DCC192

2025/1

UFMG

Desenvolvimento de Jogos Digitais

A10: Gráficos 2D

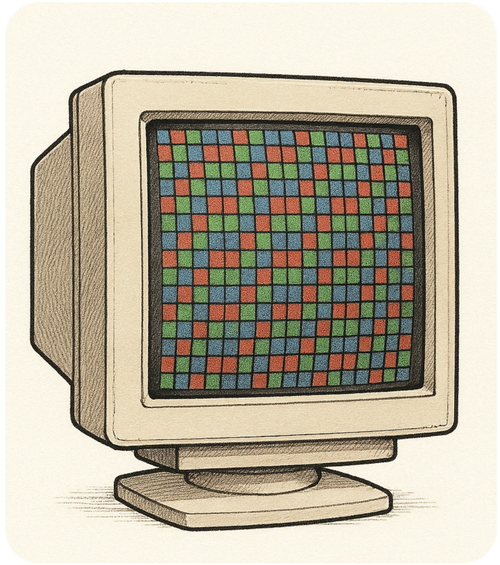
Prof. Lucas N. Ferreira

Plano de aula

- Monitores
- Imagens, cores e transparência
- Desenhando imagens na tela
- Sprites e Spritesheets
- Animações
- Tilemaps
- Rolagem de Câmera
- Efeito de Paralaxe

Monitores

Monitores são dispositivos para mostrar imagens digitais. Para isso, eles possuem uma matriz de pixels atualizada a uma determinada frequência:



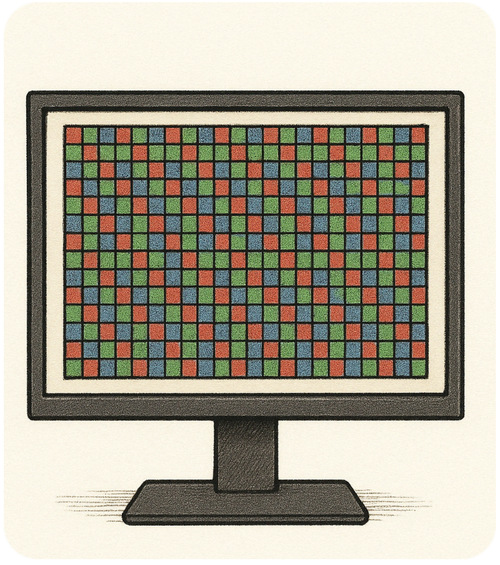
Resoluções típicas:

- 640x480
- 800x600
- 1024x768
- 1280 x 1024

Taxa de atualização:

- De 60Hz a 85Hz

Monitores CRT



Resoluções típicas:

- 1280x720
- 1920x1080
- 2560x1440
- 3840x2160

Taxa de atualização:

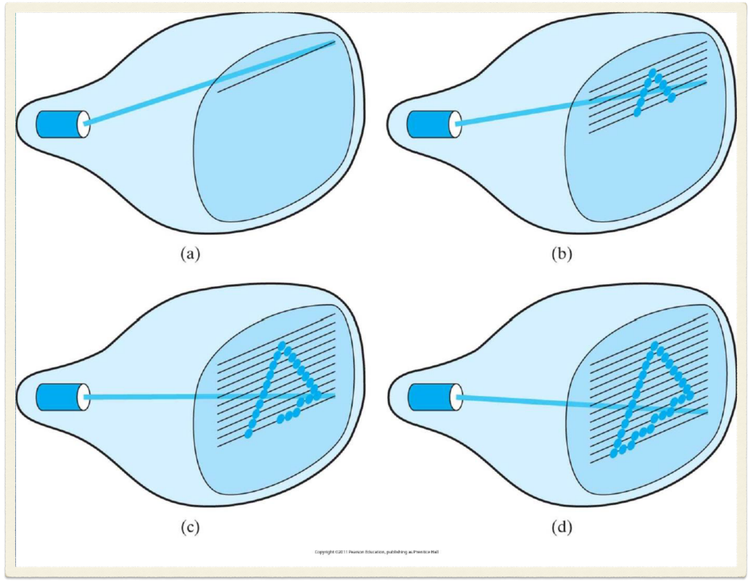
- 60Hz, 144Hz, 240Hz, ...

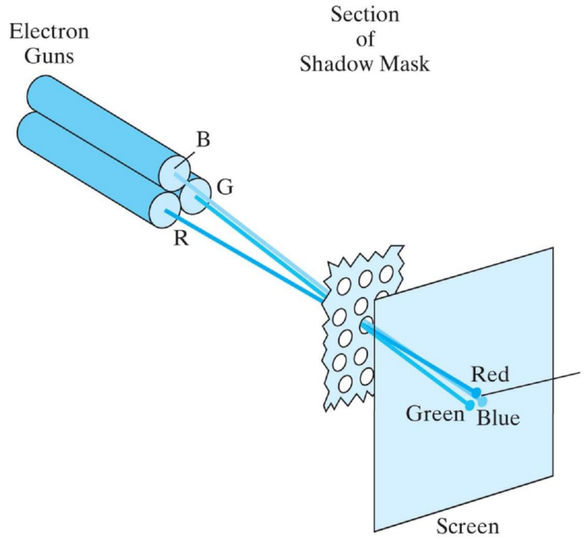
Monitores LCD/LED/OLED

A principal diferença entre as tecnologias de monitores é como elas atualizam essa matriz de pixels.

Monitores CRT (Cathodic Ray Tube)

Monitores CRT antigos desenham pixels por meio de um canhão de elétrons que dispara feixes em uma tela revestida de fósforo.

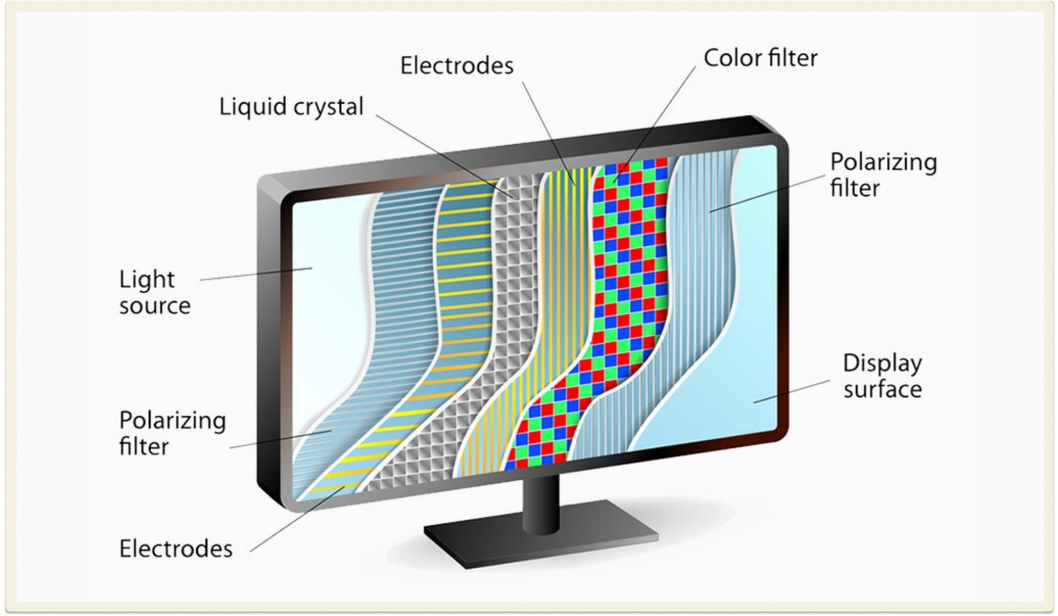




Atualizar um quadro inteiro não é um processo instantâneo, pois o canhão precisa varrer todas as linhas!

