

Cenários 2D: composição, camadas, recorte e animação com sprites

por Rossana B Queiroz

Relembrando...

- *Game Loop*

- Cada ciclo gera um *frame* a ser desenhado na janela da aplicação
- Etapas distintas, de preferência *encapsuladas* em métodos/funções



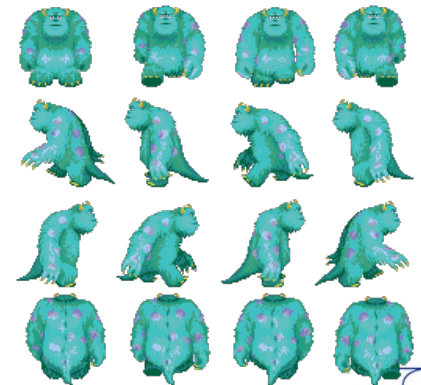
Relembrando...

- *Game Loop*

```
while(true) {  
    processInput(); // mouse, teclado, etc  
    update(); // estado + lógica  
    render(); // rotinas que fazem o desenho do frame  
    sync(); // controle tempo!  
}
```

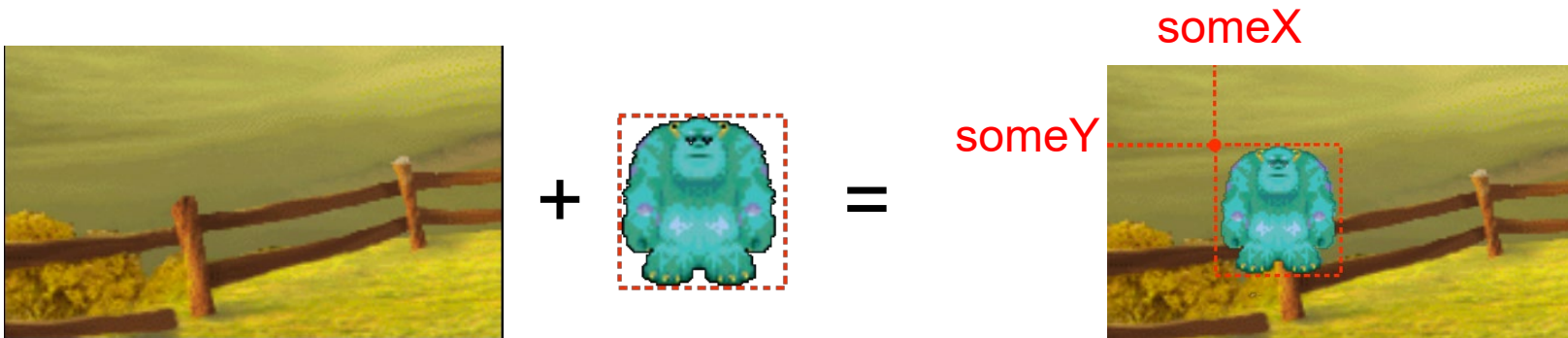
Introdução

- Desenho de cenários em camadas ou animações com *sprites* envolve composição de imagens, utilizando canal alfa.
- Cada *sprite* da *spritesheet* pode ser “recortada” em imagens separadas (1 animação por linha, 1 quadro por coluna) e armazenada em um *array* de imagens
- Desenho de imagens em *loop* exigem um temporizador



Composição de imagens

- Adicionar uma nova imagem por cima dos pixels de outra imagem (na verdade substituir pixels conforme alfa).
- Exemplo:



Composição de imagens

- Implementação

- Cada imagem da cena será um quadrilátero que mantém o *aspect ratio* da imagem, que será aplicada a ele como uma **textura**.
- As imagens deverão ser desenhadas chamando o *shader*
- Deve-se habilitar o teste de profundidade

```
glEnable(GL_DEPTH_TEST);  
glDepthFunc(GL_ALWAYS);
```

