Introdução à Computação Gráfica

Release 0.1

Carlos Hitoshi Morimoto

Mar 21, 2022

Sumário

CAPÍTULO 1

Introduções

Seja bem vindo à disciplina de Introdução à Computação Gráfica! Nesse semestre, nossas aulas voltam a ser presenciais, nos seguintes horários:

- 4as-feiras das 10:00 às 11:40 h; e
- 6as-feiras das 08:00 às 09:40 h.

Além das aulas, convidamos você para participar também das discussões no Discord enviando suas dúvidas, discutindo e respondendo às dúvidas de seus colegas. Caso precisar falar de algum assunto mais particular, use o link converse com o Hitoshi disponível na página da disciplina para enviar mensagens diretas para o Hitoshi.

Para sua conveniência, alguns detalhes sobre a disciplina também estão na seção informações sobre MAC0420 e MAC5744. Em particular, a disciplina vai adotar o seguinte critério de avaliação.

1.1 Sobre MAC0420 e MAC5744

Essas duas disciplinas serão oferecidas em conjunto e visam introduzir conceitos de computação gráfica, fornecendo conhecimento teórico para a criação de imagens sintéticas e também experiência prática no desenvolvimento de programas gráficos interativos. Os principais assuntos a serem cobertos ao longo do curso são (não necessariamente nessa ordem):

- Fundamentos: histórico e aplicações, interfaces gráficas, dispositivos gráficos, e o pipeline gráfico.
- Padrões para representação de imagens e cores.
- Representação e construção de objetos geométricos.
- Geometrias, sistema de coordenadas e transformações geométricas.
- Recortes e janelas.
- Visibilidade, oclusão, buffer de profundidade.
- Ray-tracing.
- Mapeamento de texturas.
- Representação de curvas e superfícies.
- Representação de objetos tridimensionais.
- Animação.

A avaliação da disciplina será baseada em provas, provinhas e exercícios-programa (EPs). O critério de avaliação está descrito em critério de avaliação.

Os EPs serão desenvolvidos em WebGL e JavaScript. Dessa forma, os programas poderão ser executados em qualquer navegador moderno compatível com HTML5.

1.2 Etiqueta

Gostaríamos que todos participem das aulas da forma mais confortável e segura possível ao longo de todo o semestre, mas principalmente nesse período de readaptação ao ensino presencial. Por isso, por favor, siga as recomendações constantes no site Retorno Seguro - como voltar às atividades presenciais com segurança.

Caso você apresente sintomas e/ou teste positivo para qualquer variante do vírus SARS-CoV-2, mantenha-se afastado e comunique sua situação pelo link converse com o Hitoshi.

1.3 Pré-requisitos

Para fazer essa disciplina vamos assumir que você já seja um programador de nível intermediário e, por esse motivo, a disciplina tem como pré-requisito MAC0122. Você deve fazer uso também de seus conhecimentos sobre algoritmos, estruturas de dados e fundamentos de geometria e álgebra linear. No entanto, não é necessário nenhuma experiência prévia com OpenGL ou JavaScript.

1.4 Bibliografia

Não vamos adotar um livro texto em particular, mas muito do material teórico que utilizamos, inclusive para escrever nossas notas de aula, são baseadas nos textos a seguir:

- Interactive Computer Graphics. A top-down approach with WebGL. Dave Shreiner and Edward Angel.
  - Baixe e instale o Material do livro
- Computer Graphics: Principles and Practice. James Foley, Andries Van Dam, and Steve Feiner.
- Prof._Dave Mount Computer Graphics Notes

Ao longo do curso, vamos desenvolver vários programas usando WebGL e JavaScript, que serão executados a partir de um navegador Web moderno, compatível com versões mais atuais de HTML. Você pode utilizar esses links para conhecer melhor essas ferramentas.

- Estruturando a web com HTML
- Uma reintrodução ao JavaScript
- WebGL Tutorial Introduction_to_HTML)
- WebGL2 Fundamentals

1.4.1 Outros recursos online

Há muito material a sua disposição para aprender mais sobre CG, WebGL e Javascript. Sinta-se a vontade de compartilhar seus links favoritos nas nossas listas de discussão. Vamos começar indicando apenas algumas por enquanto.

- Khronos Group – WebGL
- Khonos WebGL2 Quick Reference Guide
- Introduction to Computer Graphics - David Eck

1.5 Introdução à Computação Gráfica

A **computação gráfica** (CG) é uma área da computação que trata de assuntos relacionados à produção de imagens e animações (ou sequências de imagens) por meio de um computador.

A CG teve início no final da década de 1950 quando Ivan Sutherland, um dos pioneiros da área, desenvolveu o Sketchpad como resultado de sua tese de doutorado. Usando um hardware e software muito simples para os padrões atuais, o Sketchpad permitia produzir desenhos de linhas como ilustrado na Figura Fig. 1.1. Além de grande impacto na CG, as ideias introduzidas pelo Sketchpad influenciaram também as áreas de CAD (*computer aided design* – design assistido por computador), HCI (*human computer interaction* – interação humano-computador) e programação orientada a objetos. Por essas contribuições, Ivan Sutherland recebeu o prêmio de Turing em 1988 e o prêmio de Kyoto em 2012.

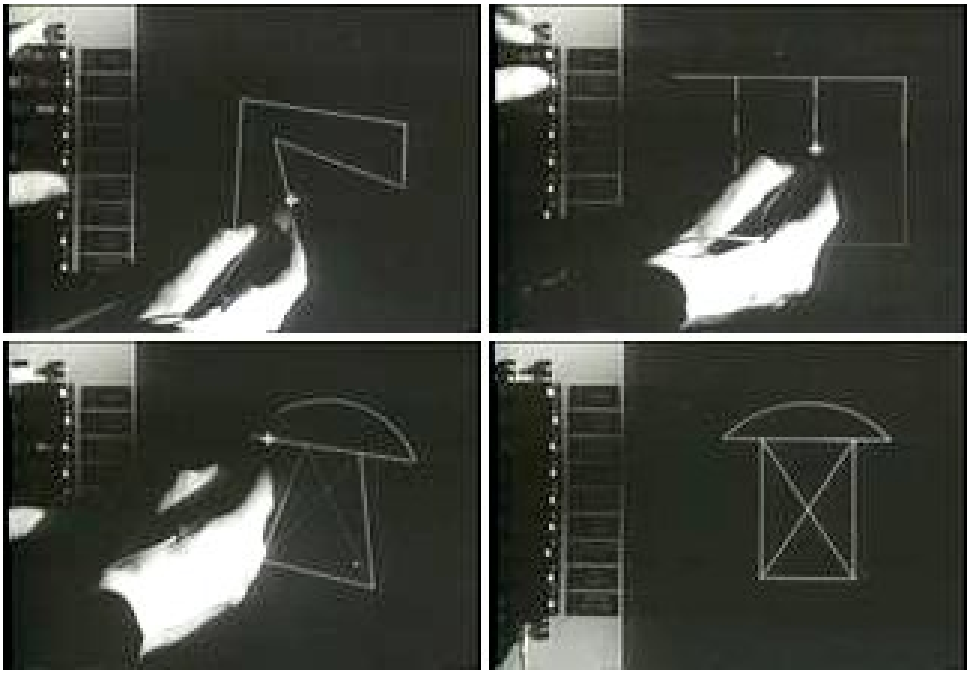

Fig. 1.1: Skethpad. (Imagem reproduzida da Wikipedia)

Como tantas outras áreas da computação a Computação Gráfica se desenvolveu muito nas últimas décadas, tanto no hardware quanto no software, e continua se desenvolvendo com grande rapidez. Hoje, por exemplo, é impossível imaginar a área de entretenimento, como jogos eletrônicos, cinema e TV, sem a CG. A evolução é tamanha que hoje é possível produzir imagens que são praticamente indistinguíveis de imagens fotográficas ou, pelo menos, criam uma ilusão de realidade bastante convincente. Além disso, as placas gráficas se tornaram tão poderosas que são utilizadas na construção de supercomputadores e tem alavancado o desenvolvimento de outras áreas de aplicação como inteligência artificial e realidade mista.

Ainda assim, a produção de imagens foto-realistas continua sendo um desafio por ser um processo extremamente complexo. Nesse curso, vamos fornecer uma breve introdução sobre os sistemas de hardware e software, e cobrir alguns fundamentos para produzir imagens de objetos 3D. Ao final você vai ser capaz de aplicar esses conhecimentos na produção de animações interativas com objetos 3D usando WebGL. O WebGL é uma versão do OpenGL, uma biblioteca gráfica considerada hoje um padrão para o desenvolvimento de aplicativos gráficos, suportada pela maioria dos navegadores modernos.

Mas assim como o foco em nossos cursos de introdução à computação não é no ensino de uma linguagem de programação, nosso foco neste curso será no desenvolvimento de habilidades para resolver problemas da CG. Por exemplo, vamos procurar compreender como fundamentos da matemática, da física, de algoritmos e estruturas de dados podem ser utilizados na sua produção de imagens digitais. Caso você esteja mais interessado na produção das imagens, ao invés dos fundamentos, há ferramentas computacionais mais específicas que você pode utilizar mas que não serão tratadas nesse curso.

1.5.1 Conteúdo do curso

A produção de imagens foto-realistas é um processo extremamente complexo pois envolve conhecimentos de muitas áreas do conhecimento, como física, biologia, geometria, álgebra linear e muitas outras além da computação. Nesse curso vamos cobrir apenas alguns fundamentos da CG que permitam você manipular e renderizar (fazer o computador desenhar) uma cena composta por objetos tridimensionais. Como esses fundamentos são os mesmos utilizados na produção de efeitos especiais no cinema e na criação de jogos eletrônicos, você vai obter uma compreensão melhor sobre o funcionamento e produção desses efeitos e sistemas gráficos. Efeitos mais sofisticados em geral envolvem apenas uma modelagem física e da luz mais complexa e uso de técnicas mais avançadas de renderização.