Monitores LED (Cathodic Ray Tube)

Monitores LED desenham pixels por meio de 3 tipos de camadas:

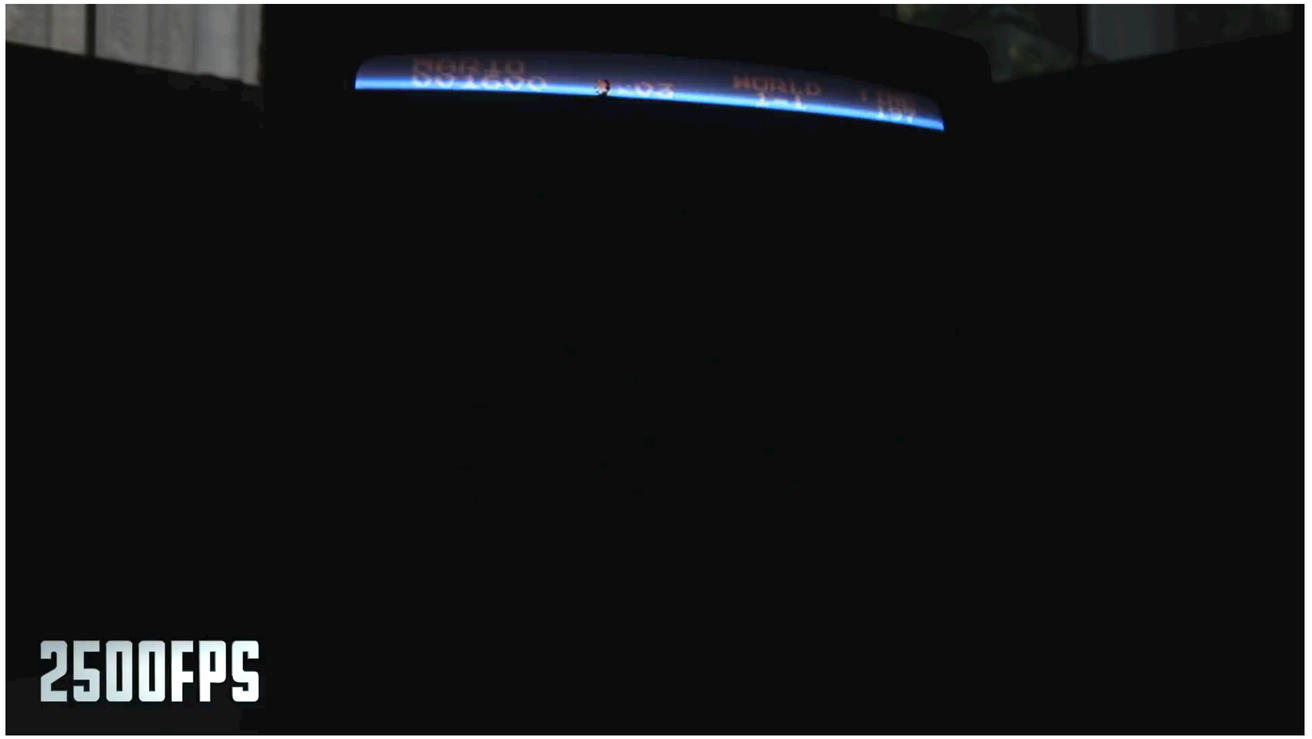


1. **Backlight**: Lâmpadas de LED que emitem luz branca
2. **Crital Líquido (LCD):** Controla o quanto da luz branca passa por cada pixel.
3. **Filtros de polarização e difusão**: Ajudam a orientar e espalhar a luz corretamente.

Atualizar um quadro inteiro não é um processo instantâneo, pois cada pixel do painel precisa mudar de estado!

Monitores CRT vs LCD/LED/OLED

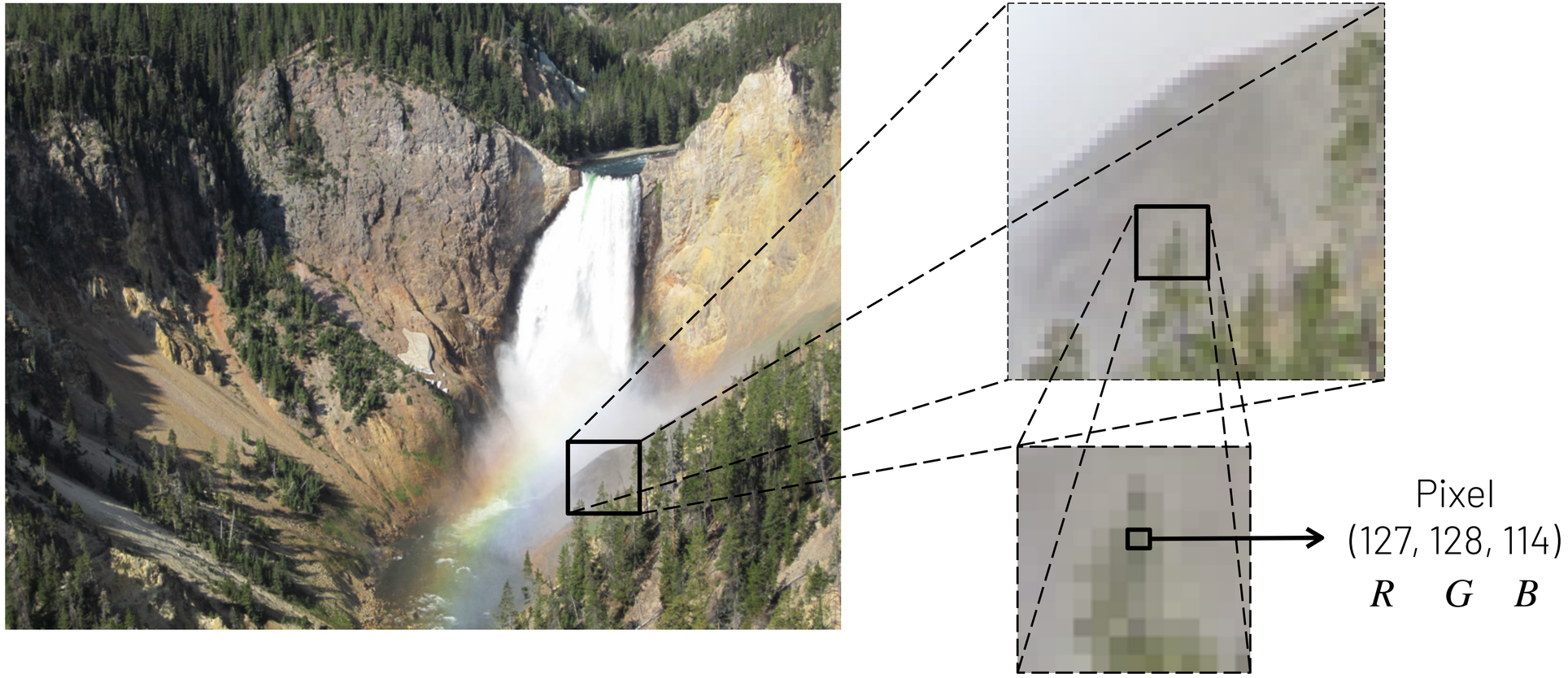
Esse vídeo utiliza câmeras com taxas de quadros mais altas do que as de TVs para visualizar como cada tecnologia (CRT vs LCD/LED/OLED) atualiza as imagens na tela.



https://www.youtube.com/watch?v=3BJU2drrtCM&ab_channel=TheSlowMoGuys

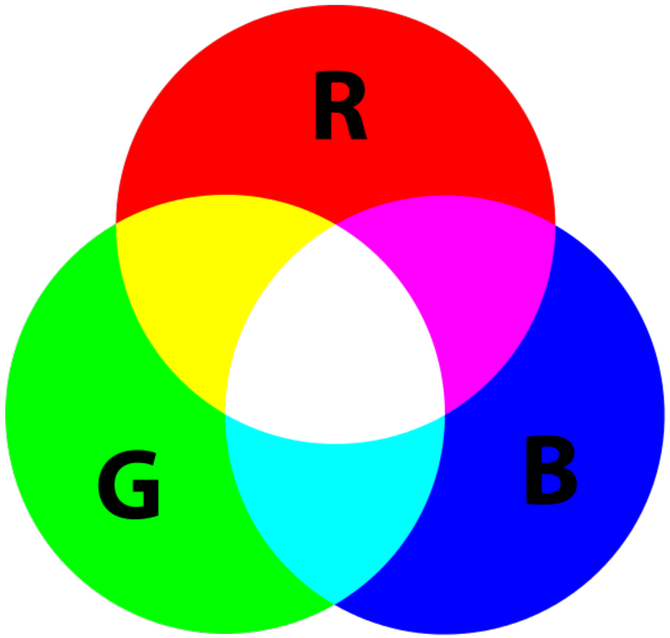
Imagens

Imagens são arranjos bidimensionais de pixels, onde cada pixel é representado por 3 valores (canais): vermelho (R), verde (G) e azul (B)



Cores

O sistema RGB é um padrão para definição de cores, onde uma cor é definida pela combinação de 4 canais: Vermelho (R), Verde (G), Azul (B) e Alpha (transparência)



- Cores representadas em 32 bits
- 8 bits por canal - intensidade varia entre 0 e 255
- 2^{32} or ~4 bilhões de combinações

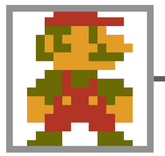
Transparência

O **Canal Alpha** é um quarto canal de cor usado para adicionar transparência às imagens. Esse canal é **uma adição de software,** monitores não desenham um canal de transparência!

