DCC192

2025/2

UFMG

Desenvolvimento de Jogos Digitais

A15: Interface com o usuário

Prof. Lucas N. Ferreira

Logística

Avisos

- O **TP3: Super Mario Bros** será disponibilizado entre hoje e amanhã

Última Aula

- Game Design
- Level Design

Plano de aula

- Sistemas de Menus
  - Menu Principal
  - Menu de Pausa
  - Botões e Fontes
- Gerenciamento de Cenas
  - Máquinas de Estados Finitos
    - Swtich/Case
    - Padrão de Projeto State
  - Efeitos de Transição

Menu Principal

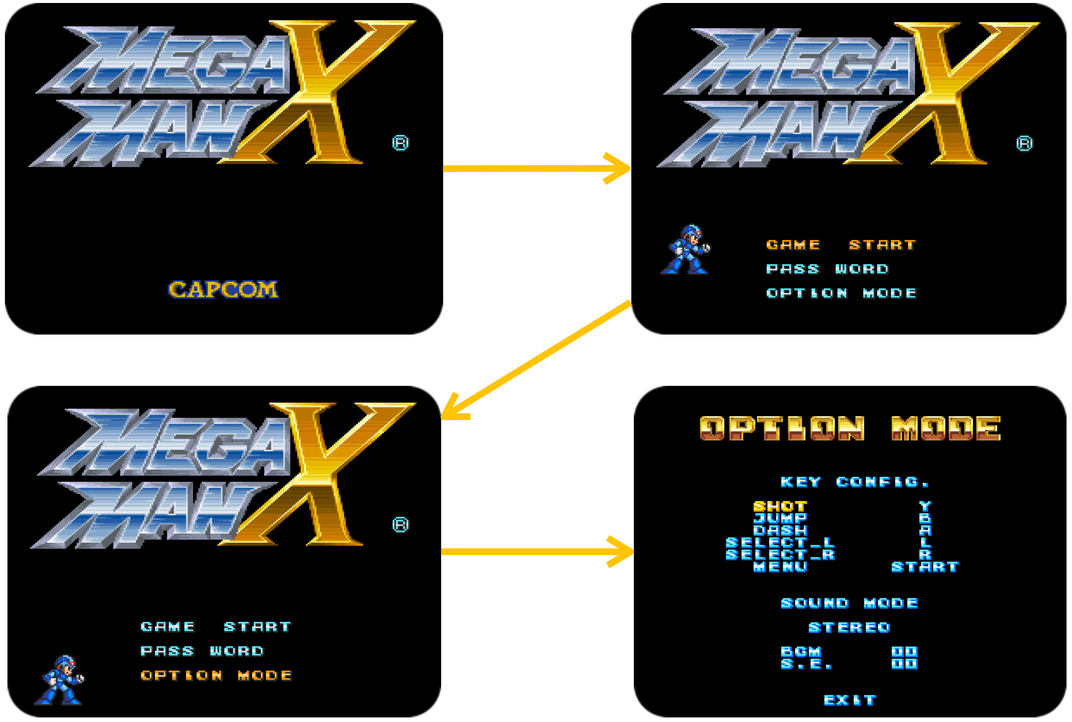
O **Menu Principal** é a primeira tela que um jogador vê ao iniciar um jogo. Ela atua comum uma interface introdutória, geralmente oferencendo alguas opções básicas, como:

- [Novo Jogo]
- [Carregar Progresso Anterior]
- [Opções]

  Vídeo, Áudio, Controle, Acessibilidade, ...
- ...

Menus

De uma maneira geral, um menu, é uma interface que pemite que o jogador navegue em um fluxo de telas. Por exemplo, no menu principal, geralmente oferece as seguintes opções:



- Press Start
  - Main Menu
    - Start Game
    - Load Game
    - Options
      - Video [back]
    - Options [back]
  - Main Menu

Sistemas de Menus

Sistemas de Menus são implementados usando uma estrutura de dados do tipo **pilha**:

Topo

Pilha

- **Topo** da pilha é a tela ativa: recebe eventos de entrada
- **Empilhar** para entrar em uma nova tela
- **Desempilhar** para voltar para a tela anterior
- **Implementação:**
  - Definir uma classe base para as telas `UIScreen`: estender essa classe para cada menu do jogo
  - A classe `Game` gerencia (push/pop) a pilha de telas
  - A pilha de telas é renderizada de baixo para cima, por isso implementamos como um vetor

UIScreen: Classe base de menus

Os menus do jogo, assim como outras partes da interface com o usuário, são implementados através de telas de interface, que podem conter botões, textos, imagens, etc...:

```
class UIScreen {
public:
  virtual void Update(float deltaTime);
  virtual void Draw(class Shader* shader);
  virtual void ProcessInput(const uint8_t* keys);

  void Open();
  void Close();

  enum UIState { EActive, Eclosing };

   void AddButton(const std::string& name,
   std::function<void()> onClick, const Vector2 &pos);
   void AddText(const std::string& text);
   void AddImage(const std::string& path);

private:
   UIState mState;
};
```

Métodos:

- `Update`: atualizar o estado da tela
- `Draw`: desenhar a tela
- `ProcessInput`: processar eventos de entrada
- `Open/Cloase:` abrir/fechar e janela
- `AddButton/AddText/AddImage`: adicionar botão/texto/imagem

Atributos:

- `mState`: estado da tela (Ativo/Fechando)
- `mButtons`: lista duplamente ligada de botões
- `mText/mImages`: listas de botões/imagens