Boa parte do curso está voltada ao entendimento do processo de produzir uma única imagem a partir de um modelo de cena bidimensional (2D) ou tridimensional (3D). Na verdade, este é um aspecto muito limitado da computação gráfica. Por exemplo, ele ignora o papel da CG em tarefas como a visualização de coisas que não podem ser descritas como "cenas". Isso inclui a renderização de desenhos técnicos, incluindo gráficos de engenharia e projetos arquitetônicos, e também visualização científica, como funções matemáticas, temperaturas do oceano, velocidades do vento e assim por diante. Iremos produzir animações simples (por meio da produção de muitas imagens únicas), mas questões mais específicas da animação, como *motion blur* (desfoque de movimento), *morphing* (transformação de forma), *blending* (mistura), *anti-aliasing* etc, não serão abordadas. Esses temas são tratados em cursos mais avançados.

O processo de geração de uma imagem a partir de um modelo é chamado de **renderização**. A renderização de uma cena 3D é ilustrado na Figura Fig. 1.2.

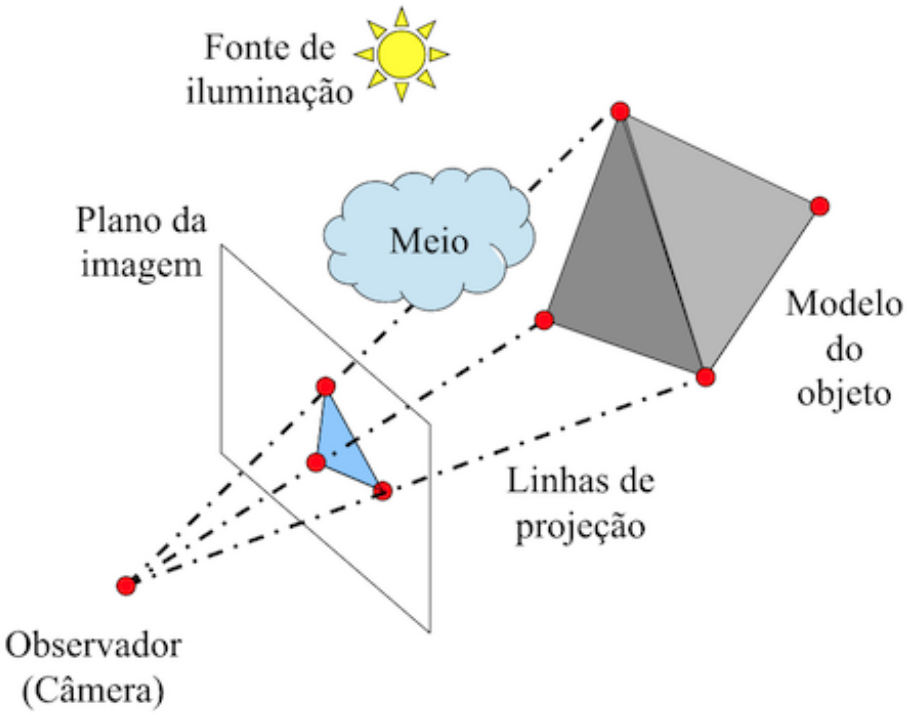

Fig. 1.2: Renderização de objetos 3D

O processo começa produzindo um modelo matemático do objeto a ser renderizado. Tal modelo deve descrever não apenas a forma do objeto, mas também várias outras propriedades como cor e tipo de acabamento da superfície como brilhante, fosco, transparente, difuso, escamoso, rochoso etc. Produzir modelos realistas é extremamente complexo, mas felizmente não é nossa principal preocupação. Vamos deixar isso para os artistas e modeladores.

O modelo de cena também deve incluir informações sobre a localização e as características das fontes de luz (sua cor, brilho) e a natureza atmosférica do meio pelo qual a luz viaja, como nevoeiro, água, fumaça etc.. Além disso, precisaremos saber a localização da câmera ou do olho do observador. Podemos pensar no observador como se estivesse segurando uma "câmera virtual" que tira uma "fotografia" da cena. Precisamos conhecer as características desta câmera como a sua distância focal e a razão de aspecto (*aspect ratio*), por exemplo.

Com base em todas essas informações, precisamos realizar uma série de etapas para produzir a imagem ("sintética") desejada.

- **Projeção**: Projetar a cena 3D sobre o plano (2D) da imagem na câmera virtual.
- **Cor e sombreamento**: Para cada ponto da imagem é necessário calcular sua cor. A cor é uma função da cor da superfície do objeto, da sua textura, da posição relativa das fontes de luz e, em modelos mais complexos de iluminação, da reflexão indireta da luz sobre as superfícies dos demais objetos da cena.
- **Remoção de superfícies escondidas**: Elementos que estão mais próximos da câmera escondem os objetos que estão atrás, mais distantes da câmera. É necessário calcular quais superfícies estão visíveis e quais não estão.
- **Rasterização**: Uma vez calculadas as cores para pintar cada ponto da imagem, a etapa final é mapear essas cores para o nosso dispositivo de exibição (monitor).

Ao final do semestre, você deve ter um entendimento básico de como cada uma das etapas é executada. Uma compreensão mais detalhada desses elementos vai além do escopo deste curso, até pela sua curta duração de um semestre. Mas, combinando o que vai aprender aqui com outros recursos (livros ou da Internet), você saberá o suficiente para, digamos, escrever um videogame simples, escrever um programa para gerar imagens altamente realistas ou produzir uma animação simples.

1.5.2 Principais tópicos

- Fundamentos
  - Programação gráfica: OpenGL (WebGL), funções gráficas primitivas, cor, observação, programação baseada em eventos, frame buffers.
  - Programação geométrica: revisão de álgebra linear, transformação afim, representação de pontos e vetores, coordenadas homogêneas, mudança de sistemas de coordenadas.
- Modelamento
  - Tipos: modelos poliédricos, hierárquicos, fractais e dimensão fractal.
  - Curvas e superfícies: representação de curvas e superfícies, interpolação, Bézier, B-splines, NURBS.
  - Acabamentos: mapeamento de textura, bump mapping.
- Projeção
  - Transformação 3D e perspectiva: escala, rotação, translação, projeção ortogonal e perspectiva, clipping.
  - Remoção de superfícies escondidas: algoritmo do pintor, back-face culling.
- Realismo
  - luz e sombreamento
  - ray-tracing
  - cor

1.6 Onde estamos e para onde vamos

Você deve ter agora um bom entendimento dos objetivos do curso, sua estrutura, e sobre o que você precisa fazer para ter um bom desempenho e ser aprovado.

Começamos descrevendo as principais etapas para produzir a imagem de um objeto e/ou cena 3D, que é um dos principais objetivos da CG.

Nosso próximo tópico será sobre sistemas gráficos e seus recursos básicos para desenhar imagens simples.

Para reforçar o conteúdo tratado nessas primeiras aulas, sugerimos as seguintes leituras:

- Computer Graphics: sobre a história da computação gráfica.
- Capítulo 1 do Livro do David Eck e/ou:
  - Capítulo 1 do livro "Interactive Computer Graphics" de Edward Angel.

CAPÍTULO 2

Formação e representação de imagens

A computação gráfica se preocupa com tudo que envolve a geração de imagens por computador. Nessa aula vamos apresentar com mais detalhes o processo de formação de imagens, a partir do padrão de luz que estimula nossos olhos, até as formas usadas para representar essas imagens. Recomendamos a seguinte leitura para complementar essas notas.

- Capítulo 1 do livro "Interactive Computer Graphics" de Edward Angel; e/ou:
  - Capítulo 1 do Livro do David Eck.

2.1 Formação de Imagens

As imagens geradas por computação gráfica são sintéticas ou artificiais, ou seja, os objetos mostrados nas imagens não são "reais" mas precisam ser gerados a partir de "modelos". Há vários aspectos que precisamos conhecer, como cor, iluminação e sua interação com as superfícies dos objetos, para que possamos gerar imagens que pareçam "realistas". Os principais elementos 3D que vamos considerar nesse curso estão ilustrados na Figura Fig. 2.1.

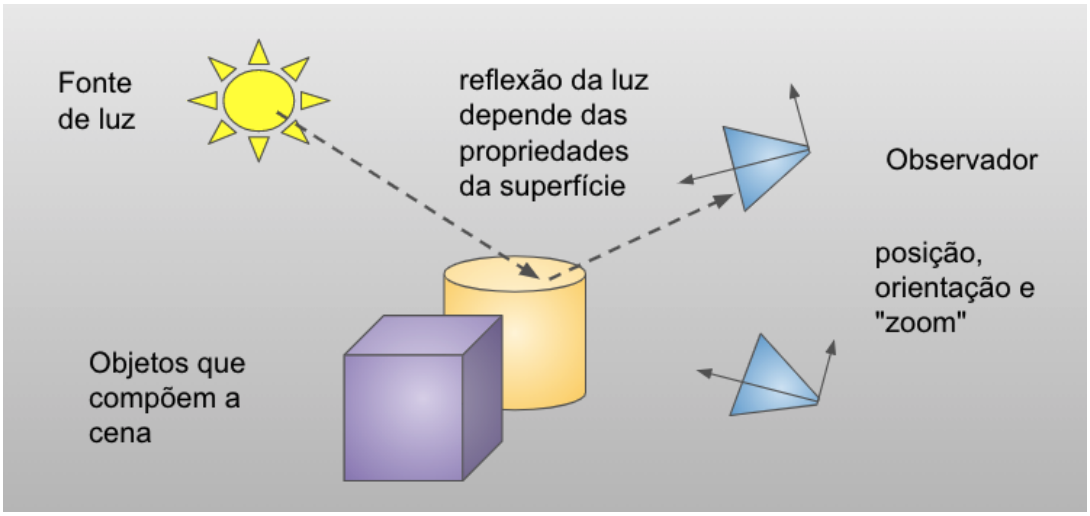

Fig. 2.1: Elementos que contribuem para a formação de uma imagem.