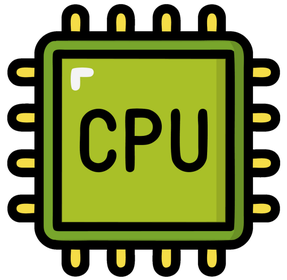
Desenhando imagens na tela

Para desenhar um imagem na tela do monitor, temos que preencher uma matriz de pixels da GPU chamada **Frame Buffer.**

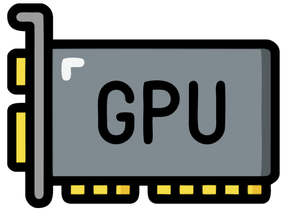
Carregar arquivo com imagem na memória RAM

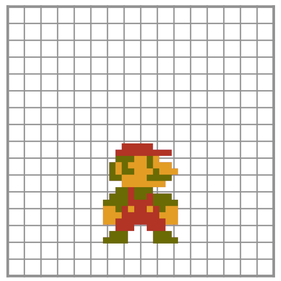


mario.png

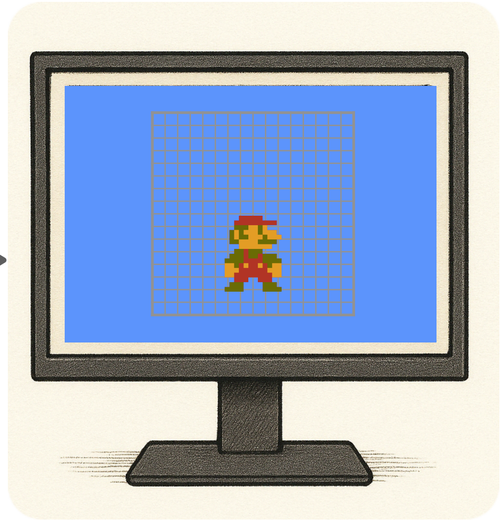


O frame buffer é representado pelo `SDL_Renreder` — atualizado na função `GenerateOutput()`





Frame Buffer (`SDL_Renderer`)



Monitor

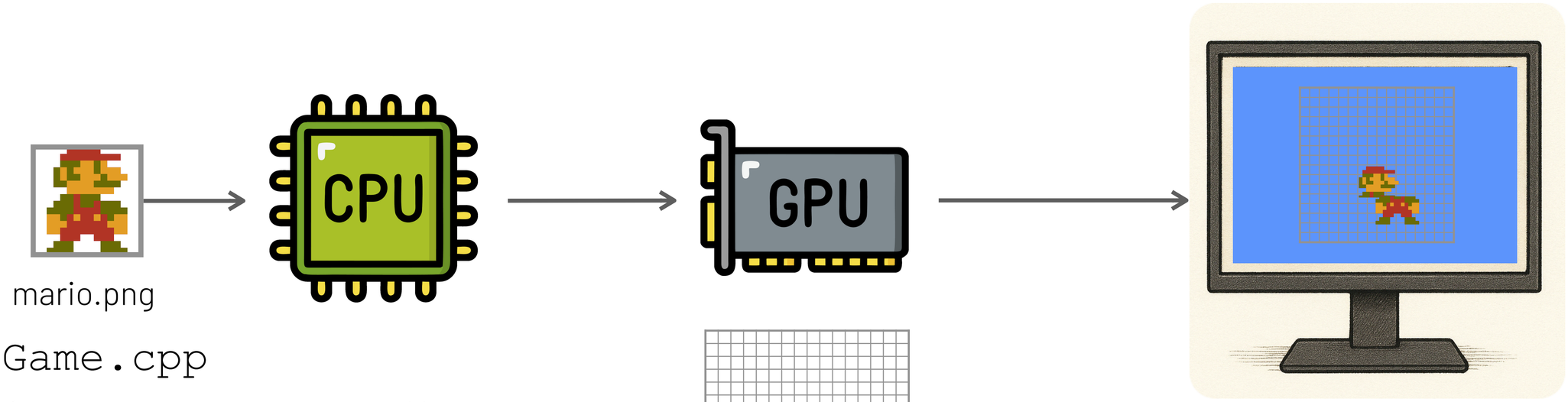
`Game.cpp`

1. `ProcessInput()`
2. `Update()`
3. **`GenerateOuput()`**

E se o jogo atualizar o buffer enquanto o monitor estiver atualizando a tela?

Screen Tearing (“Rasgo na Tela”)

Se o jogo atualizar o frame buffer enquanto o monitor estiver atualizando a tela, você pode ver uma imagem divida horizontalmente (ex. se o jogador moveu para a direita)



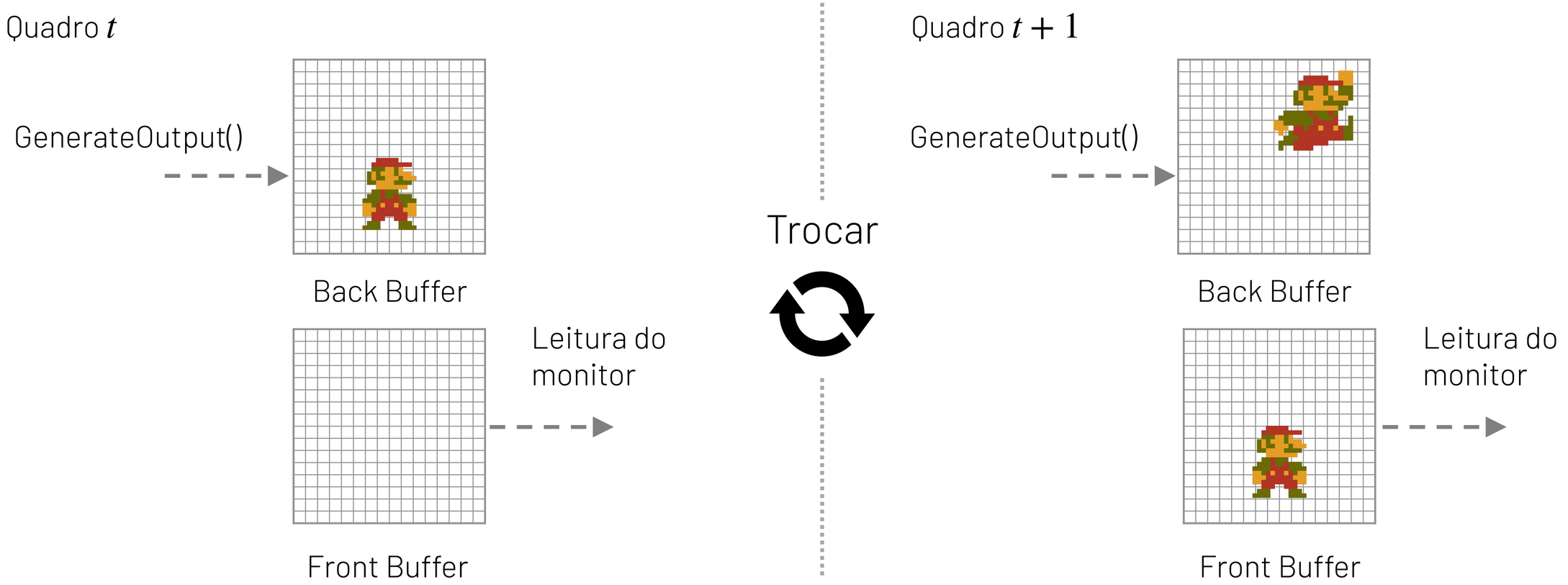

```
Game.cpp
1. ProcessInput()
2. Update()
3. GenerateOuput()
```

Frame Buffer (`SDL_Renderer`)

E se o jogo atualizar o buffer enquanto o monitor estiver lendo?