UIText: Textos de Interface

Classe utilizada para desenhar textos em interfaces, como menus ou HUDs. Possui principalmente um ponteiro para um texto que pode ser desenhado com TTF.

```
class UIText {
public:

    void SetText(const std::string& name);
    void Draw(Renderer* renderer);

protected:
    std::string mText;
    class UIFont* mFont;
    Texture *mTextTexture;

    unsigned int mPointSize;
    unsigned int mWrapLength;
    Vector2 mPosition;

};
```

Métodos:

- `SetText`: Cria uma textura para o texto dado usando a fonte passada no construtor
- `Draw`: Desenha textural criada

Atributos:

- `mText`: string contendo o texto atual
- `mFont`: fonte utilizada para renderizar texturas
- `mTextTexture`: textura atual do texto
- `mPointSize`: tamanho da fonte
- `mWrapLength`: comprimento de quebra do texto

Fontes

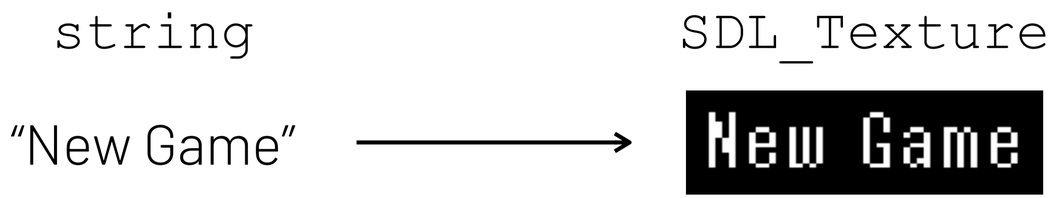
Fontes vetoriais são tipicamente implementadas desenhando contornos de caracteres individuais (*glifos*) com segmentos de retas e curvas de Bézier.

Existem diversos padrões de fontes vetoriais:
PostScript, TrueType Font (TTF), OpenType Font (OTF), ...

Fontes bitmap utilizam um sprite para cada caracter, ou seja, uma imagem para cada caracter:

Utilizando Fontes

Em ambos os casos (vetoriais ou bitmap), é necessário definir uma função `RenderText(s: string)` que mapeia uma string `s` em uma textura que irá ser desenhada na tela:




```
SDL_Texture* RenderText(SDL_Renderer* renderer,
            const std::string& text,
            const Vector3& color = Color::White,
             int pointSize = 30,
            unsigned wrapLength = 900);
```

`Font.h`

Carregando e desenhando fontes vetoriais em SDL

Para carregar fontes vetoriais em SDL, precisamos usar uma biblioteca adicional `SDL_ttf.h`

```
#include <SDL_ttf.h>

int size = 32;
TTF_Font* font = TTF_OpenFont(“font.ttf", size);
```

Com as fontas carregadas, podemos gerar uma textura a partir de uma string:

```
int wrapLength = 900;
SDL_Color color = {.r =  21, .g =  21, .b = 123, .a =  255};

SDL_Surface* surf = TTF_RenderUTF8_Blended_Wrapped(font, “New Game", sdlColor, wrapLength);

// Create texture from surface
SDL_Texture* texture = SDL_CreateTextureFromSurface(renderer, surf);
SDL_FreeSurface(surf);
```

UIButton: Botões de Interface

Classe utilizada para desenhar botões em interfaces. É similar ao UIText mas possui fundo e pode ser selecionado com o mouse ou teclado:

```
class UIButton
{
public:
   void SetText(const std::string& text);
   void SetHighlighted(bool sel);
   bool GetHighlighted() const ;

   bool ContainsPoint(const Vector2& pt) const;

   void OnClick();

private:
   std::function<void()> mOnClick;

   UIText mText;
   bool mHighlighted;
    Vector2 mPosition;
};
```

Métodos:

- `SetText`: Cria uma textura a partir de um texto
- `SetHighlighted`: Marca como selecionado
- `ContainsPoint`: Verifica se um ponto está dentro do botão
- `OnClick`: Função chamada quando o botão é clicado

Atributos:

- `mText`: string contendo o texto atual do botão
- `mPosition`: posição do botão na tela
- `mHighlighted`: estado do botão (selecionado ou não)

UIImage: Texturas de Interface

A classe que implemente imagens de interface é muito parecida com a classe de text (`UIText`)

```
class UIImage
{
public:
   bool Load(const std::string& fileName);
   void Unload();

   void SetActive(int index = 0);

   int GetWidth() const { return mWidth; }
   int GetHeight() const { return mHeight; }

private:
   SDL_Texture* mTexture;
   Vector2 mPosition;

};
```

Funções:

- `Load`: Cria uma textura a partir de um arquivo de imagem
- `Unload`: Descarrega textura aual

Atributos:

- `mText`: nome do arquivo de imagem carregado
- `mPosition`: posição da imagem na interface
- `mWidth`: largura da imagem na inteface
- `mHeight`: altura da imagem na interface

Heads-Up Display (HUD)

Camada de interface desenhada constantemente sobre a tela do jogo para mostrar informações importantes do estado do jogo, por exemplo:

Exemplos:

- Vida do personagem
- Energia do personagem
- Arma equipada
- Contadores (dinheiro, bombas, flechas, chaves, ...)
- Temporizador

Caixas de diálogo não necessáriamente são partes do HUD, pois são mais relacionadas à narrativa do que ao estado