A luz emitida por uma fonte de luz e que chega à câmera depende de propriedades das superfícies dos objetos e da câmera (ou olho da pessoa observando a cena). Outras fontes de luz, reflexões múltiplas entre objetos, influência do meio onde a luz se propaga, são apenas alguns elementos que podem melhorar a qualidade desse modelo e que não estão ilustrados na figura.

Vamos discutir alguns desses elementos com mais detalhes.

2.2 Luz e percepção de cor

Uma propriedade básica da luz é sua **cor**. A luz visível, como onda eletromagnética, tem comprimento de onda entre 400 nm a 750 nm, que vai do violeta ao vermelho, cobrindo basicamente o espectro de cores como vemos em um arco-íris.

Nossa percepção de cor no entanto é um fenômeno mais complexo. Observe que a cor "branca", por exemplo, corresponde a nossa percepção de cor quando o olho é estimulado por radiação composta por todos esses comprimentos de onda. Isso ocorre devido a fisiologia do olho, como ilustrado na Figura Fig. 2.2.

A córnea é uma membrana externa transparente que protege o olho e permite a entrada de luz. A córnea basicamente cobre a íris, a parte colorida do olho, que tem uma abertura no centro chamada de pupila. A pupila muda de tamanho constantemente para regular a quantidade de luz que chega à retina, que é uma camada de células foto sensíveis depositada no fundo do olho. A fóvea é uma região na retina que concentra os **cones**, que são células foto sensíveis responsáveis pela visão cromática. Há 3 tipos de **cones**, cada um sensível a um determinado comprimento de onda, denominados cone$_R$, cone$_G$ e cone$_B$, pois respondem aos comprimentos de onda R (de red, ou vermelho), G (de green ou verde) e B (de blue ou azul). Não é a toa que os monitores coloridos são construídos para gerar estímulos tricromáticos RGB e é por isso também que a representação mais comum de imagens coloridas segue esse padrão.

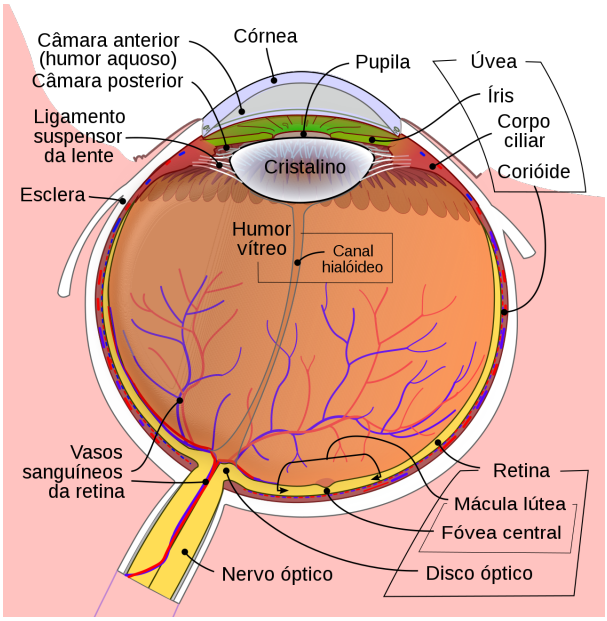

Fig. 2.2: Diagrama do olho humano. Imagem reproduzida da Wikipedia..

2.3 Objetos da cena e suas propriedades

Uma forma computacionalmente eficaz para representar a forma de objetos é por meio de elementos geométricos primitivos como pontos (vértices) e linhas. As superfícies podem ser decompostas em polígonos simples, como triângulos, que podem ser definidos por um conjunto de vértices. Curvas e superfícies complexas, como um disco ou esfera, podem ser aproximadas usando um número suficientemente grande de linhas ou triângulos. O processo de modelar objetos ainda mais complexos, como uma face humana, a partir desses elementos geométricos básicos não é simples e, por isso, existem várias ferramentas especializadas que facilitam a construção desses modelos como o Blender. No entanto, o uso dessas ferramentas não faz parte do escopo desse curso e por isso vamos nos limitar a objetos mais simples de modelar, para que possamos focar no processo de renderização desses modelos.

Vale aqui observar que a escolha de usar apenas elementos geométricos primitivos é a eficiência computacional para gerar essas imagens usando processadores gráficos modernos, que são otimizados para processar esses elementos. Processar elementos mais complexos é certamente possível mas tornaria a arquitetura desses processadores mais complexa também. Ao invés disso, há várias bibliotecas gráficas que podem ser usadas para simplificar o processo de modelagem de objetos,

Além da forma, cada objeto possui propriedades que são necessárias representar no modelo, como cor, opacidade, rugosidade, etc. Veremos em aulas futuras que algumas dessas propriedades podem ser associadas a superfícies, como texturas, ou ainda associadas a cada vértice, como cor.

2.4 Câmera ou Observador

A criação de filmes de animação requer a geração de desenhos para cada quadro. Por exemplo, cerca de 750 artistas participaram da produção do filme "Branca de Neve e os Sete Anões", dos estúdios da Disney. Em um período de 3 anos o grupo criou mais de 2 milhões de rascunhos até a versão final do filme, lançado em 1937, e que era composto por mais de 250 mil imagens distintas.

A geração de imagens por computador considera uma cena 3D, previamente modelada, e o posicionamento de uma câmera virtual, como se quiséssemos tirar uma foto da cena. Além da posição e orientação dessa câmera virtual, a imagem resultante depende ainda de outros parâmetros, como resolução da imagem, nível de zoom da lente, abertura da lente etc. A animação é feita alterando esses parâmetros que controlam a câmera, e também atualizando o estado dos objetos que constituem a cena.

Vamos utilizar o modelo de câmera **pinhole** que recebe esse nome (buraco de agulha) por poder ser construída a partir de uma caixa que tem um *buraquinho* por onde a luz entra e sensibiliza o filme fotográfico ou sensor eletrônico. Esse é um modelo simples de câmera que não possui lente e que nos permite modelar a projeção dos objetos no plano da imagem. A Figura Fig. 2.3 ilustra as principais características de uma câmera pinhole.

Considere a caixa como um paralelepípedo com um pequeno buraco em uma das faces por onde a luz penetra na caixa. O buraco define o ponto chamado de **Centro de Projeção** da câmera. O **eixo ótico** da câmera é definido pelo vetor que passa pelo buraco e é perpendicular ao plano da imagem. O plano da imagem contém a face oposta à face que contém o centro de projeção. Na prática, o filme fotográfico ou sensor eletrônico é colocado nesse plano para receber a luz que entra na câmera.

A altura de um ponto (ou objeto) distante d do centro de projeção é medida perpendicularmente ao eixo ótico. A figura mostra um objetos de altura y e a imagem desse objeto $y' = y * f/d$. A distância do centro de projeção ao plano da imagem é chamado de comprimento focal (f) da câmera.

Outro parâmetro importante de uma câmera é o campo de visão (*field of view* ou FoV) que pode ser definido pelas dimensões da caixa. Como o sensor em geral é retangular, podemos definir um FoV vertical e outro horizontal. Assim, para um sensor de largura W e altura H o FoV vertical pode ser calculado como: $FoV_v = 2atan(H/(2 * f))$.

Por fim, a razão W/H da câmera (ou sensor ou ainda da imagem) define a sua **razão de aspecto** (*aspect ratio*).

Como a câmera pinhole não usa lentes, ela possui campo de profundidade (**depth of field**) infinito, ou seja, todos os pixels da imagem ficam "em foco". No entanto, isso só é possível pois, como o buraco é muito pequeno, apenas raios de luz provenientes de uma única direção estimulam um ponto na imagem, o que pode ser uma desvantagem