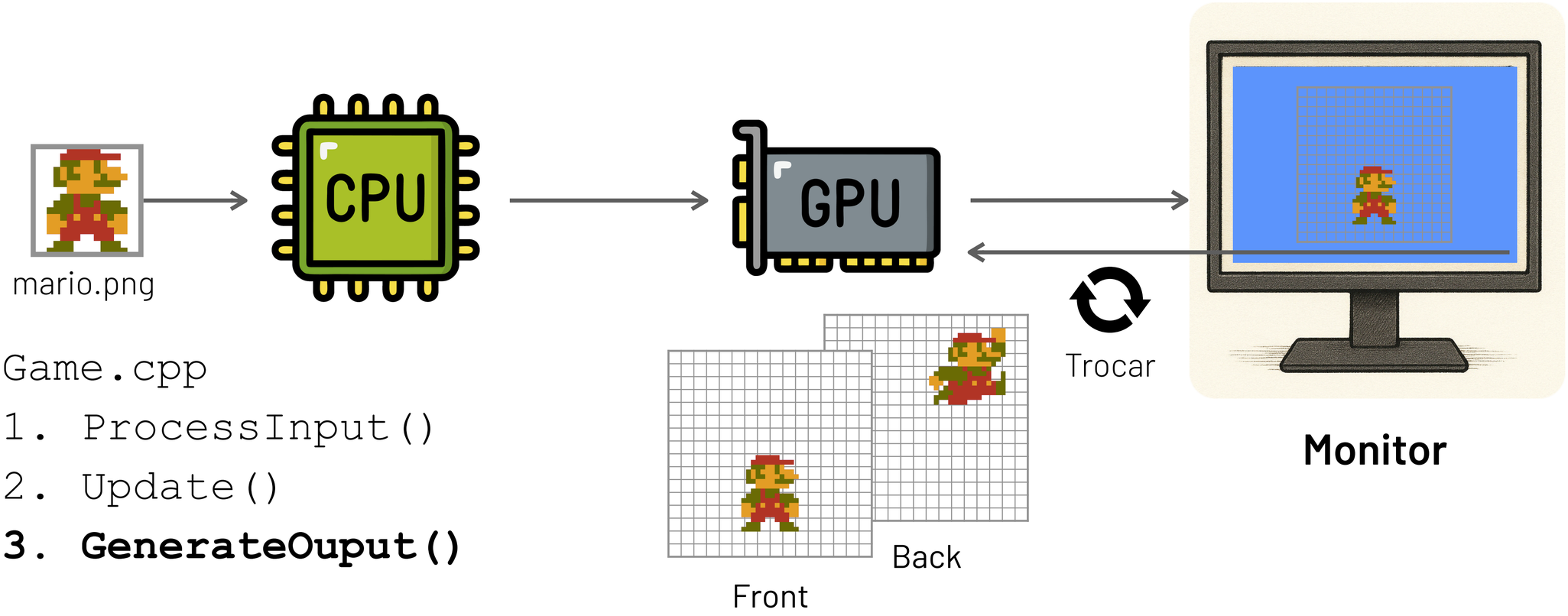
Double Buffer

Com **Double Buffering**, os gráficos são (1) desenhados em um back buffer, que (2) é trocado com o front buffer quando o quadro inteiro foi desenhado



VSync

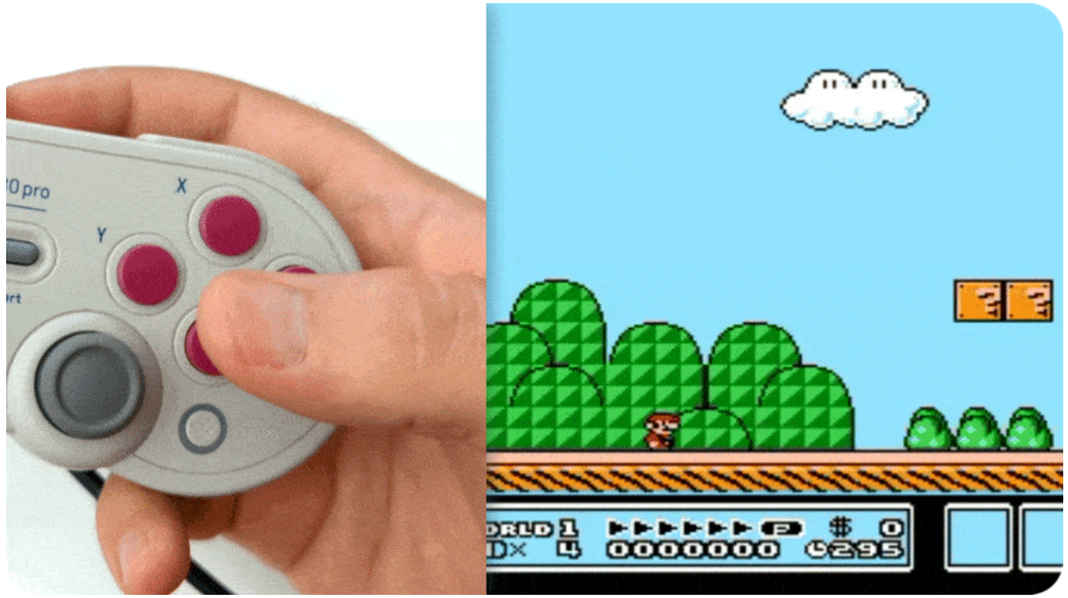
VSync é uma técnica que sincroniza o a taxa de atualização do monitor com o frame rate do jogo, trocando o front e o back buffers apenas quando o monitor terminar de desenhar.




```
Game.cpp
1. ProcessInput()
2. Update()
3. GenerateOuput()
```

Input Lag

Double buffer e VSync geram um atraso no processamento das entradas (***input lag***), pois temos que esperar a trocar de buffers e a sincronização com o monitor.

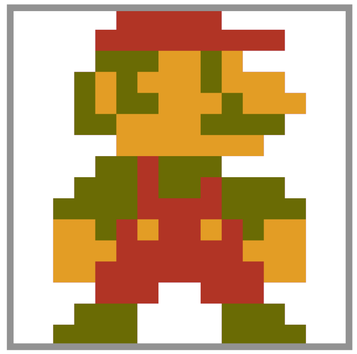


1. Limpar o back buffer
2. Desenhar a cena no back buffer
3. Esperar o fim da atualização do monitor
4. Trocar o front e o back buffers

Mesmo com esse atraso adicional, a maioria dos jogos ainda utilizam double buffer & VSync

Sprites

Um **sprite** é uma imagem utilizada para representar um estado (ou pose) de um objeto do jogo visualmente em 2D. Por exemplo: sprite do mario na pose *idle*



Um sprite é definido por:

```
class Sprite {
      SDL_Texture *texture;
      Vector2 position;
      int drawOrder;
      void Draw();
}
```

- Textura: imagem do sprite
- Posição: local de desenho na tela
- Ordem de desenho: define qual sprite é desenhado primeiro

Carregando Sprites em SDL

Para carregar imagens em SDL, precisamos usar uma biblioteca adicional `SDL_Image.h`

```
std::string texturePath = “mario.png”;

SDL_Surface* surf = IMG_Load(texturePath.c_str());
if (!surf)
{
    SDL_Log("Failed to load texture file %s", texturePath.c_str());
    return nullptr;
}

// Create texture from surface
SDL_Texture* texture = SDL_CreateTextureFromSurface(mRenderer, surf);
SDL_FreeSurface(surf);
if (!texture)
{
    SDL_Log("Failed to convert surface to texture for %s", texturePath.c_str());
    return nullptr;
}

return texture;
```

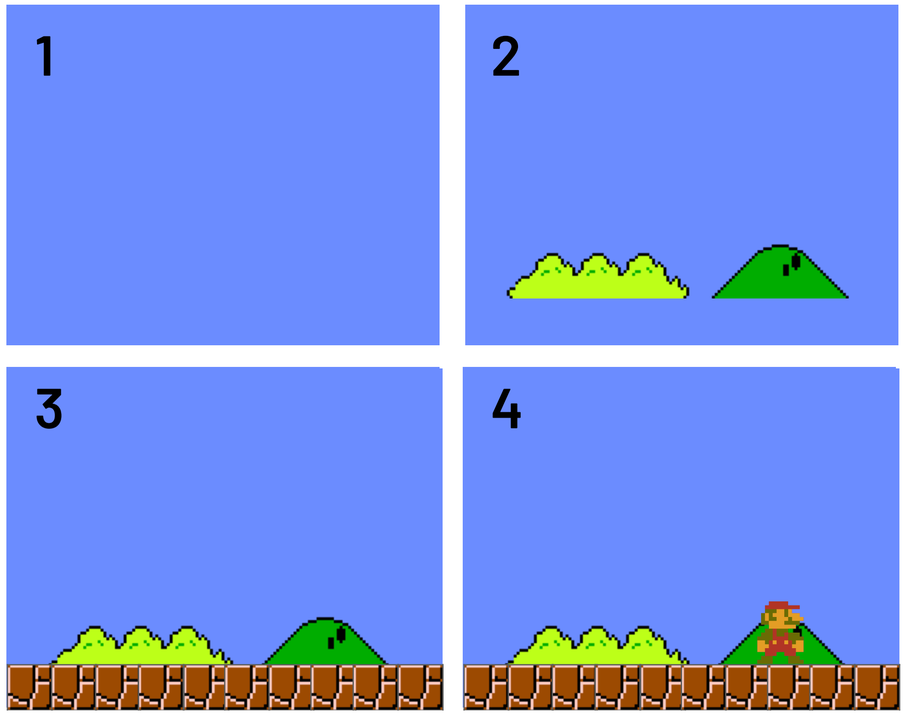
Desenhando sprites em SDL

Para desenhar imagens em SDL, precisamos usar uma biblioteca adicional `SDL_Image.h`

```
Vector2 position = Vector2::Zero;
Vector2 size = Vector2(32.0f, 32.0f);

SDL_Rect renderQuad = {static_cast<int>(position.x),
                       static_cast<int>(position.y),
                       static_cast<int>(size.x),
                       static_cast<int>(size.y)};

SDL_RendererFlip flip = SDL_FLIP_NONE;
SDL_RenderCopyEx(mRenderer, mTexture, nullptr, &renderQuad, 0.0, nullptr, flip);
```