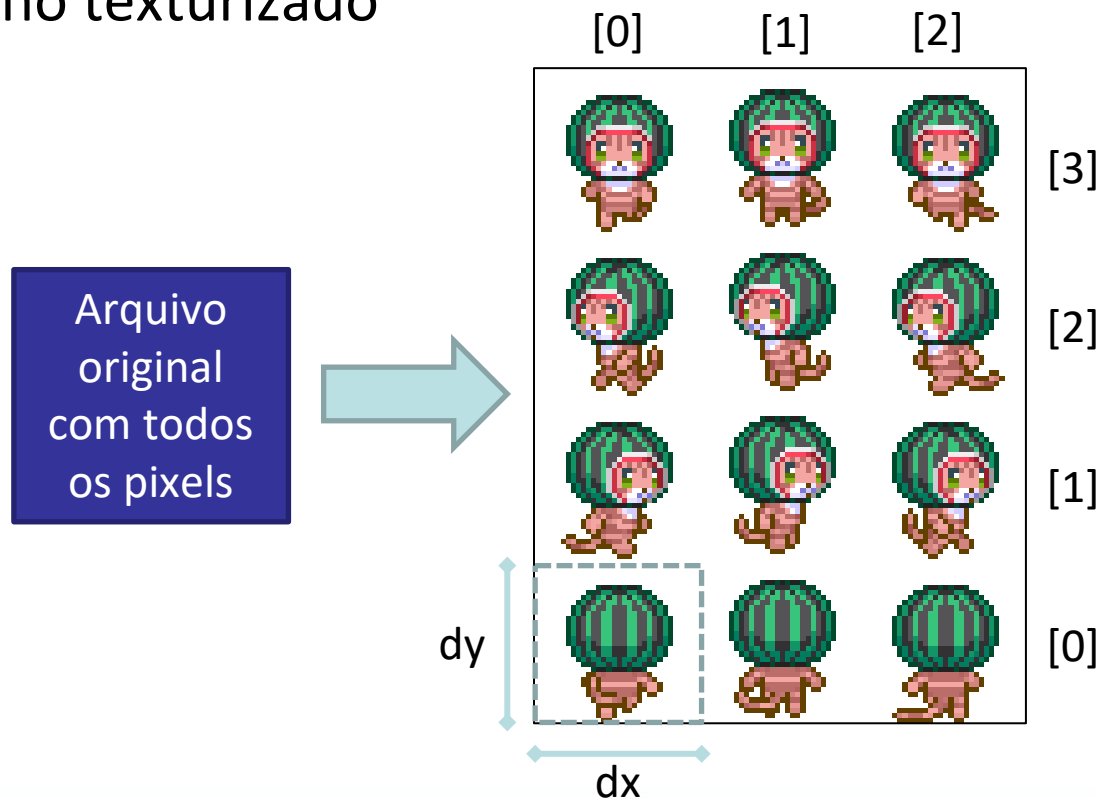
```
glClear(GL_COLOR_BUFFER_BIT |  
GL_DEPTH_BUFFER_BIT);
```

- Para transparências, deve-se habilitar

```
glEnable(GL_BLEND);  
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

“Recorte” de Imagens - *Spritesheet*

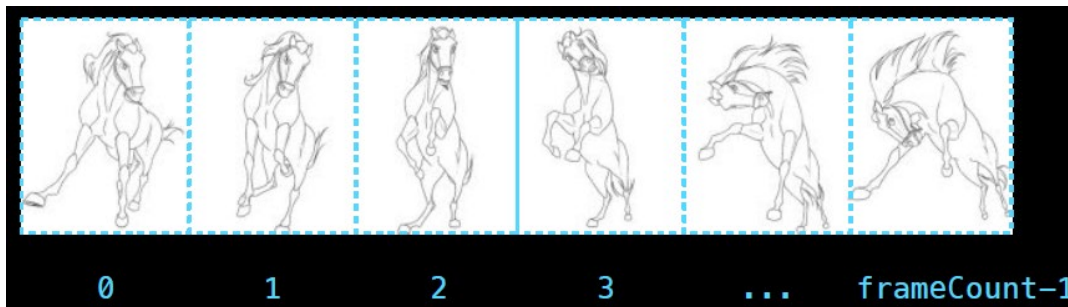
- De acordo com o índice da animação e do frame a ser desenhado, fazer o ajuste das coordenadas de textura no polígono texturizado



Animação por Sprites - Implementação

- Para desenhar, basicamente, um *frame* da seqüência é mostrado por vez.
- Há um tempo de espera (sincronização) que deve ser obedecido para desenhar o próximo quadro.
 - Este tempo de espera vai regular velocidade da ação do personagem com a capacidade de computação da plataforma operacional da aplicação.
- Quando o último quadro é desenhado, deve-se iniciar a seqüência no primeiro quadro:

```
currentFrame = (currentFrame+1) % frameCount
```



Offsets nas coordenadas de textura

iAnimation

Coordenadas de textura

1.0, 1.0

$$dy = \frac{1.0}{\text{numAnimations}} - \frac{1}{2} - 0.5$$

$dx = 1.0/\text{numFrames} = 1/8 = 0.125$

[1]

$$78\text{px}/2 = 39\text{px}$$

[0]

0.0,0.0

iFrame

 ~~$240\text{px} / 8 = 30\text{px}$~~
$$\text{offsetx} = \text{iFrame} * \text{dx} \quad 2 * 0.125$$
$$\text{offset}_y = i\text{Animation} * dy$$
$$0 \neq 0.5 = 0$$

Offsets nas coordenadas de textura

Na inicialização do sprite:

```
dx = 1.0 / (float)nFrames;  
dy = 1.0 / (float)nAnimations;  
  
float vertices[] = {  
    // positions          // colors          // texture coords  
    0.5f,  0.5f, 0.0f,    1.0f, 0.0f, 0.0f,    dx, dy, // top right  
    0.5f, -0.5f, 0.0f,    0.0f, 1.0f, 0.0f,    dx, 0.0f, // bottom right  
    -0.5f, -0.5f, 0.0f,    0.0f, 0.0f, 1.0f,    0.0f, 0.0f, // bottom left  
    -0.5f,  0.5f, 0.0f,    1.0f, 1.0f, 0.0f,    0.0f, dy // top left  
};
```

Offsets nas coordenadas de textura

Na atualização do sprite:

```
GLint offsetLoc = glGetUniformLocation(shader->Program, "offset");  
glUniform2f(offsetLoc, iFrame * dx, iAnimation * dy);  
  
iFrame = (iFrame + 1) % nFrames;
```

No shader...

```
#version 450 core
in vec2 TexCoord;

out vec4 color;

// pixels da textura
uniform sampler2D tex1;

//Texture coords offsets for animation
uniform vec2 offset;

void main()
{
    color = texture(tex1, TexCoord+offset);
}
```

Animação por Sprites - Implementação

- Usando a glfw, podemos pegar o tempo decorrido desde a inicialização usando a função `glfwGetTime`
- Um jeito simples de contabilizar o tempo:
double time = `glfwGetTime()`;

```
//GAME LOOP
while (!glfwWindowShouldClose(window))
{
    timer.start();

    //faz as chamadas de update e desenho...
```

Animação por Sprites - Implementação

- Classe Timer

```
#include <chrono>
#include <thread>
#include <ctime>

class Timer
{
public:
    Timer();

    void start() { begin = std::chrono::system_clock::now(); }
    void finish() { end = std::chrono::system_clock::now(); }
```

... continua ->

Animação por Sprites - Implementação

- Classe Timer

```
double getElapsedTimeMs()  
{  
    std::chrono::duration<double> elapsed_seconds = end - begin;  
    return elapsed_seconds.count() * 1000;  
}  
  
double getElapsedTime()  
{  
    std::chrono::duration<double> elapsed_seconds = end - begin;  
    return elapsed_seconds.count();  
}
```

... continua ->

Animação por Sprites - Implementação

- Classe Timer

```
double calcWaitingTime(int fps, double elapsedTime) {  
    double wt = 1000 / (double)fps - elapsedTime;  
    return wt;  
}  
  
protected:  
    // Using time point and system_clock  
    std::chrono::time_point<std::chrono::system_clock> begin,  
    end;  
};
```


Animação por Sprites - Implementação

- Se, simplesmente, desenharmos o próximo frame dentro do *game loop*, a animação não ficará com uma velocidade adequada. Não ficará “natural” em relação à ação do personagem.
- Precisamos sincronizar o processo de desenho para que ele tenha a velocidade de execução de acordo com a velocidade da animação. Por devemos esperar um tempo (**waitingTime**) até o próximo render!
- Este tempo de espera está relacionado a velocidade que queremos para processamento do jogo/animação (**FPS**), menos o tempo que já foi gasto com processamento (física, IA, desenho e etc.):