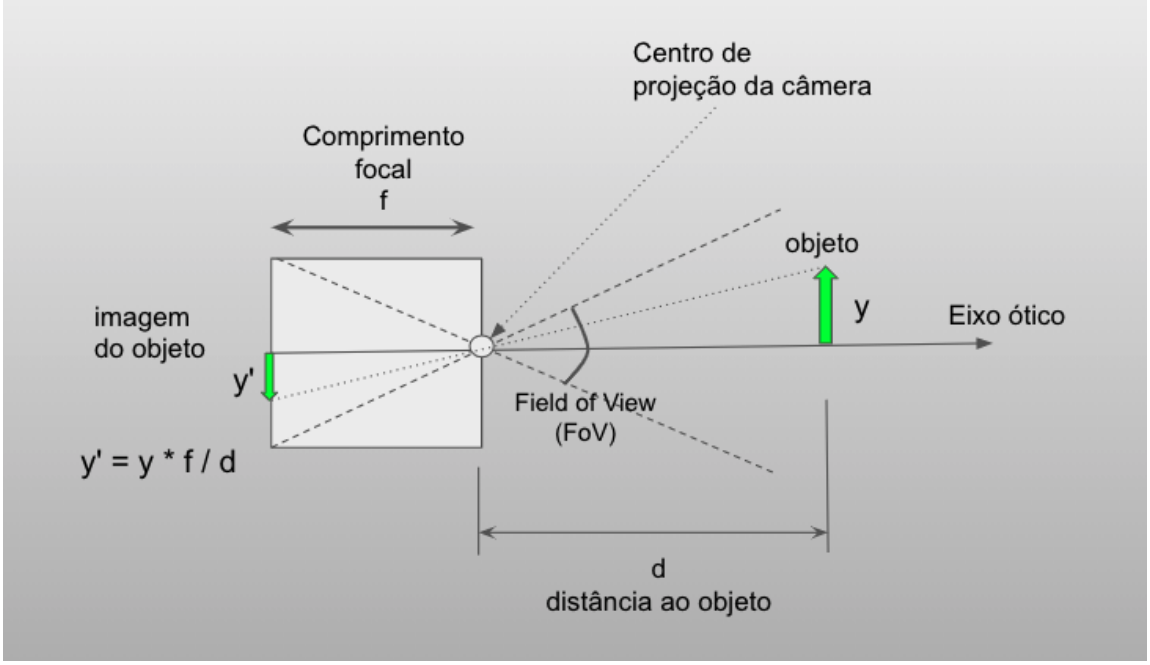

Fig. 2.3: Câmera pinhole

pois pouca luz penetra na câmera. Outra desvantagem é que, para modificar o nível de zoom, é necessário modificar fisicamente a distância da imagem ao buraco (comprimento focal), o que modifica também o FoV.

É importante você conhecer esse modelo e conceitos para entender, em aulas futuras, como representar a projeção na forma de uma transformação linear e obter a imagem dos objetos.

2.5 Representação de imagens

A forma mais comum de representar uma imagem é por meio de uma matriz de pixels, e é conhecida como **imagem raster**. O número de pixels na matriz define a **resolução** da imagem. Por exemplo, a resolução padrão *full HD* usada para monitores e TVs corresponde a 1920×1080 pixels. Quando trabalhamos com imagens digitais, a primeira dimensão costuma indicar o número de colunas (1920) e a segunda dimensão o número de linhas (1080) da imagem. Observe no entanto que, na álgebra linear e na computação, as coordenadas de um elemento de uma matriz são dadas na ordem (linha,coluna), o que pode causar alguma confusão mais tarde.

Além da resolução, cada pixel de uma imagem tem uma certa **profundidade**, como ilustrado na Figura Fig. 2.4. A profundidade de uma **imagem binária** é um bit. Usando apenas 1 bit que assume valores 0 ou 1, podemos representar que um pixel esteja apenas ligado ou desligado (ou ainda acesso ou apagado, branco ou preto, etc.). Uma **imagem em níveis de cinza** tipicamente possui 8 bits (que equivale a 1 byte) de profundidade. Com isso, é possível representar $2^8 = 256$ níveis de cinza. Mais bits podem ser utilizados para representar imagens com grande variação dinâmica entre o pixel mais claro e o mais escuro.

Uma **imagem colorida** é tipicamente representada usando 3 ou 4 bytes por pixel. Na representação **RGB** cada pixel é representando usando as 3 cores primárias: vermelho (*red*), verde (*green*) e azul (*blue*). Um byte é utilizado para quantificar a contribuição de cada uma dessas cores, no total de $3 \times 8 = 24$ bits.

O método escolhido para representar a cor também depende das características do dispositivo de saída gráfica e da aplicação. Por exemplo, monitores combinam as componentes RGB provenientes de fontes de luz de forma aditiva, enquanto os pigmentos usados em impressoras são combinados de forma subtrativa. Em muitos sistemas gráficos, é comum adicionar um quarto componente, às vezes chamado de alfa e denotado por A (em RGBA). Esse componente é usado para obter vários efeitos especiais, mais comumente para definir a opacidade de uma cor. Discutiremos essa e outras representações em aulas futuras.

Fig. 2.4: Imagem binária com profundidade de 1 bit, imagem com 256 níveis de cinza com profundidade 8 bits e imagem colorida RGB com profundidade 24 bits (8 bits por canal de cor).

2.6 Onde estamos e para onde vamos?

Compreender o processo de formação de imagens usando uma câmera virtual é muito importante para entender os próximos tópicos que serão apresentados no curso. Na próxima aula, vamos apresentar mais detalhes sobre sistemas gráficos e a arquitetura pipeline dominante nas unidades de processamento gráfico modernas. Ao final do processo, nossos programas gráficos vão produzir imagens raster.

2.7 Exercícios

1. Uma forma conhecida como `ray-tracing` (tracejamento de raios) para gerar imagens é seguir cada raio que chega a imagem até atingir um objeto e, dependendo das propriedades do material nesse ponto, verificar se ponto é iluminado por alguma fonte de luz para determinar a sua cor. Quais as vantagens e desvantagens de usar `ray-tracing` em relação ao uso de câmeras virtuais usadas pela maioria dos pipelines gráficos?
2. Considere um sensor de resolução 640×480 pixels. O sensor ocupa toda a face posterior de uma câmera pinhole. O centro de projeção da câmera está no centro da face oposta. Seja o comprimento focal 320 pixels. Nesse caso, qual o FoV vertical? E o FoV horizontal?
3. Discuta em que condições a seguinte afirmação pode ser considerada verdadeira ou falsa: *A câmera pinhole é um bom modelo para o olho humano*.

CAPÍTULO 3

Sistemas Gráficos

Nessa aula vamos apresentar alguns recursos típicos encontrados em sistemas gráficos usados para gerar imagens por computador. Recomendamos a seguinte leitura para complementar essas notas.

- Capítulo 1 do livro "Interactive Computer Graphics" de Edward Angel; e/ou:
  - Capítulo 1 do Livro do David Eck.

3.1 Componente de um sistema gráfico

Um sistema gráfico computacional possui os elementos típicos de um computador de mesa ou portátil comum, como ilustrado na Figura Fig. 3.1, ou seja, possui dispositivos de entrada (como teclado, mouse e joystick) e saída (como monitor, impressora e plotter), um processador central (CPU), um processador gráfico (GPU) e memória.

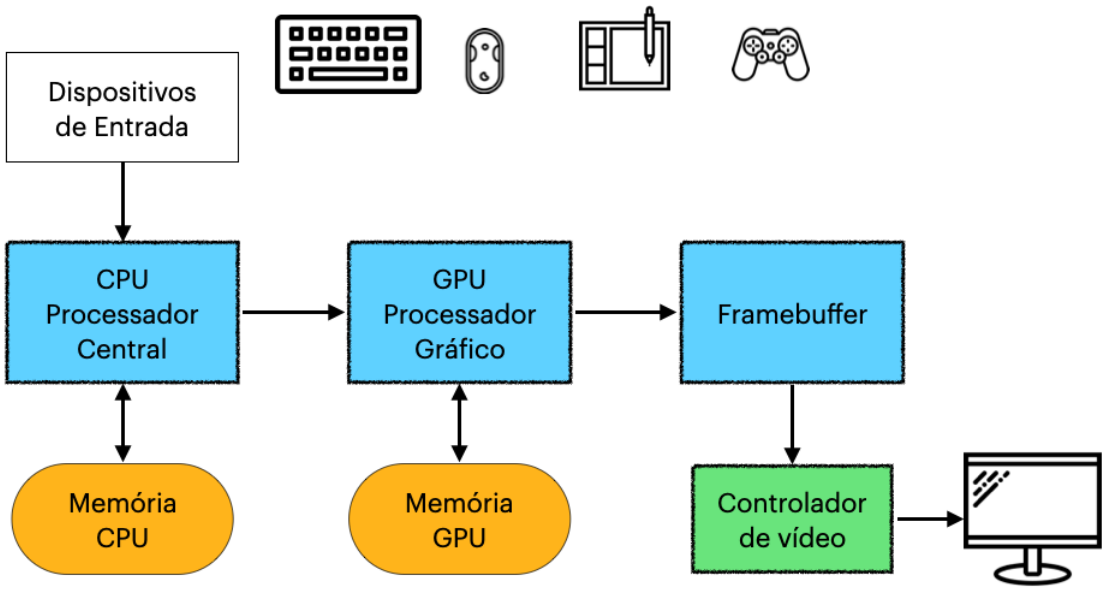

Fig. 3.1: Elementos de um sistema gráfico.

3.2 Dispositivos de entrada e saída

Os dispositivos de entrada, como teclado e mouse, são necessários para que o usuário possa interagir com computadores usando **interfaces gráficas** do tipo WIMP (*Windows, Icons, Menus, and Pointg*), ou seja, interfaces formadas por janelas, ícones, menus e um cursor para apontamento dos elementos gráficos. Outros dispositivos, como mesas digitalizadoras e controles para jogos, são comuns em muitos aplicativos gráficos interativos, para criação de desenhos e edição de imagens.