Desenhando Sprites

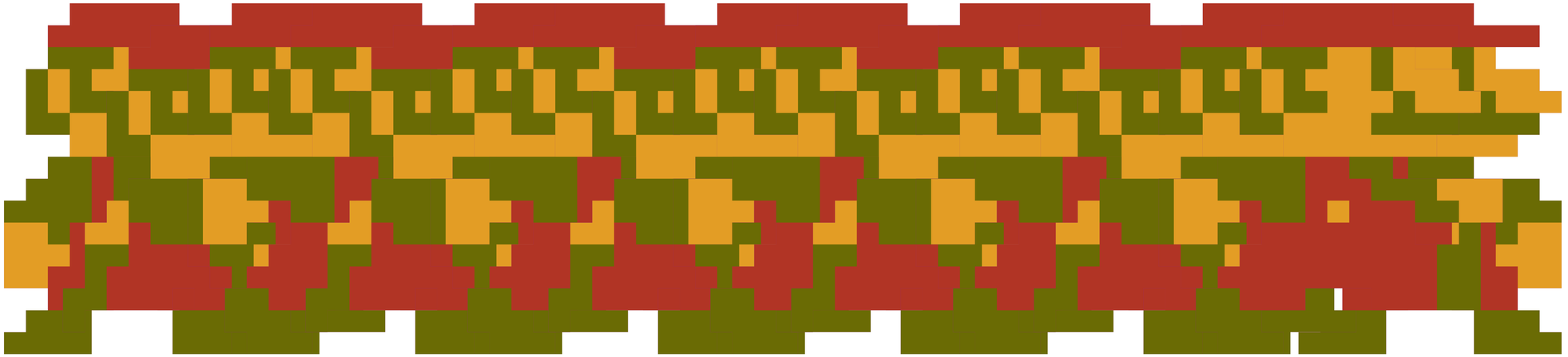
Sprites são normalmente desenhados seguindo o **algoritmo do pintor**: manter uma lista ordenada de sprites e desenhá-la de trás pra frente.




```
SortedList spriteList;

// When creating a new sprite...
Sprite newSprite = new Sprite(“img.png”);
newSprite->SetPosition(Vector2(10, 10));
newSprite->SetDrawOrder();

// Add to sorted list based on draw order value
spriteList.Add(newSprite->GetDrawOrder(), newSprite)

// When it's time to draw...
for (Sprite *s : spriteList)
   s->Draw()
```

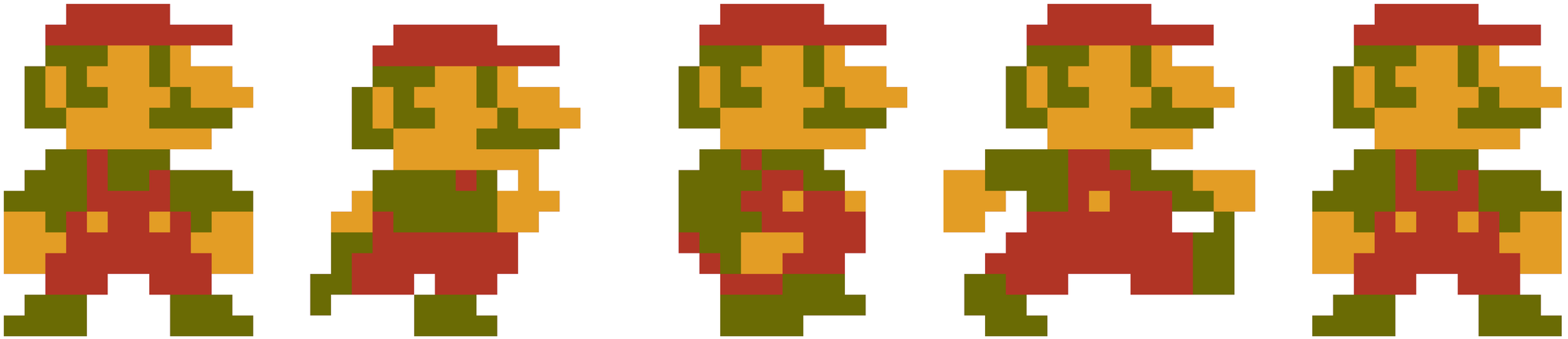
Animando Sprites

Para criar animações usando sprites, uma série de imagens estáticas reproduzidas em rápida sucessão para criar uma ilusão de movimento.



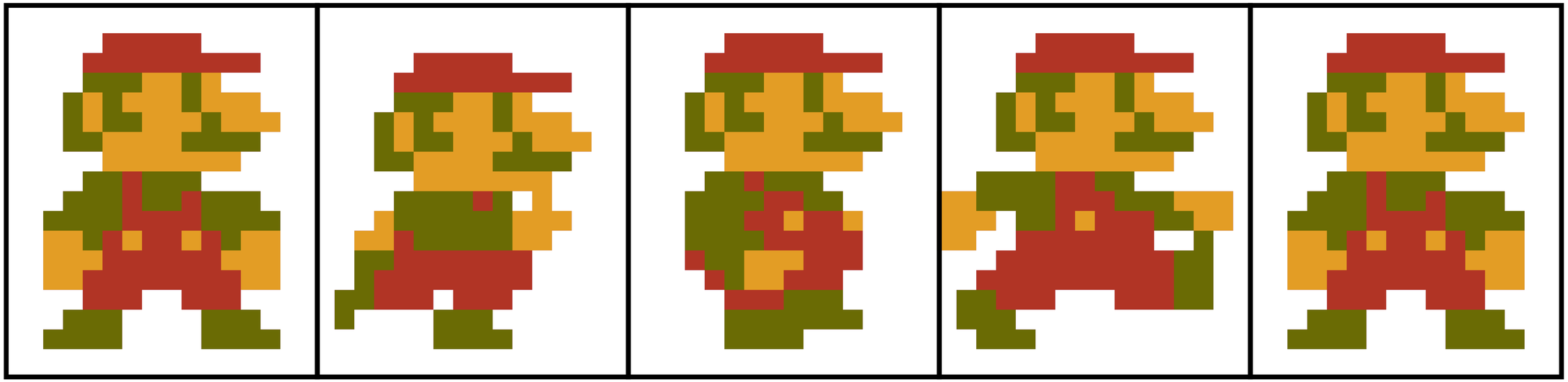
Animando Sprites

Para criar animações usando sprites, uma série de imagens estáticas reproduzidas em rápida sucessão para criar uma ilusão de movimento.



Armazenando Sprites

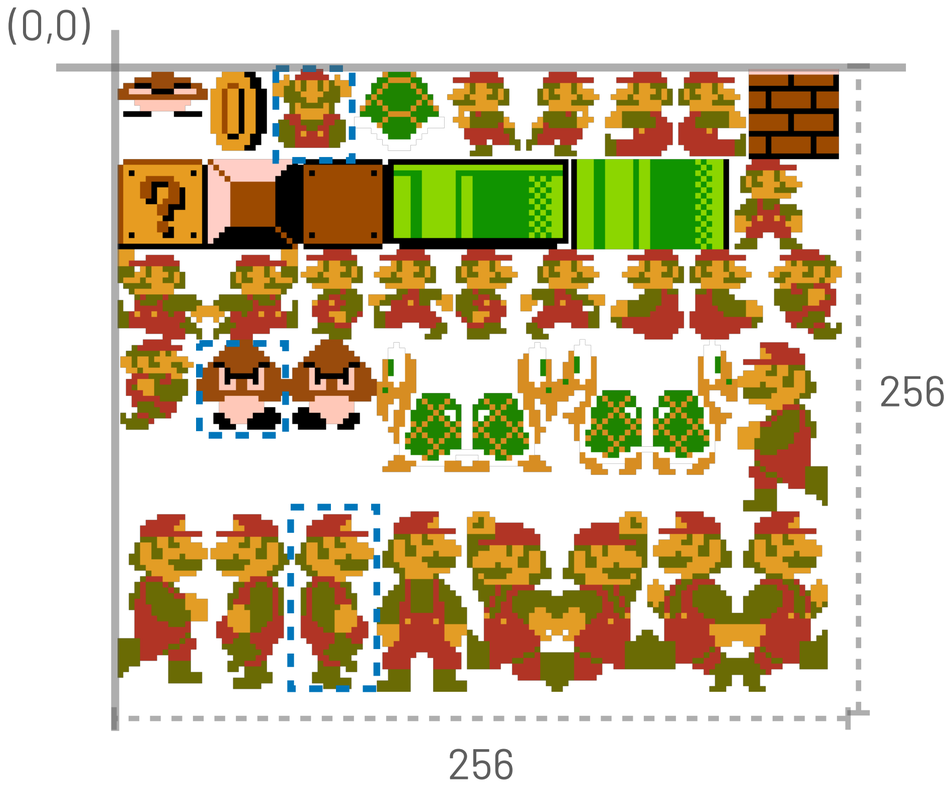
Armazenar sprites em arquivos separados pode acabar desperdiçando muita memória e processamento (considerando sprites com imagens de tamanhos iguais).



idle.png run1.png run2.png run3.png

Sprite Sheets

Para otimizar o espaço de armazenamento, geralmente agrupamos os sprites do jogo em um único imagem maior chamada de Sprite Sheet.



