter;

let engine, pointer, thumbPointer, ground;

function setup() {
  // ... (código de vídeo)

  // Cria o motor de física
  engine = Engine.create();

  // Cria um chão e pointeiros estaticos que serao presos nas pontas do
    dedo
  ground = Bodies.rectangle(width / 2, width * 3 / 4 - 50, width, 30, {
    isStatic: true });
  pointer     = Bodies.circle(0,0, 10, { isStatic: true });
  thumbPointer= Bodies.circle(0,0, 10, { isStatic: true });

  // Adiciona ao mundo
  Composite.add(engine.world, [ground, pointer, thumbPointer]);
}
```

Código 2.3. Configuração do motor de simulação Física do Matter e inclusão de elementos no mundo.

2.5.4. Criação das caixas de palavras e dos Slots do jogo

Agora, iremos os objetos principais para o jogo, as "caixas" que servirão de colisores para as palavras que irão formar as frases que o jogador deve formar, e os encaixes/plataformas em que ele deverá posicionar as caixas. Para isso, no arquivo *boxes.js* iremos montar um construtor para as caixas de colisão das palavras e das plataformas do jogo (Código 2.4).

O atributo *letters* recebe o array *gameWord* que iremos criar no *sketch.js*, ele contém a frase que deverá ser formada. O atributo *followingIndex* é um indicador de qual palavra o jogador está segurando no momento. E o atributo *slotsFilled* servirá para registrar quais encaixes foram preenchidos e o seu conteúdo; esses atributos servem para o controle da lógica do jogo, e em um próximo tópico, eles serão abordados.

Além disso, também é utilizado o componente *Bodies* para criarmos as colisões em formato de retângulo nas linhas 27 e 42 do Código 2.4.

```
class Boxes {
    constructor(num, size) {
        //quantidade e tamanho das caixas de colisão das palavras
        this.num = num;
        this.size = size;

        this.bodies = []; //array para as caixas das palavras do jogo
        this.platforms = [];//array para as caixas das plataformas das
    palavras

        this.letters = gameWord;  //palavras do jogo
        this.followingIndex = -1; //Variável que indica a caixa que est
    á seguindo o dedo indicador
        this.slotsFilled = new Array(num).fill(null);
        this.gameCompleted = false;

```

```
        this.initialize();
    }

    // Funçao que é chamada quando a classe Boxes é instanciada
    initialize() {
        // Criaçao das colisoes para cada palavra do jogo
        for (let i = 0; i < this.num; i++) {
            //posição de spawn de cada palavra e a largura(depende do tamanho da palavra)
            let wordWidth = textWidth(this.letters[i]) + 20;
            let x = random(100, 1100);
            let y = random(550, 750);

            let box = Bodies.rectangle(x, y, wordWidth * 2, this.size, {
                //propriedades físicas de cada caixa
                frictionAir: 0.1,
                restitution: 0.6,
                density: 0.002,
                angle: random(-0.1, 0.1)
            });

            this.bodies.push(box);
            Composite.add(engine.world, [box]); //adiciona as caixas para o mundo

            //Criação das plataformas para as palavras
            let platformX = width / 4 + i * 200;
            let platformY = 220;

            let platform = Bodies.rectangle(platformX, platformY + 10, this.size * 5, 17, { isStatic: true });
            this.platforms.push(platform);
            Composite.add(engine.world, [platform]); //adiciona as plataformas no mundo
        }
    }
}
```

Código 2.4. Construtor das caixas.

Com o construtor finalizado, vamos criar no *sketch.js* as palavras do jogo e instanciar a classe **Boxes** na função *Setup* (Código 2.5).

```
const gameWord = ["Develop", "Your", "Brain"];

function setup(){
    // ... restante do codigo ...
    boxes = new Boxes(gameWord.length, 40);
}
```

Código 2.5. Adição da gameWord e intanciação da classe Boxes.

2.5.5. Implementação das Regras do Jogo

Uma vez criados todos os objetos que iremos usar, agora iremos para a criação das mecânicas de jogabilidade. O jogo pode ser dividido em duas mecânicas principais:

- Pegar uma palavra - o jogador, fazendo um gesto de pinça (aproximar a ponta do dedo indicador com o polegar), poderá pegar uma das palavras ao aproximar os dedos da palavra. A palavra vai seguir os movimentos da mão do jogador se ele manter os dedos juntos. Se ele quiser soltar a caixa, precisa apenas separar os seus dedos.
- Colocar a palavra em um slot - uma vez que o jogador pegue uma palavra, ele deve movimentála até uma das plataformas, se ele posicionar na correta, a palavra irá parar de seguir a mão. Uma vez que todas os encaixes tenham sido preenchidos, o jogador ganhou o jogo.

Para implementar essas mecânicas, continuaremos incluindo funções na classe Boxes, então, para separarmos o código relacionado à construção das caixas e à lógica do jogo, iremos criar as funções no arquivo *gameLogic.js* (Código 2.6). A função *checkCollision()* é responsável por verificar a colisão entre os dedos e a caixa, e fazer com que ela os siga uma vez que tenha sido encostada. Enquanto isso, a função *checkPlatforms()* verifica a colisão entre a palavra e o encaixe, e armazena a palavra encaixada em *slotsFilled*. Uma vez que todas as palavras sejam encaixadas, a função *win* é chamada e executa um registro no console, indicando a vitória (numa aplicação real isso pode ser substituído por algum outro evento específico).