Classes HUD

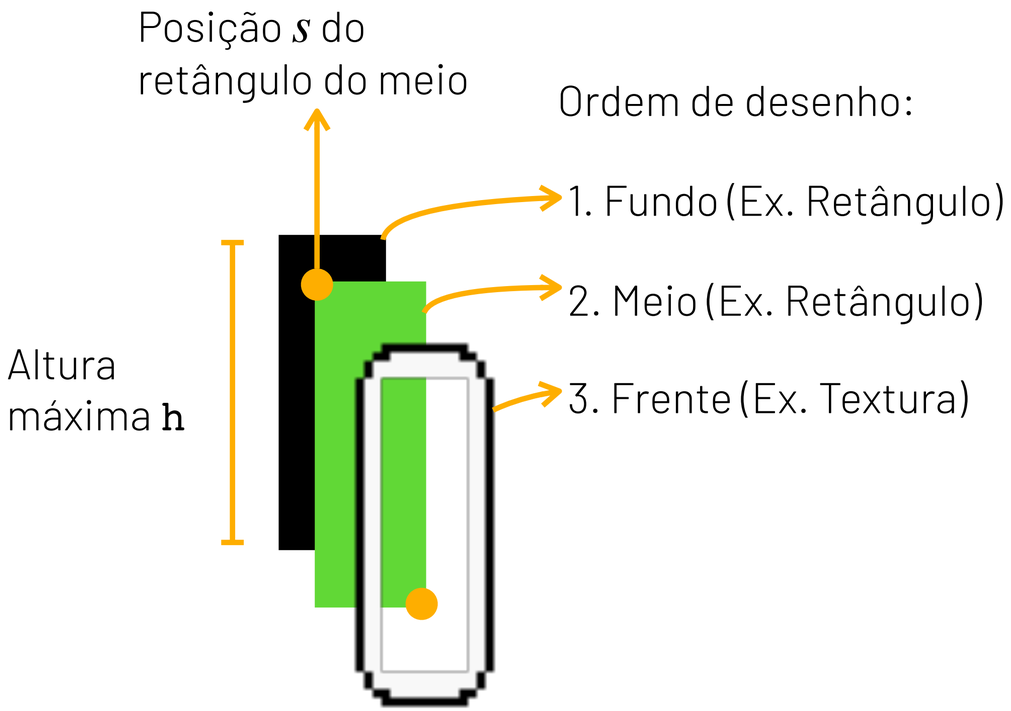
O HUD pode ser implementado como uma "tela de menu" que mostra informações armazenadas nos game objects. De uma maneira mais básica, ele deve suportar imagens e textos:


```
class HUD : public UIScreen
{
public:
   HUD(class Game* game);
   ~HUD();

   void Update(float deltaTime);
   void Draw();

private:
   class UIImage* mRupeeIcon;
   class UIText*  mRupeeCounter;
   class UIImage* mBombIcon;
   class UIText*  mBombCounter;
   class UIImage* mArrowIcon;
   class UIText*  mArrowCounter;
   ...
};
```

- Por exemplo, o objeto **`mRupeeCounter`** é um texto que mostra o número de rupias que o jogador possui.
- Esse número é armazenado no objeto do jogador.

Barras de Progresso

Para implementar barras de progressos, podemos dividí-lá em três partes: o fundo, o meio e a frente. A posição de início do meio da barra será calculada em função da sua altura máxima:



A posição $s.y$ vertical do retângulo do meio deve ser calculada em função de uma porcenragem p (entre 0 e 1) da altura máxima h:

```
void DrawProgressBar(SDL_Texture* front, float p, SDL_Rect &r)
{
   SDL_SetRenderDrawColor(renderer, 0, 0, 0, 255);
   SDL_RenderFillRect(renderer, &r);

   r.y = static_cast<int>(r.y - r.h * p);
   SDL_SetRenderDrawColor(renderer, 0, 255, 0, 255)
   SDL_RenderFillRect(renderer, &r);

   SDL_RenderCopy(r, front, nullptr, &r);
}
```

Relógio

Para implementar um relógio, podemos criar uma variável **`float mTimer`** para contar o tempo em no Update do HUD e a cada segundo, atualizar a string de tempo:


```cpp
void Game::Update(float deltaTime) {
    timer -= deltaTime;
    if (timer < 0.0f) timer = 1.0f;

    int currentSeconds = static_cast<int>(timer);
    if (currentSeconds != lastDisplayedSeconds) {
        lastDisplayedSeconds = currentSeconds;
        HUD->updateTimerTexture();
    }
}

void HUD::DrawTimer(int x, int y) {
    if (timerTexture) {
        timerRect.x = x;
        timerRect.y = y;
        SDL_RenderCopy(renderer, timerTexture, nullptr, &timerRect);
    }
}
```