- A imagem do sprite sheet deve ser acompanhada de um arquivo auxiliar (ex. json) que lista a posição e tamanho de cada sprite:

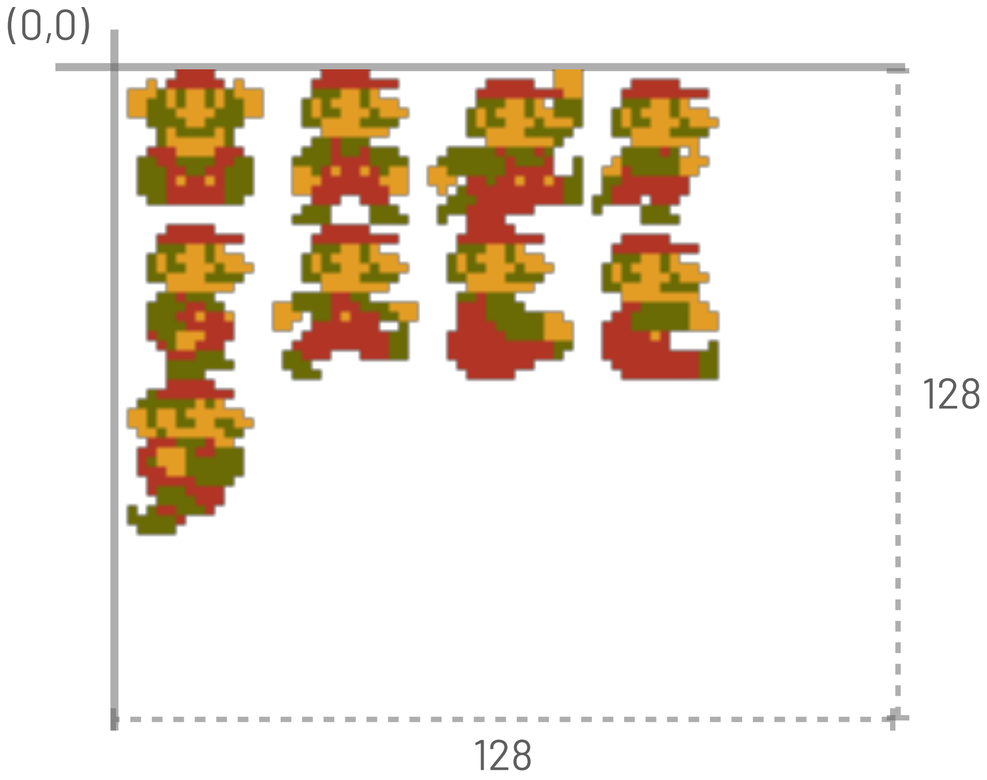

{x: 64, y: 0, w: 32, h: 32}

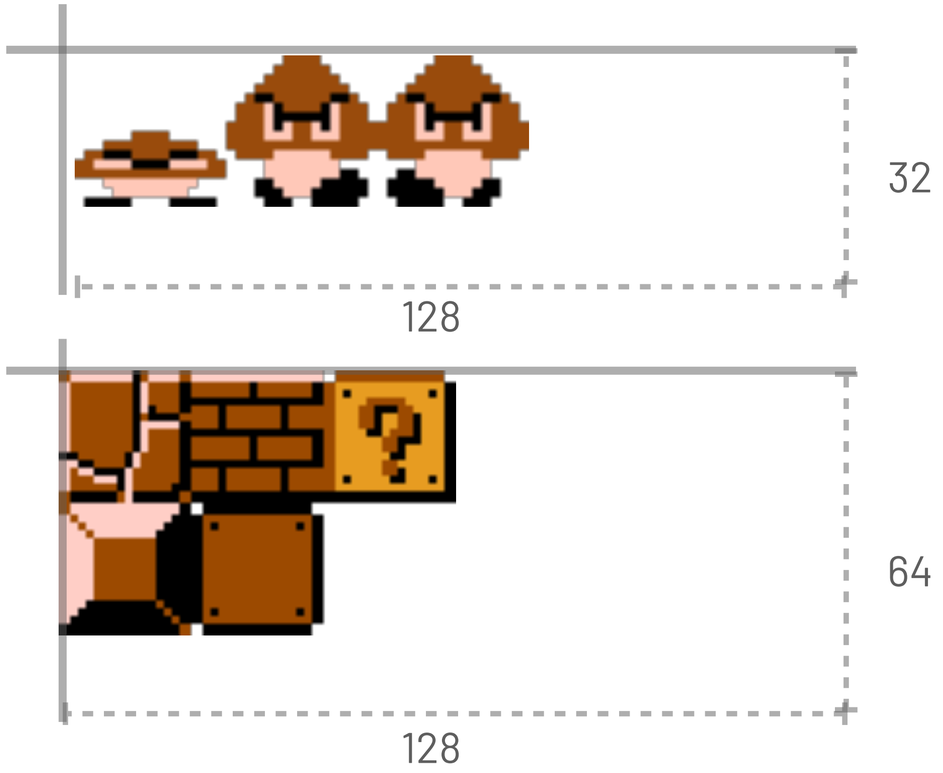

{x: 32, y: 128, w: 32, h: 32}

Historicamente, sprite sheets são criados em imagens com tamanho em potência de 2, por restrições das GPUs

Sprite Sheets

Opcionalmente, podemos criar um sprite sheet para cada personagem, facilitando a indexação de sprites e execução de animações.



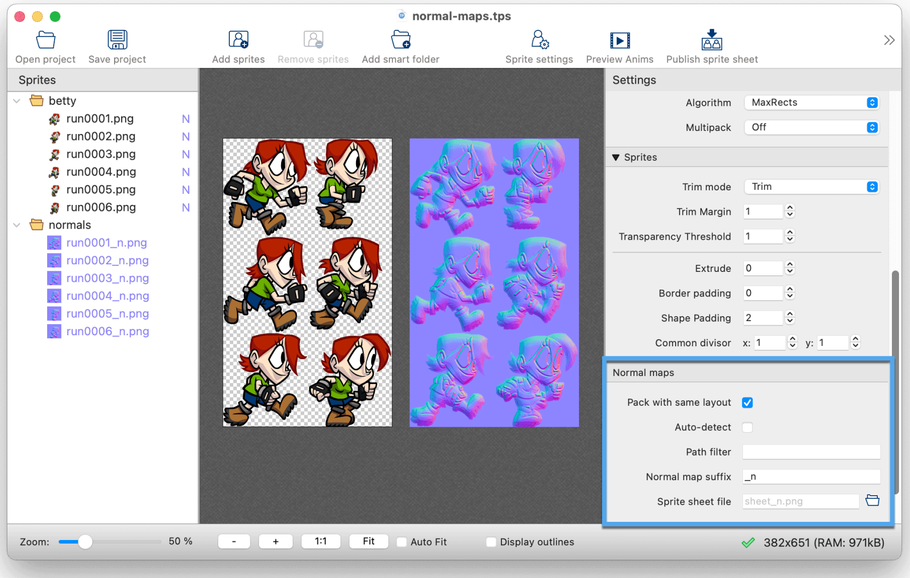


Dividir os sprites em múltiplos sprite sheets pode ser necessário dependendo da quantidade de sprites

Criando Sprite Sheets

Existem vários editores que auxiliam a criação de Sprite Sheets de maneira automática semi-automática:

- Texture Packer
  https://www.codeandweb.com/texturepacker
- Free Texture Packer
  https://free-tex-packer.com/
- Piskel
  https://www.piskelapp.com/p/create/sprite
- Aseprite
  https://www.aseprite.org/



Representando Animações

Manter uma lista de imagens com todos os quadros de um personagem:

Manter uma lista com os indices dos quadros de cada animação do personagem:

Idle	[0]
Jump	[1]
Run	[2, 3, 4]
Stomp	[5, 6]
Turn	[7]
Dead	[8]

Tocando Animações

Não podemos assumir que a taxa de quadros da animação seja mais lenta que a taxa de quadros do jogo

- FPS do Jogo: 30
- FPS de uma animação com 24 quadros: 48

Isso significa que muitas vezes precisaremos pular vários quadros na animação.