A saída gráfica pode ser feita por meio de impressoras e plotters, mas o dispositivo mais comum são os monitores de vídeo por permitirem interação em tempo real e a visualização de animações. Os monitores mais utilizados hoje são do tipo **raster**, ou seja, que representam informações gráficas por meio de uma matriz de pixels. Cada monitor pode utilizar uma tecnologia distinta na sua construção e funcionamento. Os tipos mais comuns de monitores são:

- **Tubo de raios catódicos**: consistem em uma tela com revestimento de fósforo, que permite que cada pixel seja iluminado momentaneamente ao ser atingido por um feixe de elétrons. Um pixel está iluminado (branco) ou não (preto). O nível de intensidade pode ser variado para atingir valores de cinza arbitrários. Como o fósforo mantém sua cor apenas brevemente, a imagem é repetidamente digitalizada, a uma taxa de 25 a 30 vezes por segundo. Essa tecnologia é mais antiga e não é mais utilizada devido ao seu grande peso, volume e alto consumo de energia quando comparados aos monitores LCD e LED.
- **Telas de cristal líquido (LCD's)**: usa um campo elétrico para alterar a polarização das moléculas cristalinas em cada pixel. A luz que brilha através do pixel já está polarizada em alguma direção. Alterando a polarização do pixel, é possível variar a quantidade de luz que atravessa o cristal, controlando assim sua intensidade.
- **Diodos emissores de luz (LED ou OLED)**: cada pixel é iluminado por um diodo emissor de luz (ou conjunto tricromático RGB correspondente). Como esses monitores não precisam de iluminação de fundo como o LCD, eles tendem a ser mais eficientes, simples de fabricar e mais finos (tanto que podem ser colocados sobre membranas flexíveis e enroláveis).

A **resolução** mais comum para monitores atualmente é chamada de "full HD", que corresponde a 1920×1080 pixels. Resoluções cada vez maiores estão se tornando comuns também, como monitores 4K (ou "ultra HD") por exemplo, que possuem 4 vezes mais pixels que um monitor full HD.

3.3 Framebuffer

Em sistemas mais simples, o framebuffer armazena a imagem a ser exibida no monitor. O **Controlador de Vídeo** é um módulo responsável para ler a imagem do framebuffer e gerar o sinal de vídeo para o monitor.

Para exibir uma imagem RGB full HD é necessário reservar um framebuffer com cerca de 6 MB (mega bytes). Essa quantidade de memória era indisponível em computadores mais antigos e por isso várias alternativas foram desenvolvidas para contornar essa limitação.

Uma forma para economizar memória em sistemas gráficos é reduzir a profundidade das imagens utilizadas, por exemplo, utilizando uma tabela ou palete de cores para imagens coloridas. Imagine, por exemplo, que nossas imagens possam ser representadas por um conjunto de 256 cores de 24 bits cada cor. Essas cores podem ser codificadas em uma tabela com 256 entradas, onde cada entrada corresponde a uma cor de 24 bits. Esses 256 códigos correspondem aos índices das linhas da tabela que precisam de 8 bits para serem codificadas. Esse código de 1 byte pode ser colocado em cada pixel, ao invés dos 3 bytes da cor, economizando assim 2/3 (dois terços) da memória necessária para armazenar a imagem original com 24 bits por pixel. Nesse caso, a imagem full HD pode ser agora representada com cerca de 2 MB, além da memória necessária para armazenar a tabela com os códigos das cores.

Uma alternativa capaz de reduzir ainda mais o consumo de memória é o uso de **imagens vetoriais** ao invés de imagens raster. Imagens vetoriais são apropriadas para representar desenhos formados por segmentos de linha (ou simplesmente linhas), onde um segmento é representado por dois pontos na tela do monitor. Assim, um desenho formado por centenas ou até mesmo milhares de linhas pode ser representado de forma bem mais compacta (em termos de memória) que uma imagem raster. Além de serem facilmente escaláveis, esses desenhos também podem ser exibidos de forma relativamente simples em monitores analógicos que utilizam tubo de raios catódicos, fazendo o feixe de elétrons do tubo varrer a lista de segmentos de forma contínua.

Observe que, para desenhar uma linha em monitores raster a partir de 2 pontos, é necessário calcular a posição de cada pixel interior ao segmento. **Algoritmos de rasterização** servem para calcular esses pixeis para a geração da imagem raster.

Em sistemas mais complexos, o framebuffer pode armazenar outras informações também, como profundidade ou opacidade na representação RGBA. Além disso, com mais memória disponível, não é mais necessário limitar cada cor a 256 níveis por exemplo. Framebuffers modernos permitem representar cada pixel usando 12 ou mais bits por cor, permitindo a renderização de imagens com larga faixa dinâmica (HDR ou *high dynamic range*).

3.4 Processadores e Memória

Em sistemas mais simples, o processador gráfico ou GPU (*Graphics Processing Unit*) fica integrado ao processador central ou CPU (*Central Processing Unit*). Em sistemas mais complexos o processamento gráfico fica em um circuito ou placa dedicada, com memória própria. O framebuffer faz parte da memória dedicada ao processamento gráfico e por isso faz parte da memória utilizada pela GPU ou, em sistemas sem GPU, como parte da memória do sistema.

Embora a maioria dos monitores trabalhem com frequências de 60 Hz ou mais, a taxa de geração de imagens em animações costuma ser bem inferior. No entanto, para manter uma sensação de continuidade, uma animação precisa manter uma taxa de aproximadamente 15 novos quadros por segundo (fps – *frames per second*). Abaixo de 10 fps, a sensação de descontinuidade da animação começa a ser notada pela maioria das pessoas. Para garantir uma boa qualidade, os projetores antigos nos cinemas exibiam filmes a 24 fps e na TV a 30 fps.

Chamamos de **renderização** o processo de geração dos pixels (posição e cor) no framebuffer a partir de um modelo de cena (representação geométrica dos objetos). Muito desse processo ocorre atualmente dentro da GPU que, para obter um alto desempenho, utilizam uma arquitetura paralela conhecida como pipeline gráfico. A Figura Fig. 3.2 ilustra os principais módulos de um pipeline gráfico.

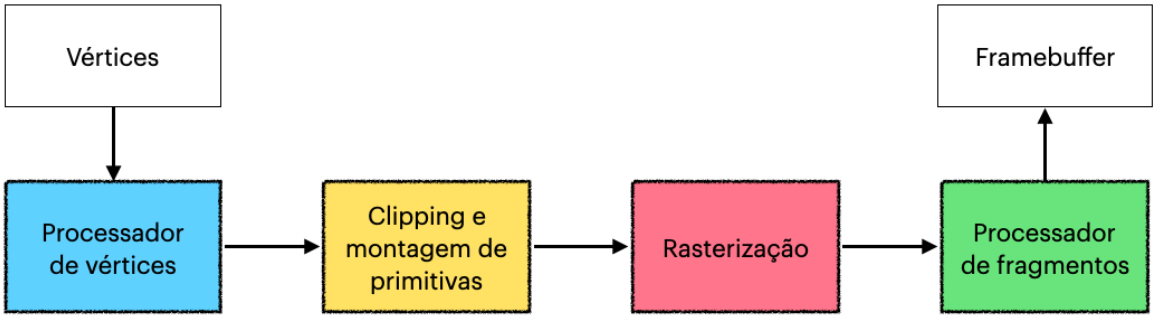

Fig. 3.2: Estágios do pipeline gráfico.

Para executar um comando simples como "desenhe um triângulo em 3D", o pipeline primeiramente recebe o conjunto de vértices que definem o triângulo. O **processador de vértices** usa os parâmetros da câmera virtual para transformar as coordenadas dos vértices para um sistema centrado na câmera. As transformações são representadas por matrizes e a mudança de coordenadas envolve a multiplicação dessas matrizes.

Após a transformação de coordenadas, alguns vértices podem ficar fora do campo visual da câmera. O módulo de **clipping** (recorte) remove esses vértices e, caso parte de segmentos ainda sejam visíveis, esses segmentos são redefinidos na forma de novos vértices.

Os segmentos visíveis são passados para o módulo de **rasterização**, que usa os vértices visíveis para pintar os pixels da parte visível do triângulo, que são chamados de **fragmentos**. Observe que a cena pode ser composta por outros triângulos que geram outros fragmentos. Esses fragmentos são combinados pelo **processador de fragmentos** para calcular a cor final de cada pixel do framebuffer.

3.5 Pipelines programáveis

As placas gráficas mais antigas possuíam um pipeline rígido com poucos recursos. As placas mais recentes permitem programar o processador de vértices e o processador de fragmentos. Com isso podemos implementar vários efeitos em tempo real que não era possível com as placas mais antigas (como *bump mapping*). Chamamos de **shaders** esses programas que são executados pela GPU.

- **Vertex shaders**: executado pelo processador de vértices e usado para aplicar transformações nos vértices e nas suas propriedades;
- **Fragment shaders**: Esse shader é usado para processar fragmentos, permitindo aplicar texturas e outros efeitos de iluminação e computações específicas para cada tipo de fragmento.

Nas placas gráficas mais antigas era também comum avaliar o desempenho das placas pelo número de primitivas geométricas processadas por segundo (**front end**) ou então pela taxa de bits que podiam ser movidos para o framebuffer (**back end**). Hoje as GPUs mais poderosas podem usar ponto flutuante em todo o pipeline, até no framebuffer, um recurso útil para representar e manipular imagens com grande amplitude dinâmica (HDR – *high dynamic range*). Essas GPUs são chamadas de **unified shading engines** pois fazem o processamento dos vértices e fragmentos concorrentemente. Hoje, esse grande poder computacional é aproveitado por muitas aplicações não relacionadas à geração de imagens.

3.6 Bibliotecas gráficas

Mesmo sendo muito eficiente, programar o pipeline para criar 30 imagens por segundo necessárias para gerar boas animações ainda é um desafio.