```
double calcWaitingTime(int fps, double elapsedTime) {  
    double wt = 1000 / (double)fps - elapsedTime;  
    return wt;  
}
```

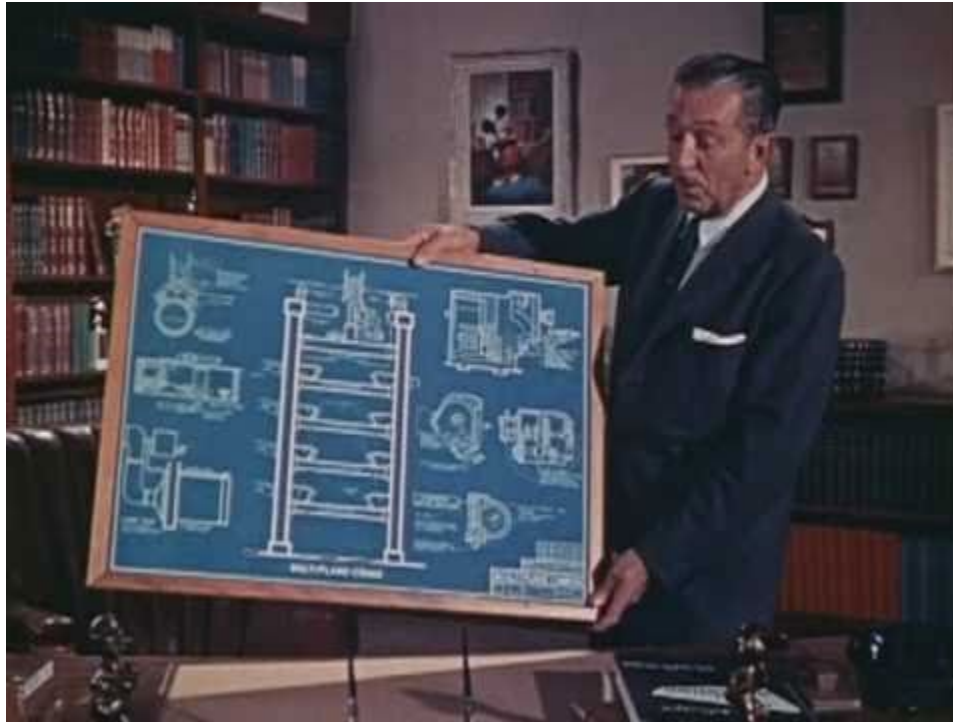
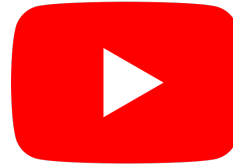
Animação por Sprites - Implementação

- Portanto, o próximo frame só será desenhado quando se passar o tempo desejado