Tocando Animações

Precisamos de dois floats para tocar uma animação:

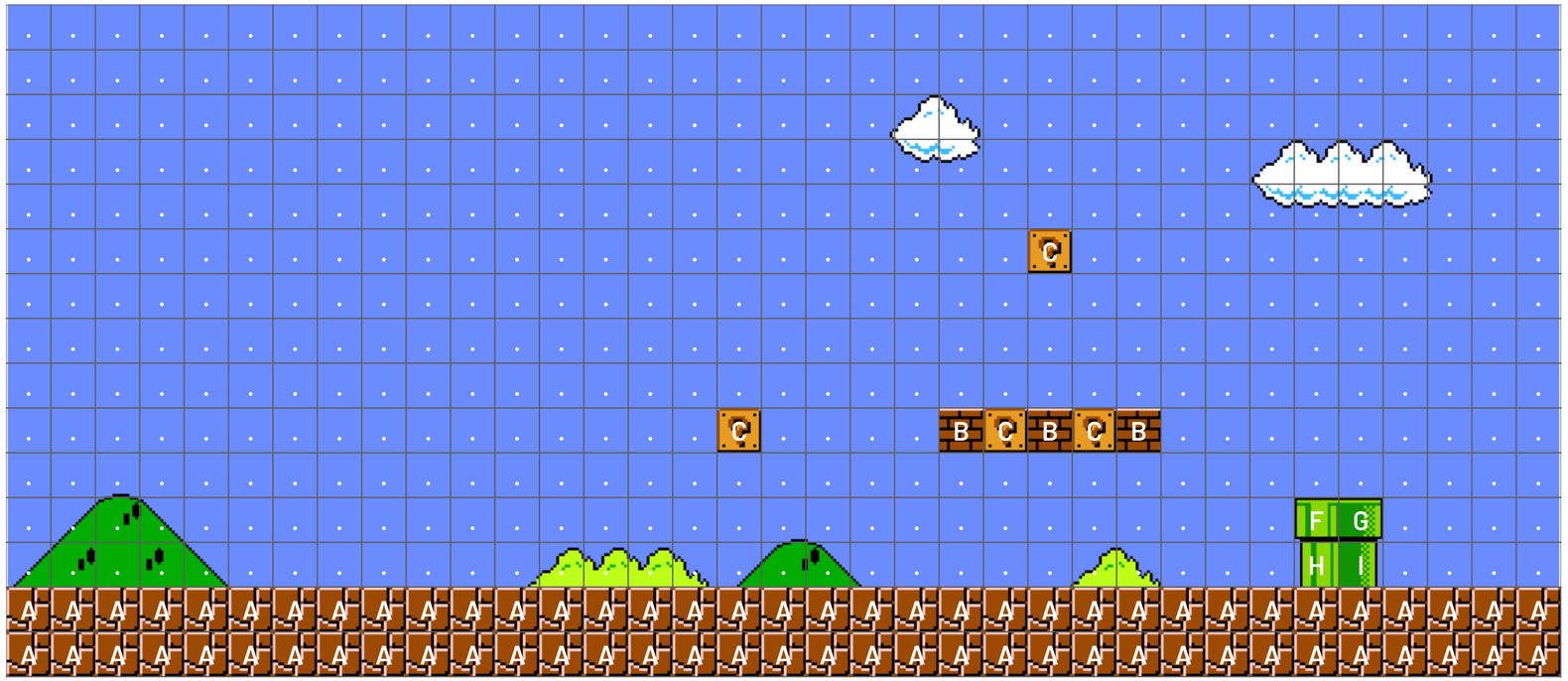
- `AnimTimer`, para armazenar o tempo corrente da animação
- `AnimFPS`, para amazenar a taxa de atualização da animação

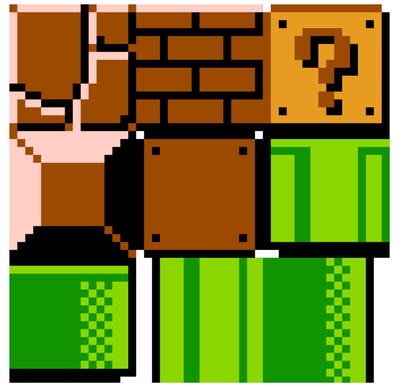
Transformar (cast) **`AnimTimer`** para inteiro para acessar o índice da animação

```
frameTime += deltaTime

// verificar se está na hora de alterar o sprite
if frameTime > (1 / animFPS) {
   animTimer = frameTime * animFPS // frameTime / (1 / animFPS) -> frameTime * animFPS
   if animTimer >= animData.frameInfo[animNum].numFrames:
      animTimer = (int)animTimer % animData.frameInfo[animNum].numFrames
}

int imageNum = animData.frameInfo[animNum].startFrame + frameNum
```

Tilemaps

Tilemaps são uma forma de organizar o mundo do jogo em uma grade de células de tamanhos iguais, cada um com número identificador, visando maximizar a repetição de sprites.



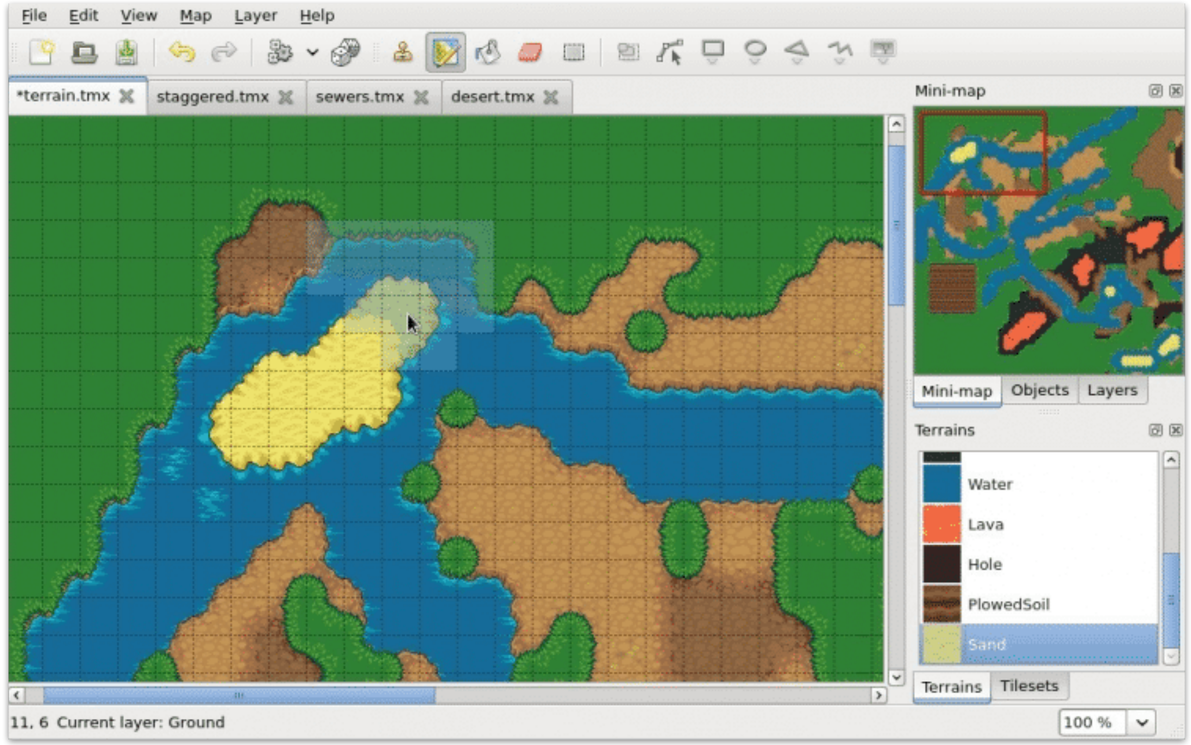


A estrutura de grade (grid) dos tilemaps facilita a editação de níveis, pois a posição dos tiles é limitado a coordenadas discretas.