Dizemos que cada redesenho faz parte de um **ciclo de atualização** (*refresh cycle*) pois o programa deve atualizar o conteúdo da imagem. O programa se utiliza de uma biblioteca ou API (*application programming interface*) para se comunicar com o sistema gráfico. Existem várias APIs diferentes usadas em sistemas gráficos modernos, cada uma fornecendo alguns recursos distintos em relação às outras. De um modo geral, as APIs gráficas podem ser classificadas em duas classes gerais:

- **Modo Retido** (*retained mode*): A biblioteca mantém o estado da computação em suas próprias estruturas de dados internas. A cada ciclo de atualização, esses dados são transmitidos à GPU para renderização.
  - Por conhecer o estado completo da cena, a biblioteca pode realizar otimizações globais automaticamente.
  - Este método é menos adequado para conjuntos de dados que variam no tempo, uma vez que a representação interna dos dados precisa ser atualizada com frequência.
  - Isso é funcionalmente análogo à compilação de um programa.
  - Exemplos: Java3d, Ogre, Open Scenegraph.
- **Modo Imediato** (*immediate mode*): O aplicativo fornece todas as primitivas a cada ciclo de exibição. Em outras palavras, seu programa transmite comandos diretamente para a GPU para serem executados.
  - A biblioteca só pode realizar otimizações locais, pois não conhece o estado global. É responsabilidade do programa do usuário realizar otimizações globais.
  - Isso é adequado para cenas altamente dinâmicas.
  - Isso é funcionalmente análogo à interpretação do programa.
  - Exemplos: OpenGL, DirectX.

3.7 OpenGL

O OpenGL se tornou uma API padrão largamente utilizada. Ela está disponível em praticamente todas as plataformas e pode ser acessada pelas mais utilizadas linguagem de programação como C/C++, Java e Python. Para que posso funcionar em plataformas tão diferentes, ela precisa ser genérica, em contraste por exemplo com o DirectX, que foi desenvolvido para funcionar principalmente em sistemas da Microsoft.

Para facilitar essa generalidade e se tornar o OpenGL independente do sistema operacional e do sistema que controla as janelas da interface, o OpenGL não fornece recursos para entrada e saída para interação com usuárias e usuários.

O OpenGL trabalha, em sua maior parte, no modo imediato. Isso significa que cada chamada de função resulta no envio de um comando diretamente para a GPU. No entanto, há alguns elementos retidos como, por exemplo, transformações, iluminação e texturização que precisam ser configurados previamente para que possam ser aplicadas no cálculo. Por exemplo, o OpenGL não oferece recursos para criar uma janela, redimensionar uma janela, determinar as coordenadas atuais do mouse ou detectar se uma tecla do teclado foi pressionada. Para atingir esses outros objetivos, é necessário usar um kit de ferramentas adicional. Nesse curso vamos adotar a versão WebGL, que permite criar aplicações gráficas que fazem uso do OpenGL a partir de qualquer navegador moderno.

3.8 Onde estamos e para onde vamos

Nessa aula continuamos a cobrir conceitos da computação gráfica, muitos deles associados ao desenvolvimento histórico dos sistemas gráficos. Com a evolução desses sistemas, temos muito mais recursos computacionais a disposição (tanto de hardwares quanto de software) e GPUs muito poderosas mas otimizadas para tratar de elementos geométricos básicos.

O OpenGL é uma API gráfica genérica que permite o desenvolvimento de aplicativos gráficos para praticamente qualquer plataforma. A versão WebGL permite usar o OpenGL dentro de um navegador. Do ponto de vista pedagógico, isso facilita bastante o ensino pois reduz a necessidade de instalação de ferramentas de desenvolvimento, pois necessitamos basicamente de um editor de texto, e estimula a prática pois os exercícios que você vai desenvolver podem ser executados por qualquer navegador.

Na próxima aula vamos começar a introduzir alguns elementos básicos de HTML e JavaScript para desenvolver nossas primeiras aplicações. Logo em seguida vamos introduzir o WebGL, usando esse ambiente Web de programação com HTML+JavaScript.

3.9 Exercícios

1. Considere um segmento de linha definido por dois pontos no espaço 2D e uma janela de exibição retangular. Mostre que basta redefinir um ou dois desses pontos para recortar a parte visível do segmento, ou seja, a parte do segmento interna à janela de exibição.
2. Filmes de cinema eram tipicamente filmados em películas de 35 mm com uma resolução de aproximadamente 3000×2000 pixels. Que implicação isso resulta quando esses filmes eram exibidos em TVs antigas? E nas modernas?
3. Em um vídeo full HD, cada imagem possui resolução 1920×1080 pixels, com profundidade de 24 bits. Se o vídeo exibe 30 imagens por segundo, qual a frequência necessária para transmitir esse vídeo de forma serial?

CAPÍTULO 4

Desenhando no Canvas HTML com JavaScript

Vamos usar o canvas para gerar desenhos e animações 2D e 3D em uma página web, usando HTML e JavaScript.

Nessa aula vamos introduzir as ferramentas de programação que vamos utilizar para resolver os exercícios. Embora essas ferramentas sejam muito relevantes nessa disciplina, elas não constituem nosso foco principal. Por isso, nessa aula vamos apenas introduzi-las por meio de exemplos ilustrando alguns recursos básicos. Você pode copiar e depois estender esses exemplos para realizar os exercícios.

Para aprender mais detalhes dessas ferramentas recomendamos os seguintes links:

- Tutorial do canvas
- Uma reintrodução ao JavaScript
- Tutorial de HTML

4.1 Esqueleto HTML para usar o canvas

Vamos começar apresentando um esqueleto HTML que cria um canvas (área de desenho) para começar a desenhar em 2D, usando algumas primitivas gráficas comuns.

A sigla HTML significa `HyperText Markup Language`, ou seja, é uma linguagem que permite criar hipertexto como links URL e outros elementos como o próprio canvas usando `tags`.

O documento HTML abre e fecha usando as tags <`html`> e </`html`>. A parte delimitada por <`head`> e </`head`> não é exibida pelo navegador quando a página é carregada. Ela tipicamente contém:

- um título delimitado pelas tags <`title`> </`title`>);
- links para os arquivos CSS usados para configurar os estilos usados pelos elementos da página, com extensão `.css`;
- e outros meta-dados. No nosso caso, deve conter também o arquivo que contém os programas em JavaScript, como o `meu_script.js`. Observe que os arquivos em JavaScript tem a extensão `.js`.

O trecho delimitado por <`body`> </`body`> é a parte que será exibida pelo navegador. Para criar um canvas basta copiar o seguinte trecho de código HTML para um arquivo `index.html`. Observe que no corpo (<`body`>) precisamos incluir uma tag <`canvas`> </`canvas`>, para criar uma área de desenho. Dentro de cada tag você pode definir algumas opções para configurar o elemento HTML. No caso da tag <`canvas`>, o `index.html` cria um elemento canvas com 640×480 (`width` $\times$ `height`) pixels em um sistema de coordenadas com o eixo `x` para a direita e `y` para baixo como:

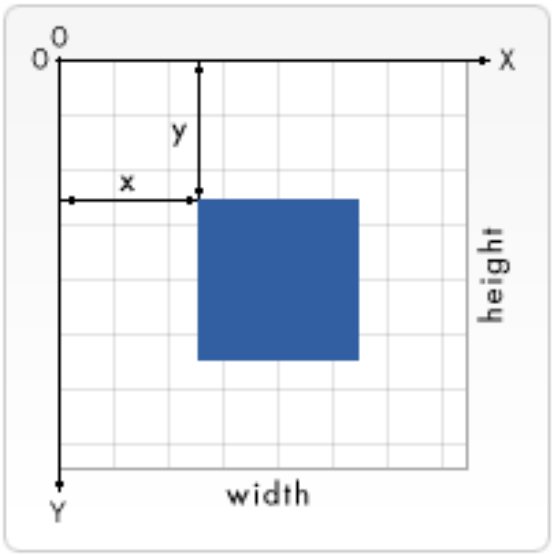

Fig. 4.1: Coordenadas no Canvas (reproduzido de Tutorial do Canvas)

Infelizmente, nem todos os navegadores são compatíveis com o canvas. Para esses navegadores, podemos incluir uma mensagem alternativa entre os tags <canvas> </canvas>, como por exemplo a mensagem "Opa! pode ser que seu navegador não suporte o canvas do HTML5.". Assim, esses navegadores, ao invés do canvas, exibem a mensagem alternativa.

No exemplo incluímos também outros tags HTML para você se divertir ainda mais criando listas, sessões etc.

4.1.1 Exemplo de um esqueleto HTML usando o canvas

```
<html lang="pt-br">
<head>
    <meta charset="utf-8">
    <title>Meu primeiro canvas</title>
    <link rel="stylesheet" href="./webgl.css" type="text/css">
    <script src="meu_script.js" defer></script>
</head>

<body>
    <!--  comentário em html -->
    <h1>Meu primeiro canvas</h1>
    <p>Um parágrafo</p>

    <!-- lista -->
    <ul>
        <li> item um</li>
        <li> item dois</li>
        <li> outro item dois</li>
    </ul>

    <!-- O CANVAS -->
    <canvas id="meucanvas" width="640" height="480">
        Opa! pode ser que seu navegador não suporte o canvas do HTML5.
    </canvas>
```

(continues on next page)

(continued from previous page)

```
    <!-- separador linha horizontal -->
    <hr>

    <h3>tchauzinho...</h3>
</body>
</html>
```

4.2 Usando arquivos CSS para formatar elementos do HTML

Os arquivos de estilo no formato CSS (Cascading Stylesheets) facilitam a formatação de sua página.

Salve o trecho abaixo em um arquivo `webgl.css` na mesma pasta do seu arquivo `index.html`, e brinque um pouco, por exemplo, mudando algumas cores de `black` para `red` ou `green`. Para isso, abra o arquivo com um editor de texto de sua preferência.