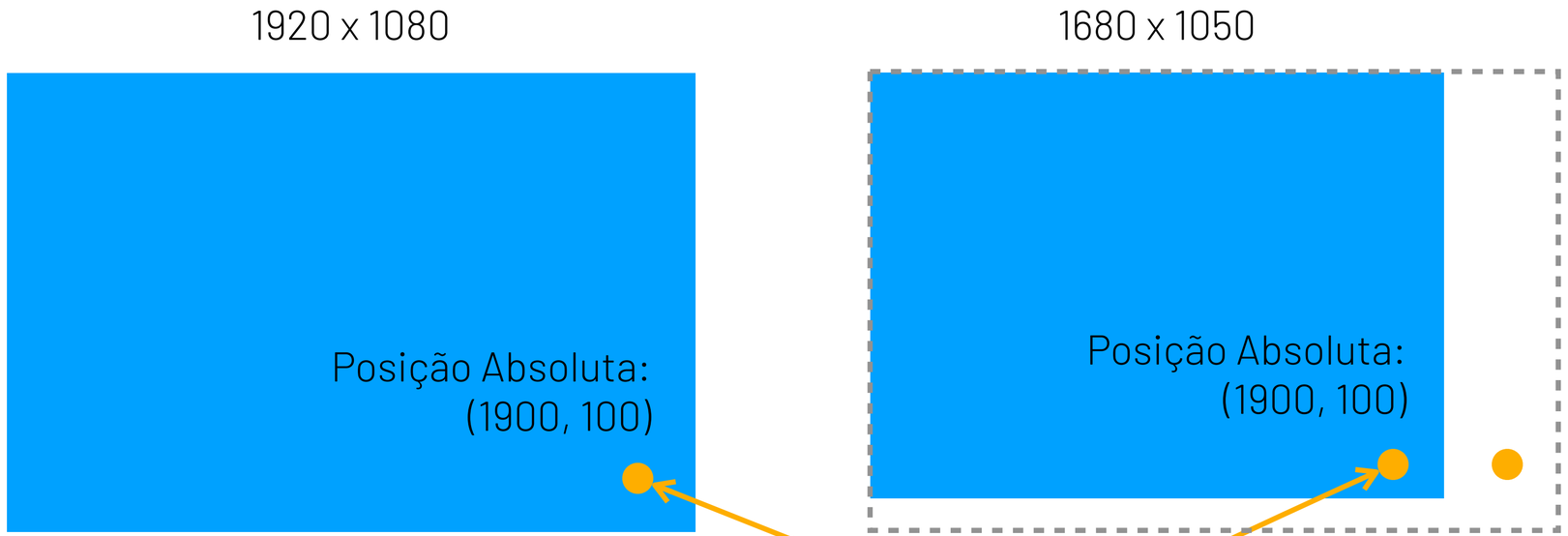
Relógio

Para implementar um relógio, podemos criar uma variável **`float mTimer`** para contar o tempo em no Update do HUD e a cada segundo, atualizar a string de tempo:


```
void HUD::UpdateTimerTexture() {
    if (timerTexture) {
        SDL_DestroyTexture(timerTexture);
        timerTexture = nullptr;
    }

    std::string timeStr = std::to_string(lastDisplayedSeconds);

    SDL_Color color = {255, 255, 255, 255}; // White
    SDL_Surface* surface = TTF_RenderText_Blended(font,
                    timeStr.c_str(), color);
    if (!surface) return;

    timerTexture = SDL_CreateTextureFromSurface(renderer, surface);
    timerRect.w = surface->w;
    timerRect.h = surface->h;

    SDL_FreeSurface(surface);
}
```

Posições Relativas para Resoluções Diferentes

Para suportar diferentes resoluções de monitores, nós geralmente utilizamos **posições relativas** ao invés de posições absolutas para posicionar os elementos de interface:



Posição Relativa: (-100, -100)

Relativa ao canto direito inferior

Se usarmos posições absolutas para posicioar elementos de interface, eles podem ficar fora da tela.

Localização

Uma das formas mais simples para suportar diferentes idiomas no seu jogo, é criar um dicionário para cada idiona, mapeando as mesmas chaves textuais para as strings do jogo:

```
{
   “TextMap”: {
      “NewGame”: “New Game”,
      “LoadGame”: “Load Game”,
      “Options”: “Options”,
      ...
   }
}
```

en_us.json

```
{
   “TextMap”: {
      “NewGame”: “Novo Jogo”,
      “LoadGame”: “Carregar Jogo”,
      “Options”: “Opções”,
      ...
   }
}
```

pt_br.json

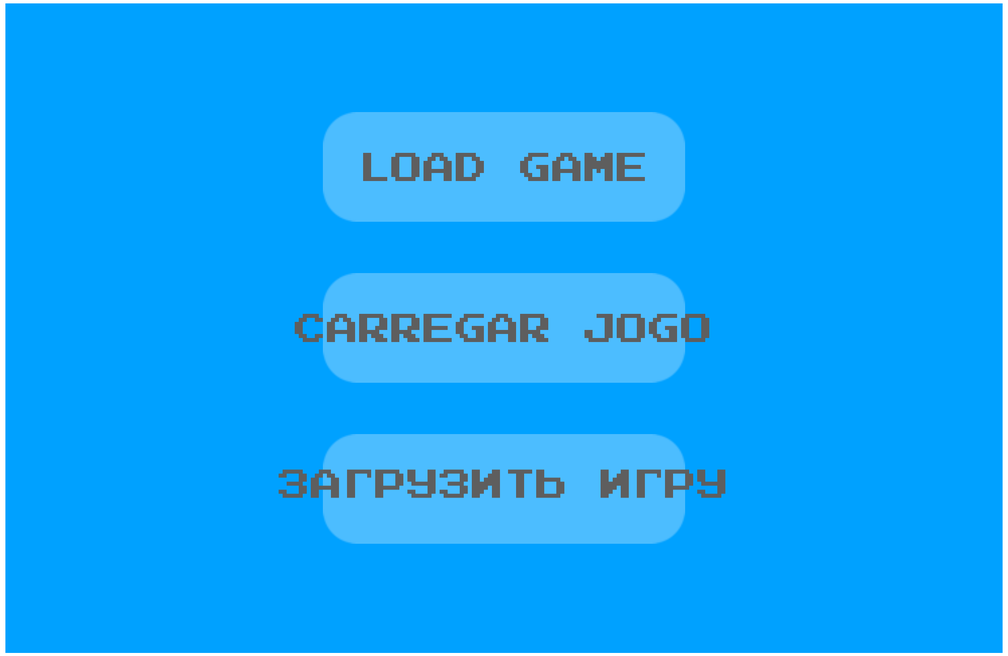
```
{
   “TextMap”: {
      “NewGame”: “Nuevo Juego”,
      “LoadGame”: “Cargar Juego”,
      “Options”: “Opciones”,
      ...
   }
}
```

es_ar.json

Ao invés de usar a string **`“New Game”`** no seu jogo, você irá utilizar a chave associada a essa string para acessar a tradução correta no dicionário **`TextMap[“NewGame"]`**

Localização

Diferentes idiomas podem ter tamanhos diferentes de palavras, portanto a arte da interface deve acomodar esses diferentes tamanhos:

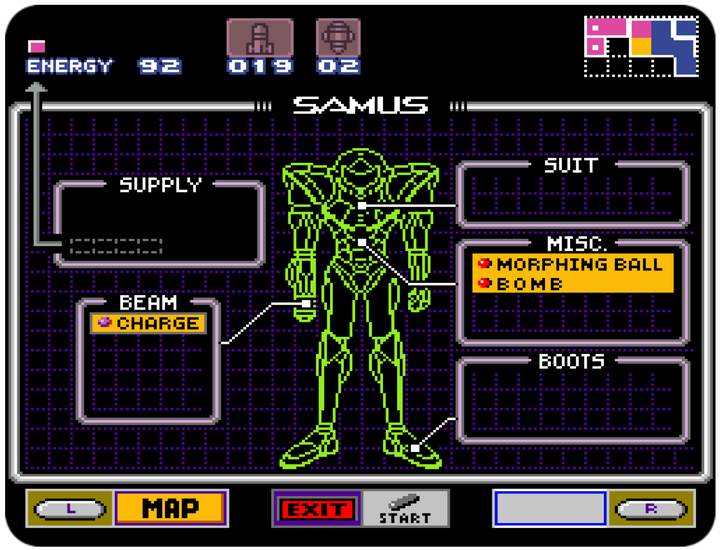


Menu de Pausa

O **menu de pausa** é um menu acionado por meio de um botão no controle ou teclado para congelar o estado jogo, geralmente mostrando algum menu (de tela única ou múltiplas telas):

Super Mario World
Pausa sem menu

The Legend of Zelda: A Link to the Past
Pausa com menu tela única



Super Metroid
Pausa com menu de múltiplas telas

Menu de Pausa

Para congelar o estado do jogo, basta criar um atributo booleno `mIsPaused` na classe `Game` que **desabilita a atualização e processamento de entrada dos *game objects***:

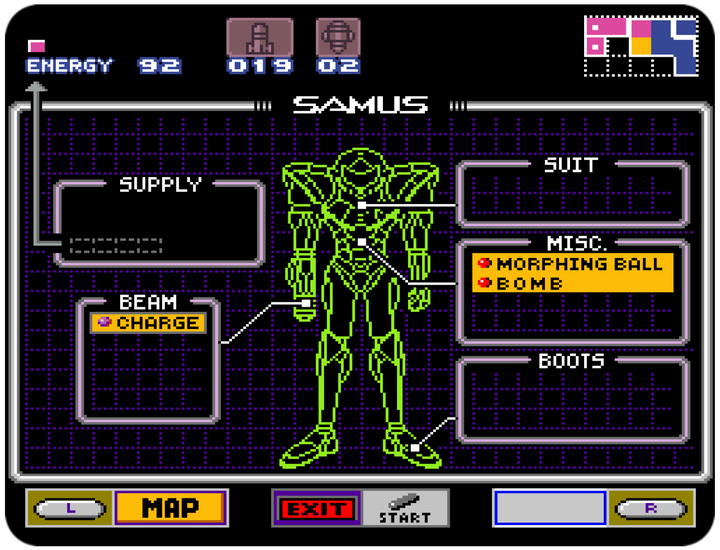
Super Mario World
Pausa sem menu

```
if (!mIsPaused) {
    for (auto actor : mActors) {
        actor->ProcessInput(state);
    }
}
```

```
if (!mIsPaused) {
   UpdateActors(deltaTime);
}
```

Menu de Pausa

Em casos onde o menu de pausa ocupa toda a tela, você pode utilzar a flag `mIsPaused` para desabilitar o desenho dos game objects também:



Super Metroid
Pausa com menu de múltiplas telas

```
if (!mIsPaused) {
    for (auto actor : mActors) {
        actor->ProcessInput(state);
    }
}
```

```
if (!mIsPaused) {
   UpdateActors(deltaTime);
}
```

```
if (!mIsPaused) {
   for (auto actor : mActors) {
        actor->Draw();
    }
}
```

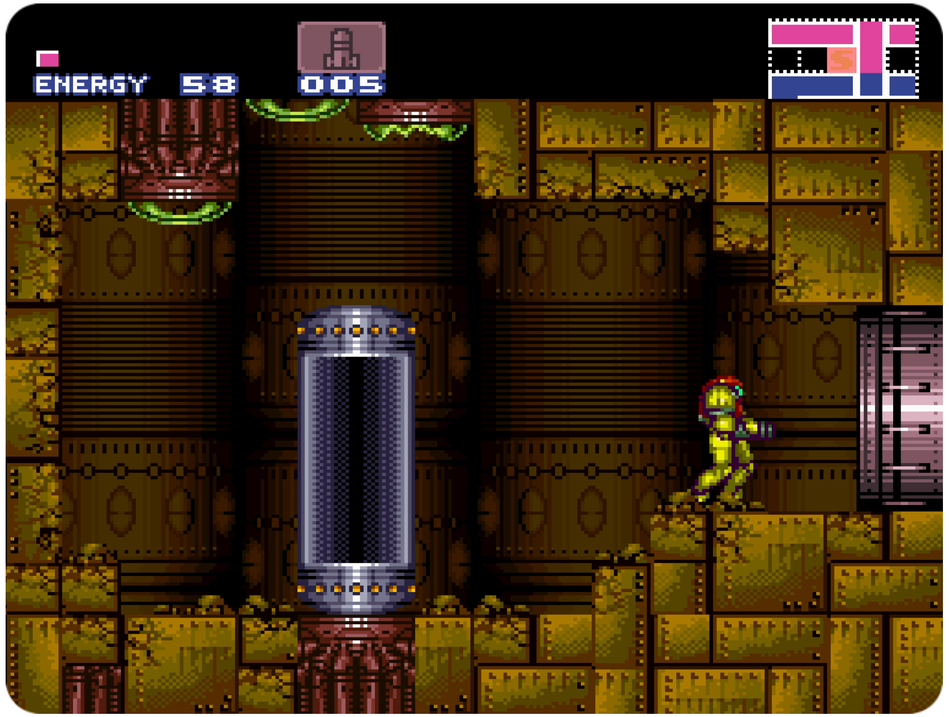
Cenas

Jogos modernos são compostos de múltiplas **cenas** (menus, níveis, áreas de um mapa, ...), onde cada cena possui uma lista de game objects independente.