```
1  // Função que vai checar se o usuário está fazendo o gesto de pinça
       para pegar uma palavra
2  Boxes.prototype.checkCollision = function (pointer, thumb) {
3
4      let distance = dist(pointer.position.x, pointer.position.y, thumb.
       position.x, thumb.position.y);
5
6      //Primeiro verifica se tem alguma caixa seguindo o dedo
7      if (this.followingIndex === -1) {
8
9          // Verifica se o gesto de pinça foi feito e então checka a
       proximidade do dedo com a caixa
10         if (distance < 50) {
11             for (let i = 0; i < this.bodies.length; i++) {
12                 let box = this.bodies[i];
13                 let boxDistance = dist(box.position.x, box.position.y,
       pointer.position.x, pointer.position.y);
14
15                 if (boxDistance < this.size * 1.5 && !this.slotsFilled.
       includes(i)) {
16                     this.followingIndex = i;
17                     break;
18                 }
19             }
20         }
21     } else {
```

```
22          let box = this.bodies[this.followingIndex]; //pega a caixa que
    está seguindo o dedo
23
24          // Se os dedos se afastarem, parar de seguir
25          if (distance > 120) {  // Define um limite para soltar a caixa
    - ao afastar os dedos a caixa é solta
26              this.followingIndex = -1;
27          } else {
28              let newPosition = {
29                  x: lerp(box.position.x, pointer.position.x, 0.2), // o
    lerp é usado para dar suavidade no movimento de seguir da caixa, nã
    o obrigatório
30                  y: lerp(box.position.y, pointer.position.y, 0.2)
31              };
32              Body.setPosition(box, newPosition);
33          }
34      }
35  }
36  Boxes.prototype.checkPlatforms = function () {
37
38      for (let i = 0; i < this.bodies.length; i++) {
39          let box = this.bodies[i];
40          let platform = this.platforms[i];
41
42          //Traça uma distancia entre cada caixa e a sua respectiva
    plataforma
43          let distance = dist(box.position.x, box.position.y, platform.
    position.x, platform.position.y);
44
45          if (distance < this.size) {
46              //posiciona a caixa no mesmo lugar da plataforma ao
    encostar
47              Body.setPosition(box, { x: platform.position.x, y: platform
    .position.y - this.size / 2 });
48              Body.setStatic(box, true);
49
50              if (!this.slotsFilled[i]) { // Só armazena se ainda não
    estiver preenchido
51                  this.slotsFilled[i] = this.letters[i]; //armazena a
    palavra da caixa
52              }
53
54              if (this.followingIndex === i) {
55                  this.followingIndex = -1;
56              }
57          }
58      }
59
60      if (this.slotsFilled.every(slot => slot !== null)) {
61          this.win();
62      }
63
64      console.log(this.slotsFilled);
65  }
66
```

```
67 //função que define o que será feito quando o jogador ganha o jogo
68 Boxes.prototype.win = function () {
69     console.log("Você venceu!");
70     this.gameCompleted = true;
71 }
```

Código 2.6. Lógica das mecânicas do jogo.

2.5.6. Desenhando os elementos do jogo

Até este momento, todos os objetos e corpos que criamos até agora ainda não podem ser visualizados pelo usuário, pois nenhum deles foi renderizado pelo P5.js. Isto é feito por meio de funções de desenho que serão colocadas em uma função especial do P5 chamada *draw()*. Mas antes de voltarmos para o *sketch.js*, é preciso primeiro criar as funções de desenho das caixas no construtor da classe Boxes(Código 2.7. Fazemos isso criando uma função da classe Boxes chamada *display()* que depois será chamada no *draw()*. Nela, utilizamos as funções *rect()* - que criam retângulos - para fazermos os encaixes das palavras e *text()* para escrever as palavras nas caixas de colisão, que não serão visualizadas pelo usuário.