A sintaxe do CSS é relativamente simples. Basta, para cada elemento, criar um dicionário de nomes e valores. Assim, para o elemento "canvas", esse arquivo `webgl.css` define algumas de suas propriedades, nesse caso, "border" e "background-color". Para o elemento "h3", o arquivo define "color".

Para saber mais, consulte materiais disponíveis na Internet, como esses tutoriais da Mozilla.

Para a maioria dos nossos exemplos e exercícios, o uso de CSS é facultativo e talvez seja suficiente definir as propriedades desejadas em tags de estilo (style).

4.2.1 Exemplo de um arquivo CSS

```
canvas {
    border: 2px solid black;
    background-color: yellow;
}
h3 {
    color: green;
}
```

4.3 Desenhando no canvas usando JavaSCript

O JavaScript é uma linguagem de script (`scripting language`) criada para a Web. Toda vez que uma página mostra algo "dinâmico" (como updates de tempos em tempos) ou algo "interativo" (como um mapa), seu navegador deve estar executando um script (ou programa) em JavaScript.

O exemplo a seguir mostra o conteúdo do arquivo `meu_script.js` que foi usado no arquivo HTML fornecido anteriormente. Crie um arquivo com esse mesmo nome na mesma pasta dos arquivos anteriores e salve o código abaixo nesse arquivo.

4.3.1 Exemplo: arquivo `meu_script.js`

```
//=====================================================================
/*
    Meu script desenha alguma coisa no canvas
*/
//=====================================================================

// variáveis globais
var ctx;  // contexto de desenho

//=====================================================================
// função principal
function main() {
    // veja o canvas id definido no arquivo index.html
    const canvas = document.getElementById('meucanvas');
    // vamos definir um contexto para desenhar em 2D
    ctx = canvas.getContext('2d');
    if (!ctx) alert("Não consegui abrir o contexto 2d :-( ");

    let cor = 'blue';
    desenhe_rect( cor, 20, 40, 160, 80 );
    cor = 'red';
    desenhe_rect( cor, 100, 60, 340, 280 );
    cor = 'green';
    desenhe_rect( cor, 320, 240, 300, 220 );
};

//=====================================================================
// outras funções
function desenhe_rect( cor, left, top, width, height ) {
    // recebe uma cor e os parâmetros de um retângulo.
    // desenha a região interna do retângulo com cor.
    console.log("Desenhando retângulo ", cor)
    ctx.fillStyle = cor;
    ctx.fillRect( left, top, width, height );
};

//=====================================================================
// Chamada da função principal, executada quando o arquivo é carregado
// em JavaScript.
main();
```

4.3.2 Discussão

Nesse e em futuros exemplos vamos adotar um esqueleto e sintaxe que se assemelham a programas em C e Python. Além de facilitar o entendimento desses exemplos por programadores que dominam outras linguagens, esperamos com isso facilitar também sua transição para o JavaScript. Dessa forma, nossos exemplos vão conter uma função `main()` **sempre** definida no início do arquivo e que é **chamada na última linha** do arquivo. Essa é a **única** função a ser chamada durante o carregamento do arquivo pelo navegador. Todas as demais funções são chamadas a partir da `main()`.

Para definir uma função em JavaScript devemos usar o comando `function`. Suas demais funções podem ser definidas, em qualquer ordem, após a `main()`.

Observe nesse exemplo que, ao programar em JavaScript, devemos também:

- terminar cada comando com um ponto-e-vírgula (`;`);

- indicar o início e fim de cada bloco usando chaves, a menos quando o bloco conter apenas um comando.

Em JavaScript, as variáveis precisam ser declaradas como `var`, `let` ou `const`. As definidas como `var` são **variáveis globais** e as demais são consideradas **locais**, ou seja, seus nomes são válidos apenas dentro da função onde foram definidas. As variáveis do tipo `const` são **constantes**, ou seja, seus valores não devem ser alterados depois de criados.

Depois de criar os três arquivos (HTML, CSS e JavaScript) na mesma pasta, carregue o arquivo `index.html` em um navegador de sua preferência. A Figura Fig. 4.2 exibe o resultado desse nosso primeiro programa em JavaScript em uma janela do navegador Chrome.

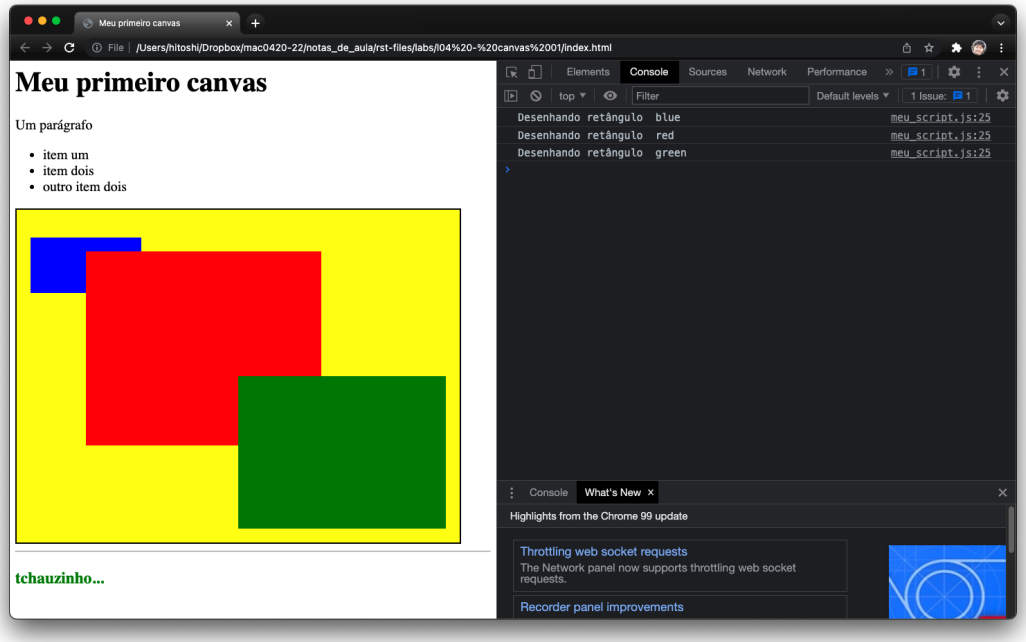

Fig. 4.2: Navegador Chrome exibindo o resultado do programa JavaScript. O canvas é exibido na metade esquerda e, à direita, está o console do JavaScript.

Aproveite também para abrir o console do JavaScript clicando, a partir da barra de menu do Chrome, em `View -> Developer -> JavaScript Console`. O console está também disponível em outros navegadores modernos, como o Firefox, Edge e Safari, e pode lhe ser muito útil para depurar os seus programas, por exemplo, exibindo mensagens usando o comando `console.log()` como mostrado na função `desenhe_rect()`. Antes porém de explicar o desenho, vamos entender melhor o comportamento da `main()`.

4.3.3 O que faz a função `main()`?

As páginas da Web são estruturadas como objetos DOM (*Document Object Model* – modelo de objeto de documento), permitindo que os programas em JavaScript (ou em outra linguagem) possam ter acesso aos elementos da página. Esse documento é carregado automaticamente e o `canvas` pode então ser obtido pelo método `document.getElementById()`, usando o `id` definido no arquivo `index.html`.

Associada ao canvas podemos fazer uso de APIs gráficas distintas para desenho, também chamadas de contextos. Nesse exemplo, a variável global `ctx` recebe um contexto para desenhos 2D. Em aulas futuras, vamos fazer uso do contexto `webgl2`, a API gráfica para desenhar cenas 3D usando o WebGL. Observe que, por ter sido declarada como global, `ctx` pode ser utilizada dentro da função `desenhe_rect()` diretamente, ao invés de passá-la como argumento da função. Como é possível que o contexto que você deseje utilizar não esteja disponível no navegador utilizado, é recomendado testá-lo antes de prosseguir. O programa envia um alerta caso o contexto não esteja disponível.

A função `main()` chama a função `desenhe_rect()` 3 vezes. Cada vez que a função é chamada, uma mensagem é impressa no `console.log` indicando a cor do retângulo a ser pintado.

Antes de desenhar algum elemento gráfico, como uma linha ou um retângulo, é necessário definir todas as propriedades desejadas do elemento. Nesse exemplo, apenas a cor usada para preenchimento foi definida no atributo `fillStyle` do contexto, antes de desenhar cada retângulo usando o método `fillRect()`. Métodos identificados como `fill` incluem o interior da região (no caso um retângulo), enquanto métodos identificados como `stroke` apenas desenham o contorno da região.

Note: Modelo de plotter com caneta

Esse comportamento, de definir as condições (ou estado) das ferramentas de desenho, como cor do pincel ou lápis, grossura do pincel, tipo da linha (se contínua ou tracejada), etc. **antes** de desenhar é tipico de ferramentas de desenho 2D tradicionais conhecida como modelo de **plotter com caneta** (*Pen Plotter*). Os plotters são dispositivos ainda bastante utilizados para impressão de folhas grandes, usadas por exemplo para exibir a planta de uma casa ou edifício. Nesse modelo, devemos mover a caneta para uma determinada posição, baixar a caneta, e arrasta-la (move-la) para outras posições, desenhando as linhas que compoem o desenho. Note também a adequação desse modelo para desenhar imagens vetoriais.

Experimente desenhar apenas o contorno do retângulo, ao invés de preencher seu interior, usando o método `strokeRect()` (experimente, modificando o `meu_script.js` e recarregando o `index.html`!).

O contexto 2D possui recursos simples para desenhar retângulos e outros figuras formadas por linhas. Permite também desenhar texto e imagens raster (carregadas de um arquivo, por exemplo).