Por exemplo, no Mega Man X, o jogo é dividido por níveis:

Gerenciamento de Cenas

A função do **sistema de gerenciamento de cenas** é possibilitar funcionalidades básicas de definição e transições de cenas:

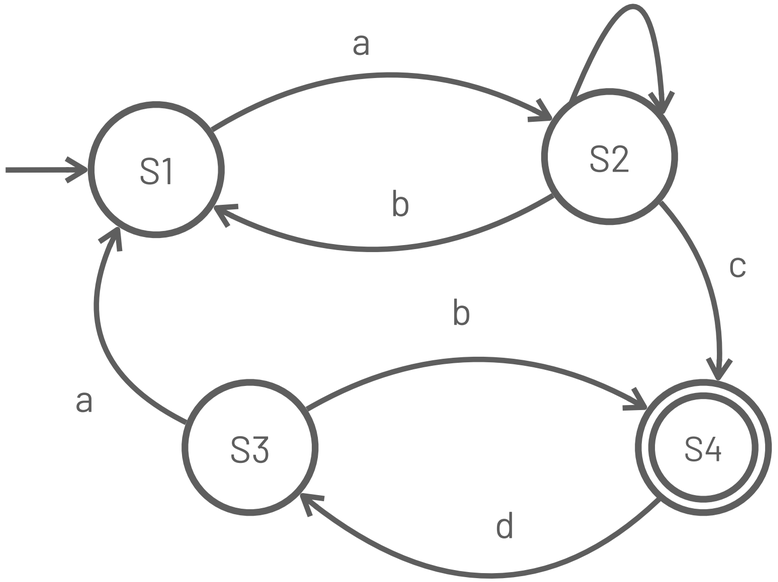


Super Metroid

- Criar e Destruir uma cena
- Transição entre cenas:
  - Descarregar cena atual: Destruir game objects da cena atual
  - Carregar próxima cena: Instanciar game objects da próxima cena
  - Opcionalmente aplicar efeitos de transição (ex. Crossfade)
- Geralmente são implementados como **máquinas de estados finita**

Máquinas de Estados Finitos

Uma **Máquina de Estados Finita** (FSM - *Finite State Machine*) é um modelo matemático tradicionalmente utilizado para representar programas de computador ou circuitos lógicos.



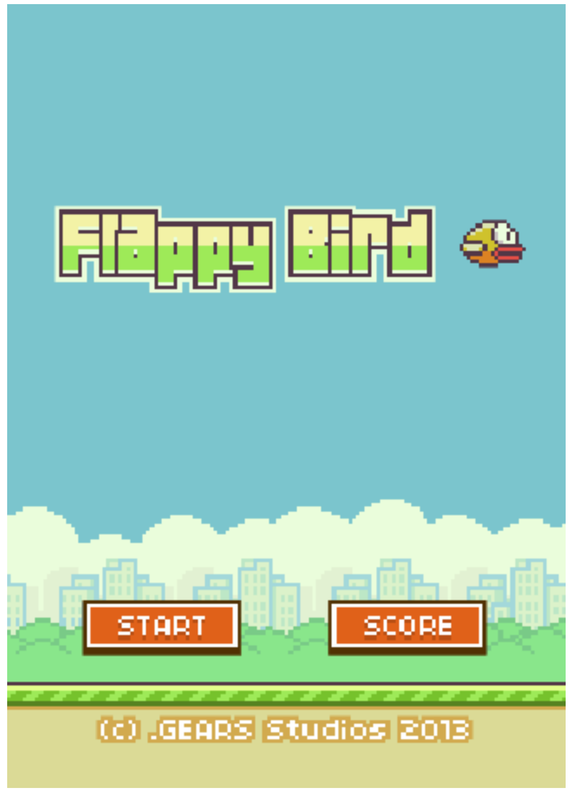
Uma FSM é definida por dois conjuntos:

1. **Estados** que a máquina pode estar (um por vez)
   S1, S2, S3, S4 (final)

2. **Condições** para transições de estados
   a, b, c, d

Exemplo 1: Flappy Bird

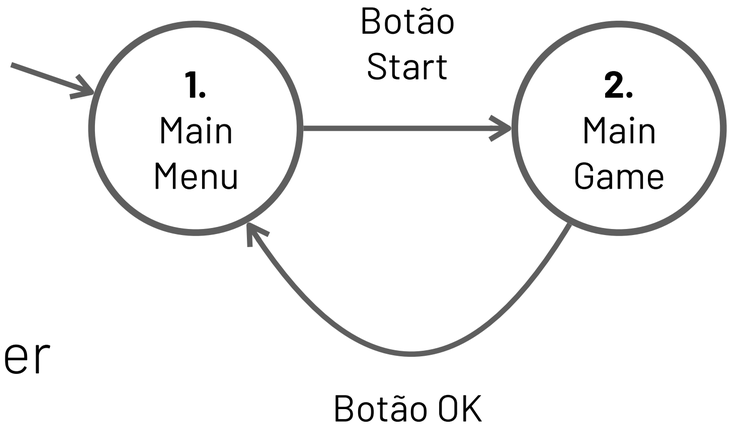
Um exemplo bastante simples de FSM para controle de cenas é o Flappy Bird:



Estados:

1. Menu Principal
2. Main Game

   Note que o Tutorial e o Game Over são telas de menu, e não cenas

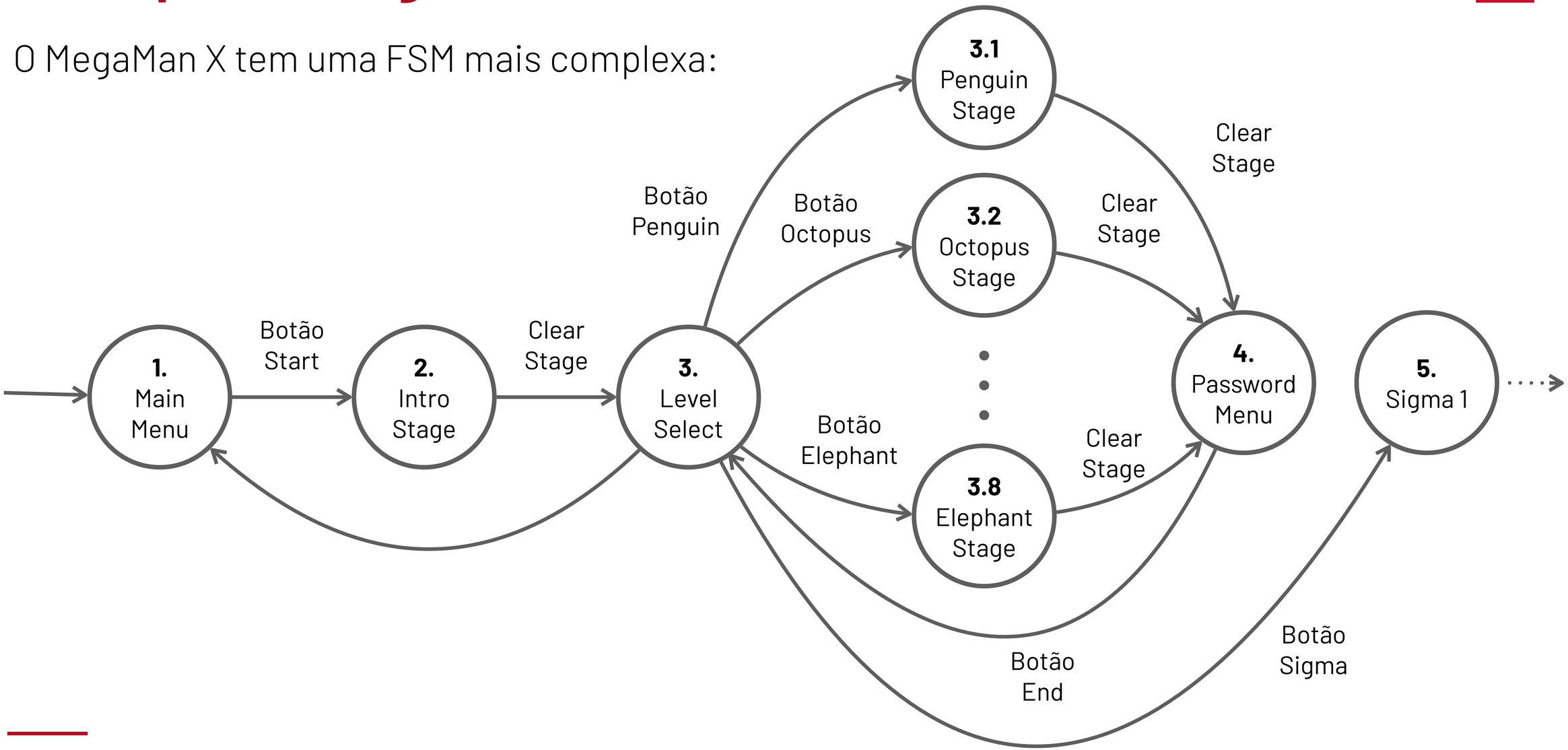


Transições:

- 1→2: Clicar no botão Start
- 2→1: Clicar no botão OK

Exemplo 2: Mega Man X

O MegaMan X tem uma FSM mais complexa:



Implementando FMS

As técnicas mais comuns para implementação de FSMs são:

- Comando Switch/Case
- Padrão de Projeto State
- Interpretadores (mais usado para IA dos personagens)

Implementando FMS com Comando Switch/Case

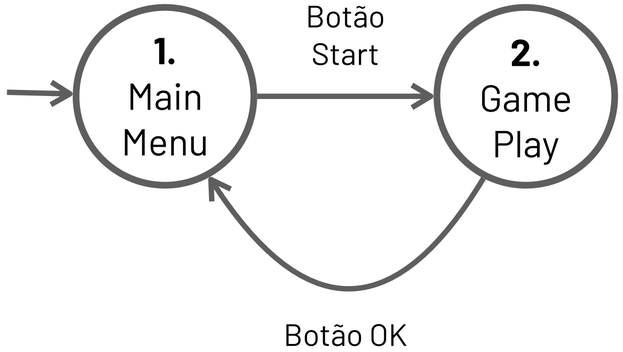
Todas as transições são codificadas em um só lugar, escritas como uma grande verificação condicional com múltiplas ramificações (instruções case em C++).

```
class Game
{
public:

   enum class GameScene
   {
       MainMenu,
       GamePlay,
   };

private:

   GameScene mGameScene;
}
```