```
1  display() {
2
3      textAlign(CENTER, CENTER); //alinhar as palavras com as colisoes
4      textSize(24);
5
6      // desenho das plataformas
7      fill("#d3d3d3");
8      for (let i = 0; i < this.platforms.length; i++) {
9          let px = this.platforms[i].position.x;
10         let py = this.platforms[i].position.y - 20;
11         let wordWidth = textWidth(this.letters[i]) + 20;
12
13         rectMode(CENTER);
14         rect(px, py, wordWidth * 1.5, this.size * 1.3);
15     }
16
17     // "desenho" das palavras que serão posicionadas nas caixas
18     for (let i = 0; i < this.bodies.length; i++) {
19         let x1 = this.bodies[i].position.x;
20         let y1 = this.bodies[i].position.y;
21
22         textSize(40);
23         text(this.letters[i], x1, y1);
24     }
25 }
```

Código 2.7. Desenho das plataformas e das palavras.

Feito isso, só resta desenharmos o resto dos elementos no *sketch.js* por meio da função *draw()* 2.8. A função draw é executada repetidas vezes por segundo, por isso, além das funções de desenho, também está nela o *Engine.Update()*, para que o motor de simulação seja atualizado constantemente, assim permitindo a fluidez dos movimentos dos corpos. As funções de desenho presentes no draw incluem:

- **image()** - Desenha uma imagem no canvas, como ela está na função *draw()* e está recebendo "video" como parâmetro, essa função está servindo como um reprodutor da imagem capturada pela câmera.

- **ellipse()** - Desenha formas elípticas. No jogo esses círculos são usados para indicar as pontas dos dedos do usuário

- **rect() e text()** - presentes dentro da função *boxes.display()*, como citado anteriormente, desenham retângulos e textos, respectivamente.

E por fim, a função *CheckCollision()* está sendo chamada com os círculos das pontas dos dedos como parâmetro, para que a posição seja verificada, e também a função *checkPlatform()* é chamada para constantemente verificar o estado do jogo.

```
function draw() {

  background(220);

  Engine.update(engine);
  // Exibe a imagem da webcam ajustada ao tamanho do canvas
  image(video, 0, 0, width, height);


  // Desenha Círculos nas pontas dos dedos
  if (hands.length > 0) {
    index = hands[0].keypoints[INDEX_FINGER_TIP];
    thumb = hands[0].keypoints[THUMB_TIP];

    pointer.position.x = index.x;
    pointer.position.y = index.y;

    thumbPointer.position.x = thumb.x;
    thumbPointer.position.y = thumb.y;

    fill("#49E60F");
    ellipse(index.x, index.y, 10);
    ellipse(thumb.x, thumb.y, 10);
  }

  boxes.checkCollision(pointer, thumbPointer);
  boxes.checkPlatforms();
  boxes.display();