4.4 Arrays no JavaScript

Arrays no JavaScript são estruturas sequenciais dinâmicas, muito semelhante ao tipo `list` do Python, ou seja, um mesmo `array` pode conter elementos de tipos distintos, que podem ser inseridos e removidos dinamicamente usando os métodos `push()` e `pop()`. O seguinte exemplo ilustra o uso de array para a definição de pontos que definem um polígono.

```
/*
    Esse script mostra mais recursos do canvas e da JS
*/

// variáveis globais
var ctx;  // contexto de desenho

//================================================================
// função principal
function main() {
    // veja o canvas id definido no arquivo index.html
    const canvas = document.getElementById('meucanvas');
    // vamos definir um contexto para desenhar em 2D
    ctx = canvas.getContext('2d');
    if (!ctx) alert("Não consegui abrir o contexto 2d :-( ");

    let pontos = [ [20, 40], [180, 120], [180, 40], [20, 120] ];
    desenhe_poligono( pontos );

    pontos = []  // array vazio
    pontos.push( [100, 60] ); // coloca um ponto
    pontos.push( [440, 60] ); // experimente alterar
    pontos.push( [440, 340] ); // a ordem para ver o que acontece!
    desenhe_poligono( pontos, 'red' );

    pontos.push( [100, 340] ); // descomente essa linha
    desenhe_poligono( pontos, 'black', 2 );
```

(continues on next page)

(continued from previous page)

```
    cor = 'green';
    pontos = [ [320, 240], [620, 240] ] // array não começa vazio
    pontos.push( [320, 460] );
    pontos.push( [620, 460] );
    desenhe_poligono( pontos, 'green', 0 );

    desenhe_texto("Exemplo de stroke e fill", 120, 420, 36, 'magenta' );
};

//=====================================================================
// outras funções
function desenhe_poligono( pts, cor='blue', wid = 10 ) {
    // recebe um array de pontos no canvas descrevendo
    // um polígono, uma cor e a uma largura wid da linha
    // usada para desenhar o contorno.
    // Caso wid=0, a região interna é preenchida com a cor.
    let tam = pts.length;
    console.log("Desenhando poligono", cor, pts, tam);

    let poli = new Path2D();
    poli.moveTo( pts[0][0], pts[0][1] );
    for (let i = 1; i < pts.length; i++) {
        poli.lineTo( pts[i][0], pts[i][1] );
        console.log( pts[i][0], pts[i][1]  );
    }
    poli.closePath(); // cria um contorno fechado.

    if (wid > 0) {
        ctx.strokeStyle = cor;
        ctx.lineWidth = wid;
        ctx.stroke( poli );
    }
    else { // wid <= 0 preenche o polígono
        ctx.fillStyle=cor;
        ctx.fill( poli );
    }
}

// ---------------------------------------------------------------------
function desenhe_texto (msg, x, y, tam=24, cor = 'black') {
    // recebe uma String msg e uma posicao (x, y) e
    // desenha o texto msg na posicao (x,y)
    ctx.fillStyle = cor;
    ctx.font = `${tam}px serif`;
    ctx.fillText(msg, x, y);
}

//=====================================================================
// Chamada da função principal, executada quando o arquivo é carregado
// em JavaScript.
main();
```

4.4.1 Discussão

A função `main()` desse exemplo ilustra como criar e acessar os elementos de um array `pontos`. Você pode substituir o conteúdo do arquivo `meu_script.js` com esse novo programa, e recarregar o `index.html` em seu navegador, para ver o resultado, que está ilustrado na Figura Fig. 4.3.

Um array pode ser criado com todos os valores entre colchetes (`[ ]`). O atributo `length` contém o tamanho do array. Um segundo polígono (triângulo em vermelho) é criado usando um array inicialmente vazio e cada elemento é inserido em seguida usando o método `push()`. Observe que o triângulo se torna um retângulo (em preto) ao inserir mais um canto. Observe também que a ordem dos pontos no array é importante na definição das linhas que formam o objeto.

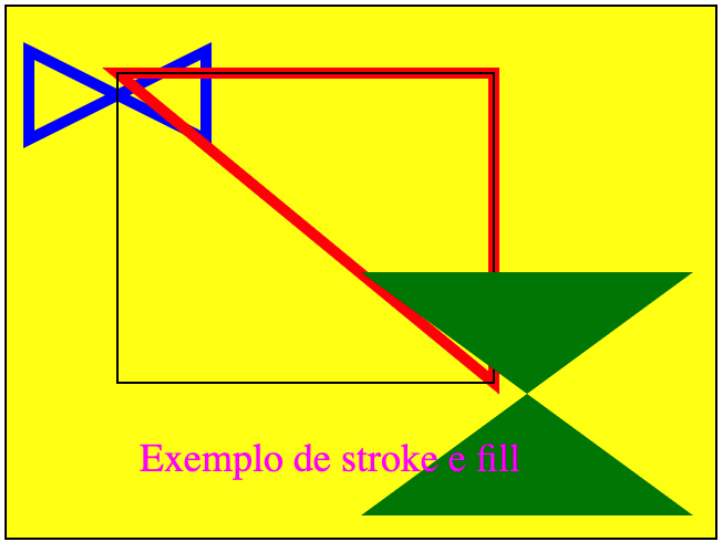

Fig. 4.3: Resultado do novo script usando arrays de pontos para desenhar polígonos.

Esse exemplo explora também o polimorfismo da função `desenhe_poligono()`, chamando a função com e sem alguns dos parâmetros opcionais. No caso, os valores default de `cor` e `wid` são `blue` e `10`, respectivamente.

A função `desenhe_poligono()` recebe um array de pontos e define um `path2D`, um objeto no contexto 2d representado por uma sequência de linhas, também chamado de polilinhas (*polylines*). O método `closePath()` é usado para criar um **contorno fechado**, ou seja, o último ponto é conectado ao primeiro. Sem a chamada desse método, o contorno fica **aberto**. Experimente comentar a chamada desse método para ver o que acontece com o desenho.

> Essas linhas (contorno do polígono) são desenhadas usando o método `stroke()` com a cor e a largura de linha passados como argumentos da função. No entanto, quando o parâmetro `wid` tem valor 0, a região interna é preenchida com a `cor` usando o método `fill()`.

Além do uso de arrays, há vários detalhes também ilustrados nesse exemplo, como o uso dos comandos `if-else` e `for`, o uso de parâmetros opcionais na função `desenhe_poligono` e o uso de `stroke` no canvas.

Outro recurso interessante oferecido pelo canvas é a possibilidade de desenhar textos. A função `desenhe_texto()` mostra um exemplo desse recurso.

4.5 Onde estamos e para onde vamos

Começamos a introduzir as ferramentas de programação que vamos utilizar ao longo desse curso. Em particular, vamos adotar um esqueleto e sintaxe simples (ou semelhante a outras linguagens) para os programas em JavaScript para facilitar a leitura desses programas, e a escrita de futuros programas, por programadores e programadoras familiarizadas com outras linguagens.

Nessa aula vimos como criar desenhos usando o contexto 2d do canvas. Na próxima aula vamos apresentar recursos para criar desenhos interativos e introduzir o conceito de programação baseada em eventos.

4.6 Exercícios

1. Escreva um programa em JavaScript (+ HTML) que desenha, em um canvas de tamanho 300×300, a aproximação de um círculo de raio 100 usando 8 segmentos de reta. Depois disso, altere esse número para 16, 32, 64 segmentos para ver o que acontece com a qualidade visual desse círculo.

2. Escreva um programa em JavaScript (+ HTML) que desenha, em um canvas de tamanho 400×400, 50 elementos aleatórios de uma grade de dimensão 10×10, com uma cor também aleatória.

   Para sortear um inteiro no intervalo `[min, max]`, e uma cor RGB, você pode utilizar as seguintes funções em JS:

```
function sorteie_inteiro (min, max) {
     return Math.floor(Math.random() * (max - min) ) + min;
 }

 function sorteie_corRGB () {
     let r = sorteie_inteiro(0, 255);
     let g = sorteie_inteiro(0, 255);
     let b = sorteie_inteiro(0, 255);
     return `rgb( ${r}, ${g}, ${b} )`;  // retorna uma string
 }
```

   A figura abaixo ilustra 3 exemplos de como deve aparecer seu canvas. Clique em reload várias vezes para gerar novos desenhos.

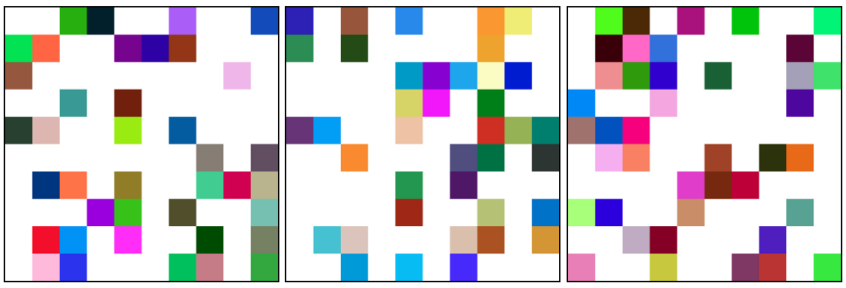

Fig. 4.4: Exemplos de grades 10×10 com aproximadamente 50% de seus elementos pintados de forma aleatória.

3. Escreva um programa em JavaScript (+ HTML) que desenha, em um canvas de tamanho 640×480, 10 quadrados de tamanhos e cores aleatórias, em posições também aleatórias.

CAPÍTULO 5

Índice

- genindex
- modindex
- search