Game.h

```
void Game::InitializeActors() {
    switch (mGameScene) {
        case GameScene::MainMenu:
            ...
            break;
        case GameScene::GamePlay:
            ...
            break;
        default:
            break;
    }
}

void Game::SetScene(GameScene gameState)
   UnloadData();
   mGameScene = gameState;
   InitializeActors();
}
```

Game.cpp



Implementando FMS com Padrão State

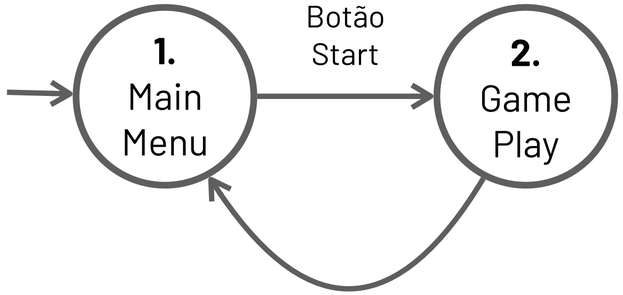
Cada estado é responsável por carregar e descarregar seus dados, além de determinar seu estado sucessor:

```
class GameScene {
public:
    virtual void Enter(Game *game) {};
    virtual void Update(Game *game) {};
    virtual void Exit(Game *game) {};
private:
    float mStateTime;
};

class MainMenuScene : public GameScene {
public:
    void Enter(Game *game);
    void Update(Game *game);
    void Exit(Game *game);
};
```

GameScene.h

```
class Game {
   ...
   GameScene mGameState;
   ...
}
```

Game.h




```
void Game::Update(GameScene gameState) {
   …
   mGameState->Update();
}

void Game::SetScene(GameScene gameState)
   mGameState->Exit(); // Unload current Data
   mGameState = gameState;
   mGameState->Enter(); // Load new data
}
```

Game.cpp

Efeitos de Transição: CrossFade

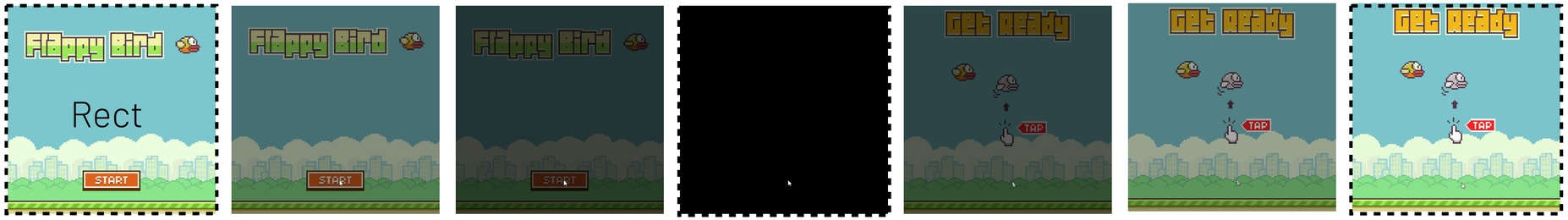
Para que a transição não seja muito abrupta, muitos jogos implementam um **efeito de crossfade** que é aplicado durante a transição de cenas:

```
bool Game::Initialize() {
   ...
   mGameState = GameScene::MainGame;
   InitializeActors();
}
```

```
UnloadGameData();
mGameState = GameScene::MainGame;
InitializeActors();
```

1. Criar um Rect Preto com alpha = 0% no momento da transição (Ex. Botão Start)

alpha = 0% | alpha = 33% | alpha = 66% | alpha = 100% | alpha = 66% | alpha = 33% | alpha = 0%



2. Aumentar alpha em função do tempo de fade out

Tempo de fade out (ex. 1s)

3. Diminuir alpha em função do tempo de fade in

Tempo de fade in (ex. 1s)

Efeitos de Transição: CrossFade

No método *Draw*, calculamos o alpha em função do tempo percorrido desde o início do efeito `mfadeTime` e da duração do efeito `FADE_IN_TIME` ou `FADE_OUT_TIME`:

```
void Game::Draw()
{
    if (mFadeState == FadeState::FadeOut)
    {
        float alphaOut = mfadeTime/FADE_OUT_TIME;
        SDL_SetRenderDrawBlendMode(mRenderer, SDL_BLENDMODE_BLEND);
        SDL_SetRenderDrawColor(mRenderer, 0, 0, 0, 255 * alpha);
        SDL_RenderFillRect(mRenderer, nullptr);
    }
    else if (mFadeState == FadeState::FadeIn)
    {
        float alphaIn = mfadeTime/FADE_IN_TIME;
        SDL_SetRenderDrawBlendMode(mRenderer, SDL_BLENDMODE_BLEND);
        SDL_SetRenderDrawColor(mRenderer, 0, 0, 0, 255 * (1.0f - alphaIn));
        SDL_RenderFillRect(mRenderer, nullptr);
    }
}
```

Na função *Update*, contamos o tempo desde o início do efeito com `mfadeTime`:

```
void Game::Update(float deltaTime)
{
    if (mFadeState == FadeState::FadeOut)
    {
        mfadeTime += deltaTime;
        if (mfadeTime >= FADE_OUT_TIME)
        {
            mfadeTime = 0.0f;
            mFadeState = FadeState::FadeIn;
            UnloadData();
            InitializeActors();
        }
    }
    else if(mFadeState == FadeState::FadeIn)
    {
        mfadeTime += deltaTime;
        if (mfadeTime >= FADE_IN_TIME) {
            mfadeTime = 0.0f;
            mFadeState = FadeState::None;
            mIsPaused = false;

            mCamera->Set(.0f, .0f,);
            mScene->Enter();
        }
    }
}
```

Próxima aula

A16: Síntese de Áudio

- Sinais de Áudio
- Espectrogramas
- Soma de Sinais
- Sintetizando Músicas com Sequenciadores