}
```

Código 2.8. Desenho das plataformas e das palavras.

Com todos esses passos concluídos, a aplicação imersiva está completa, possuindo funções de machine learning e de simulações físicas, sendo configurada apenas com lógica básica de JavaScript.

2.6. Conclusão

Ao longo deste capítulo, discutimos os fundamentos da interatividade por gestos em sinalização digital partindo do surgimento dessa tecnologia até como ela é aplicada atualmente. Em seguida, exploramos a importância do aprendizado de máquina para o treinamento e a inferência de modelos capazes de reconhecer gestos, e como o WebGL viabiliza essa computação no navegador. Na sequência, apresentamos o conjunto MMP — ml5.js, Matter.js e P5.js — detalhando, em cada subseção, como ml5.js fornece uma interface amigável para acesso a modelos de visão, como Matter.js adiciona realismo físico às interações e como P5.js enriquece a renderização gráfica. Por fim, demonstramos tudo isso na prática por meio de um tutorial que integra detecção de gestos, simulação 2D e desenho em canvas para criar uma aplicação interativa.

Espera-se que, com essa trajetória — do conceito à aplicação prática —, o leitor tenha uma visão clara não apenas das teorias envolvidas, mas também do potencial dessas ferramentas para se criar verdadeiras experiências digitais imersivas.

References

[Asif et al. 2024] Asif, R., Naveed, A., Farasat, M., and Khan, M. I. (2024). Impact of digital signage and social media advertising on consumer buying behavior: Mediating role of emotional processes. *Journal of Asian Development Studies*, 13(1):1147–1160.

[BrightSign LLC 2025] BrightSign LLC (2025). Brightsign®: Soluções de mídia para sinalização digital. Empresa líder global em players de mídia para sinalização digital, fundada em 2002 e sediada em Los Gatos, Califórnia.

[Brignull and Rogers 2003] Brignull, H. and Rogers, Y. (2003). Enticing people to interact with large public displays in public spaces. In *Interact*, volume 3, pages 17–24.

[Brummit 2023] Brummit, L. (2023). Matter.js is a 2d physics engine for the web. Disponível em: `https://brm.io/matter-js/`. Acesso em: 14 de março de 2025.

[Chen et al. 2009] Chen, Q., Malric, F., Zhang, Y., Abid, M., Cordeiro, A., Petriu, E. M., and Georganas, N. D. (2009). Interacting with digital signage using hand gestures. In Kamel, M. and Campilho, A., editors, *Image Analysis and Recognition*, pages 347–358, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Chung 2017] Chung, S. Z. (2017). Smart digital signage with eye tracking system. Masters thesis, Universiti Sains Malaysia.

[Coldewey 2019] Coldewey, D. (2019). This hand-tracking algorithm could lead to sign language recognition. `https://techcrunch.com/2019/08/19/this-hand-tracking-algorithm-could-lead-to-sign-language-recognition/`. Acessado em 9 de maio de 2025.

[Davies et al. 2022] Davies, N., Clinch, S., and Alt, F. (2022). *Pervasive displays: understanding the future of digital signage*. Springer Nature.

[de Oliveira and de Melo 2022] de Oliveira, B. V. N. and de Melo, F. T. (2022). Fundamentos da visão computacional: Arcabouço teórico do reconhecimento artificial de imagens e vídeos. *Humanidades & Inovação*, 10(17):313–324.

[Digital 2025] Digital, T. (2025). Google lança recurso de publicidade para digital out-of-home. Disponível em: https://www.teoriadigital.com.br/google-publicidade-digital-dooh/. Acesso em: 22 de abril de 2025.

[HIIT 2025] HIIT (2025). Citywall. Disponível em: https://citywall.org/. Acesso em: 25 de abril de 2025. Helsinki Institute for Information Technology.

[IBM Brasil 2023] IBM Brasil (2023). O que são redes neurais convolucionais? https://www.ibm.com/br-pt/think/topics/convolutional-neural-networks. Acessado em maio de 2025.

[McCarthy 2025] McCarthy, L. L. (2025). P5.js - friendly tool for learning to code and make art. Disponível em: https://p5js.org/. Acesso em: 14 de março de 2025.

[McDonald's Corporation 2020] McDonald's Corporation (2020). Self-Ordering Kiosks Improve McDonald's Customer Experience. https://corporate.mcdonalds.com/corpmcd/scale-for-good/our-planet/kiosk-experience.html. Acesso em: 28 abr. 2025.

[Microsoft 2010] Microsoft (2010). Kinect for Windows Sensor. https://developer.microsoft.com/en-us/windows/kinect/. Acesso em: 28 abr. 2025.

[Microsoft 2019] Microsoft (2019). Onnx.js: Run onnx models in browser using javascript. https://github.com/microsoft/onnxjs. Acessado em: 2025-05-09.

[ml5.js contributors 2018] ml5.js contributors (2018). ml5.js: Friendly machine learning for the web. https://ml5js.org. Acessado em: 2025-05-09.

[Müller et al. 2012] Müller, J., Walter, R., Bailly, G., Nischt, M., and Alt, F. (2012). Looking glass: a field study on noticing interactivity of a shop window. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, page 297–306, New York, NY, USA. Association for Computing Machinery.

[Parisi 2012] Parisi, T. (2012). *WebGL: up and running*. " O'Reilly Media, Inc.".

[Peltonen et al. 2008] Peltonen, P., Kurvinen, E., Salovaara, A., Jacucci, G., Ilmonen, T., Evans, J., Oulasvirta, A., and Saarikko, P. (2008). It's mine, don't touch! interactions at a large multi-touch display in a city centre. In *Proceedings of the SIGCHI conference on human factors in computing systems*, pages 1285–1294.

[SEAT S.A. 2021] SEAT S.A. (2021). SEAT Creates Digital Showroom Experience. https://www.seat.com/company/news/corporate/digital-showroom-technology.html. Acesso em: 28 abr. 2025.

[Shiffman et al. 2018] Shiffman, D. et al. (2018). ml5.js: Friendly open source machine learning library for the web. https://itp.nyu.edu/adjacent/issue-3/ml5-friendly-open-source-machine-learning-library-for-the-web/. Acessado em: 12 de maio de 2025.

[TensorFlow.js Team 2023] TensorFlow.js Team (2023). Platform and environment. https://www.tensorflow.org/js/guide/platform_environment. Acessado em maio de 2025.

[TV 2023] TV, O. (2023). Using Proximity Sensors with Digital Signage. https://onsign.tv/blog/technology/motion-sensors-with-digital-signage/11. Acesso em: 28 abr. 2025.

[Walter et al. 2013] Walter, R., Bailly, G., and Müller, J. (2013). Strikeapose: revealing mid-air gestures on public displays. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 841–850.