Criando Tilemaps

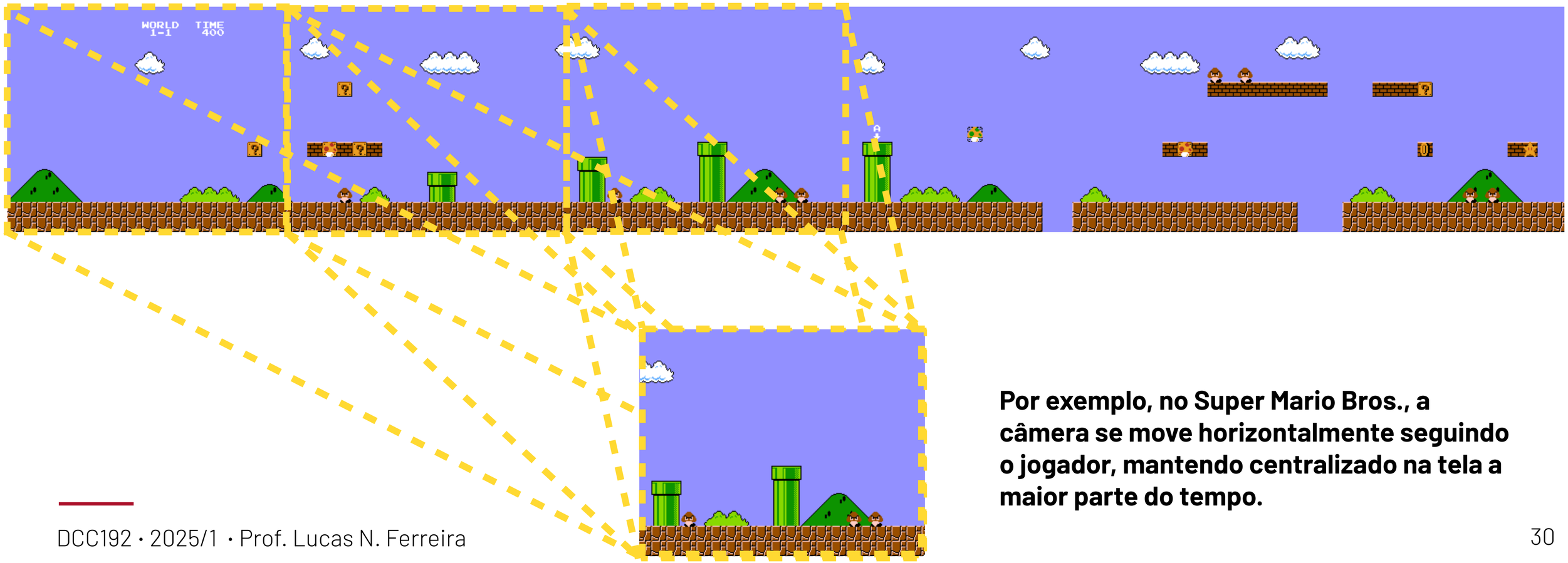
Existem vários editores que auxiliam a criação de **Tilemaps**. Além de faciliar a criação dos mapas em si, eles geralmente possibilitam a definição de colisores estáticos, triggers, entre outros:

- Tiled
  https://www.mapeditor.org/
- Sprite Fusion
  https://www.spritefusion.com/
- PixLab 2D Tilemap Maker
  https://tilemap.pixlab.io/



Rolagem de Câmera

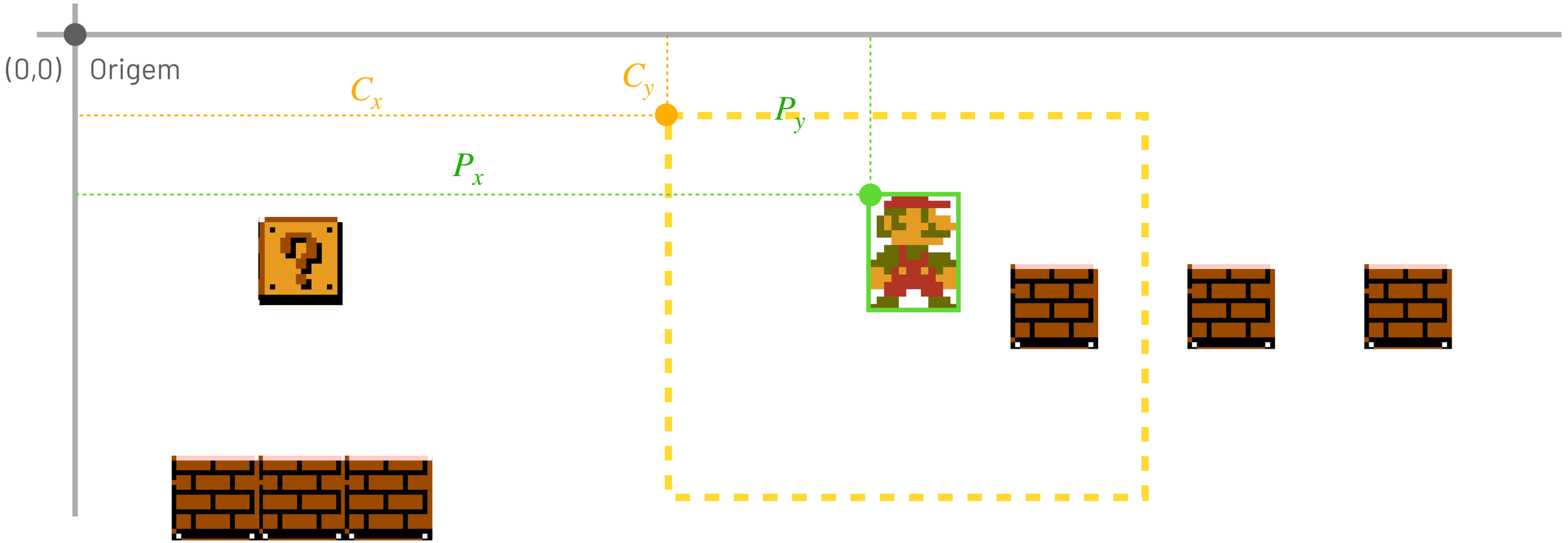
É muito comum que jogos 2D tenham mundos grandes que não cabem na tela. Nesse caso, precisamos implementar uma câmera, que se move para mostrar a região de interesse atual.



Por exemplo, no Super Mario Bros., a câmera se move horizontalmente seguindo o jogador, mantendo centralizado na tela a maior parte do tempo.

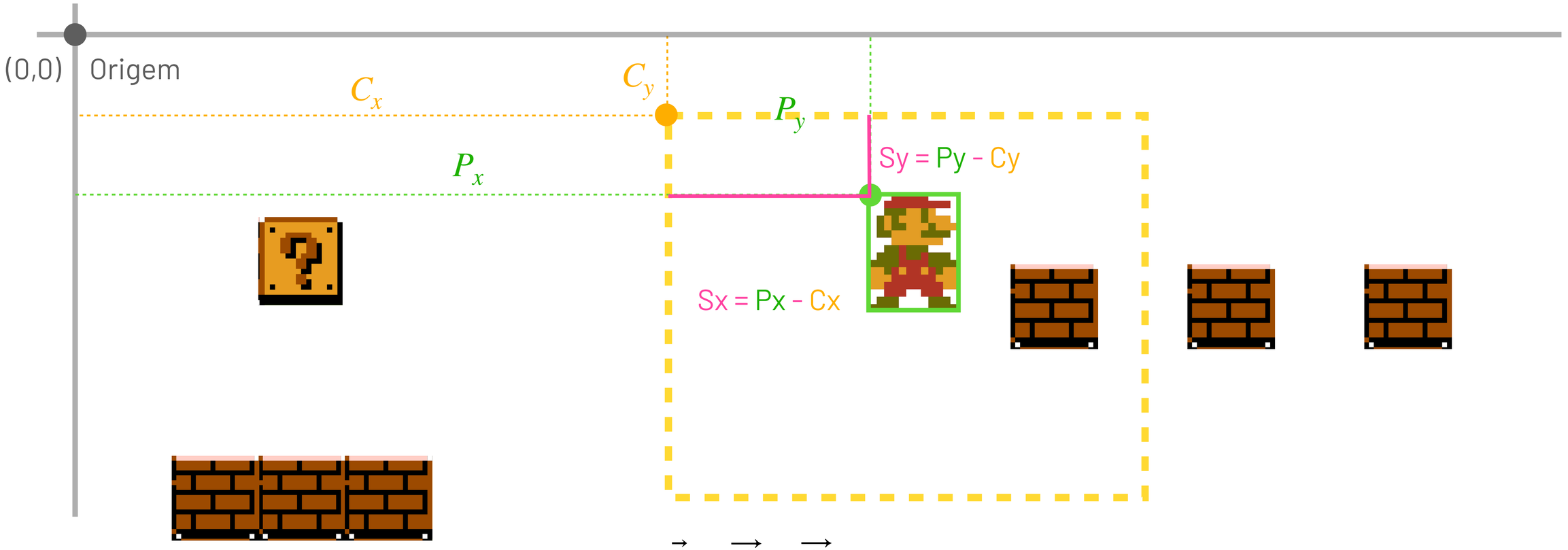
Rolagem de Câmera

Basta desenhar os objetos com relação à posição da **câmera** C, representada por uma posição relativa à origem do mundo.



Rolagem de Câmera

Basta desenhar os objetos com relação à posição da **câmera** C, representada por uma posição relativa à origem do mundo.



$$\vec{S} = \vec{P} - \vec{C}$$

```
screenPosition = worldPosition - cameraPosition
```

Efeito de Paralaxe

Objetos mais distantes se movem mais lentamente do que objetos mais próximos:

Para implementar esse efeito, basta multiplicar a posição da câmera $\vec{C}$ por um fator de paralaxe p:

$$\vec{S} = \vec{P} - p\vec{C}$$

Por exemplo:

- $p = 1.0$ (camada do jogador)
- $p = 0.5$ (camada do meio)
- $p = 0.25$ (camada do fundo)

Próxima aula

A11: Interface com o usuário

- Sistemas de Menus
- Gerenciamento de Cenas
- Heads-up Display