```
//GAME LOOP
while (!glfwWindowShouldClose(window))
{
    timer.start();
    glfwPollEvents();
    //...Chama métodos de atualização e desenho dos objetos da cena....
    //Sincronizando o FPS
    timer.finish();
    double waitingTime = timer.calcWaitingTime(60, timer.getElapsedTimeMs());
    if (waitingTime)
    {
        std::this_thread::sleep_for(std::chrono::milliseconds((int)waitingTime));
    }
    glfwSwapBuffers(window);
}
```

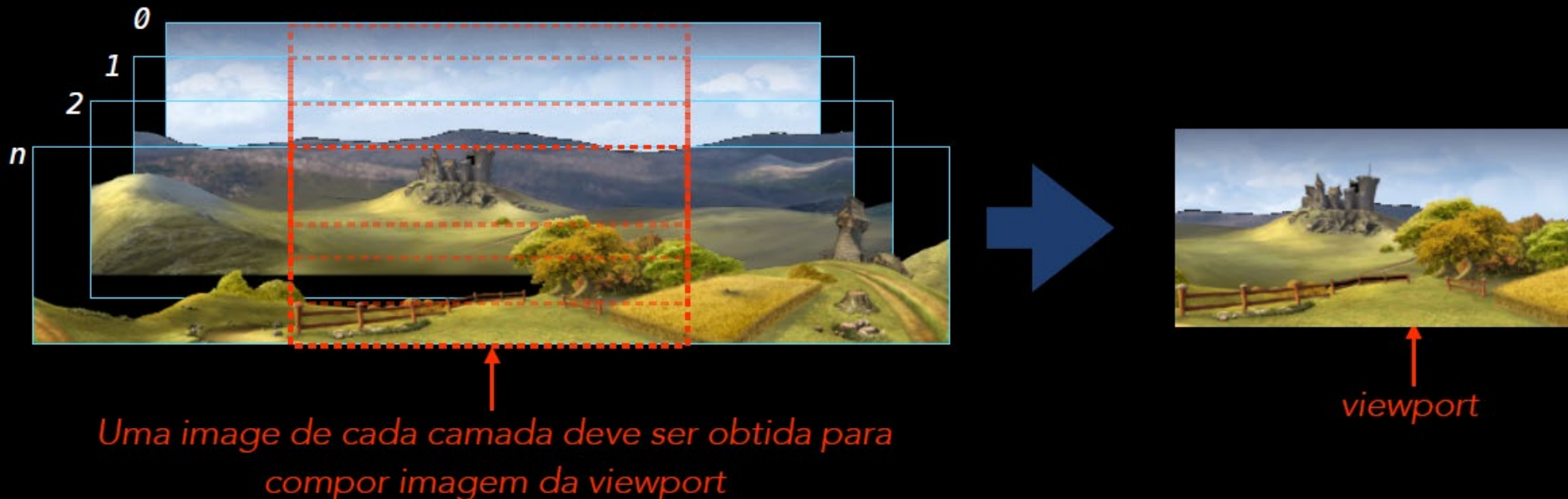
Camadas e Parallax

- História: Multiplane Camera



Camadas e Parallax

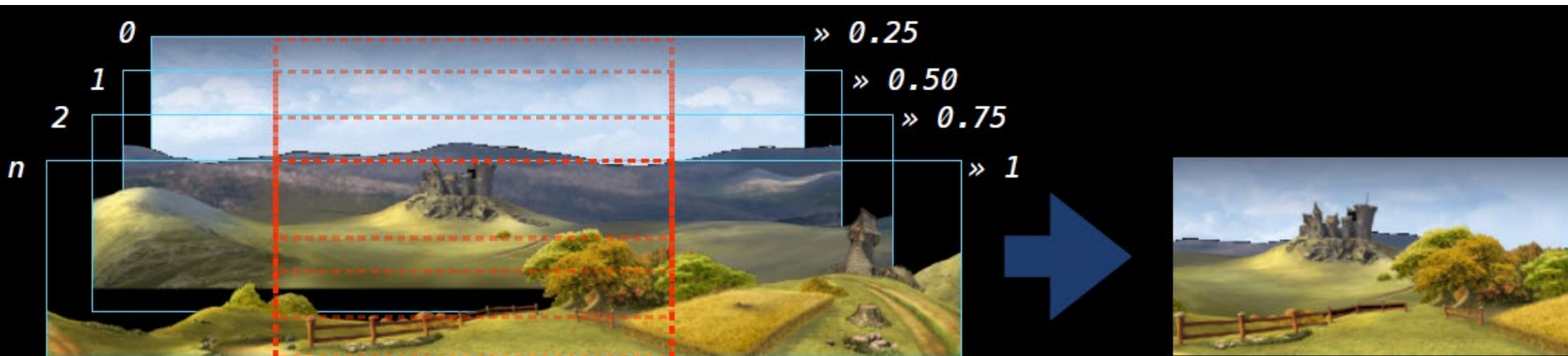
- Para desenhar cenário com múltiplas camadas e com efeito *parallax* devemos considerar a implementação da classe Layer
- Camadas devem ser compostas para gerar a cena final. Os elementos animados das camadas também devem ser renderizados adequadamente.



Camadas e Parallax

- Quando o jogador/usuário movimenta a viewport, ele espera visualizar outra parte da imagem camada, portanto, deve ocorrer scrolling das camadas.
- Seguindo o conceito de *Parallax* (conforme visto em aula), temos que associar uma taxa de scrolling com cada camada.
- Considere que quanto mais ao fundo a camada, menor é a sua taxa de movimentação. Calculando uma proporção a partir da camada de ação (primeiro plano ou movimentação 1:1):

```
layers[i]->setOffsets(offsetx,offsety);
```



Camadas e Parallax

- Sendo assim, para formar a imagem da composição das camadas **sem a animação**, devemos:

- Definir taxa de scrolling da camada:

```
for all layers [0..n]
    layers[i]->computeScrollRateX(mainLayer->getWidth());
    layers[i]->computeScrollRateY(mainLayer->getHeight());
```

- Quando ocorrer evento de movimentação:

```
for all layers [0..n]
    layers[i]->scroll(forward);
```

- Quando for necessário compor uma imagem para viewport:

```
for all layers [0..n]
    layers[i]->plot(viewport); // será alterado para animação
```

Camadas e Parallax

- Considerando que temos objetos animados na cena, cada objeto deve conter na sua posição as coordenadas x e y, e a profundidade (z ou *layer*) em que estão.
- Basicamente, duas abordagens podem ser seguidas para a composição da cena:
 1. A cada alteração de cena, apagar a cena e redesenhar todas as camadas com todos os elementos que pertencem a ela
 2. Manter uma imagem estática da composição das camadas e atualizar a região de cada personagem de acordo com a visibilidade da camada onde o personagem está

Camadas e Parallax

- Agora considerando animação de personagens:
 - Definir taxa de scrolling da camada:

```
for all layers [0..n]  
    layers[i]->computeScrollRateX(mainLayer->getWidth());  
    layers[i]->computeScrollRateY(mainLayer->getHeight());
```

- Quando ocorrer evento de movimentação:

```
for all layers [0..n]  
    layers[i]->scroll(forward);
```


Referências

- <https://learnopengl.com/>
- Anton's OpenGL 4 Tutorials

