DCC192

2025/1

UFMG

# Desenvolvimento de Jogos Digitais

## A11: Interface com o usuário

Prof. Lucas N. Ferreira

# 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

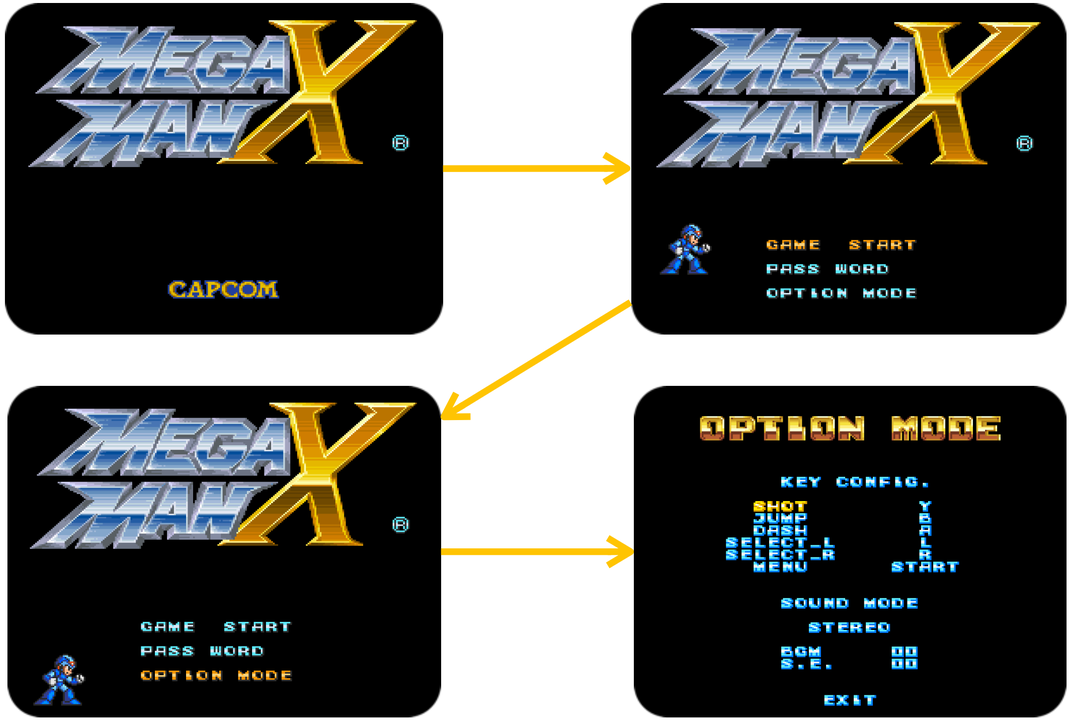
O **Menu Principal** é a primeira tela que um jogador vê ao iniciar um jogo. Ela atua comum uma interfáce introdutória, geralmente oferencendo alguas opções básicas, como:


- [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 pilha de telas é  renderizada de baixo para cima, por isso implementamos como um vetor

# Classe base de menus (UIScreen)

```
class UIScreen {
public:
  enum UIState { Active, Closing };

  virtual void Update();
  virtual void Draw();
  virtual void ProcessInput();

  void Close();

private:
  UIState mState;
  std::string mTitle;
  std::list<UIButtons *> mButtons;
}
```

**Funções**:

- `Update`: atualizar o estado da tela
- `Draw`: desenhar a tela
- `ProcessInput`: processar eventos de entrada

**Atributos**:

- `mState`: estado da tela (Ativo/Fechando)
- `mTitle`: título da tela
- `mButtons`: lista (duplamente ligada) de botões

- Teclas para navegar na lista duplamente ligada (next, prev)
- Mouse para clicar em um botão específico

# Botões

```
class UIButton {
public:
  bool ContainsPoint(Vector2& pt);
  void OnClick();

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

  Vector2 mPosition;
  Vector2 mDimensions;
}
```

**Funções**:

- `ContainsPoint`: verificar se um ponto (ex., cursor do mouse) está dentro do botão
- `OnClick`: função chamada quando o botão é clicado

**Atributos**:

- `mOnClick`: função que executa a ação do botão
- `mName`: nome do botão
- `mPosition`: posição do botão (cima/esquerda)
- `mDimensions`: dimensões do botão

# Fontes

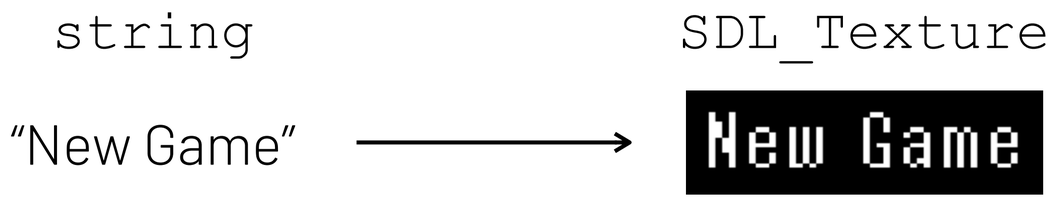
**Fontes vetoriais** são tipicamente implementadas desenhando contornos de caracteres individuais (*glifos*) com segmentos de retas e curvas de Bézier.

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 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);

if (font == nullptr) {
    SDL_Log("Failed to load font %s in size %d: %s", fileName.c_str(), size, TTF_GetError());
    return false;
}

return font;
```

# Desenhando fontes vetoriais em SDL

Para desenhar fontes vetoriais em SDL, precisamos gerar uma textura em SDL:

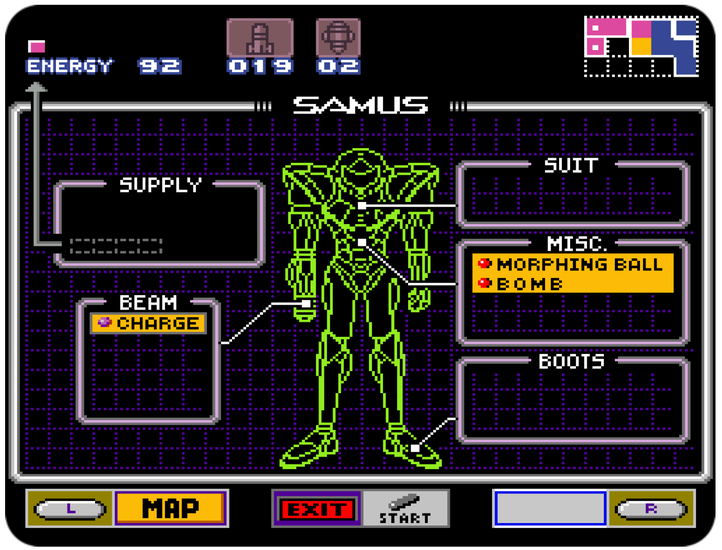
```
int wrapLength = 900;
SDL_Color color = {.r =  21, .g =  21, .b = 123, .a =  255};

SDL_Surface* surf = TTF_RenderUTF8_Blended_Wrapped(font, “New Game", sdlColor, wrapLength);
if (!surf) {
     SDL_Log("Failed to create surface for text: %s", “New Game");
    return nullptr;
}

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

if (!texture) {
  SDL_Log("Failed to render surface to texture");
   return nullptr;
}

return texture;
```

# 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***:

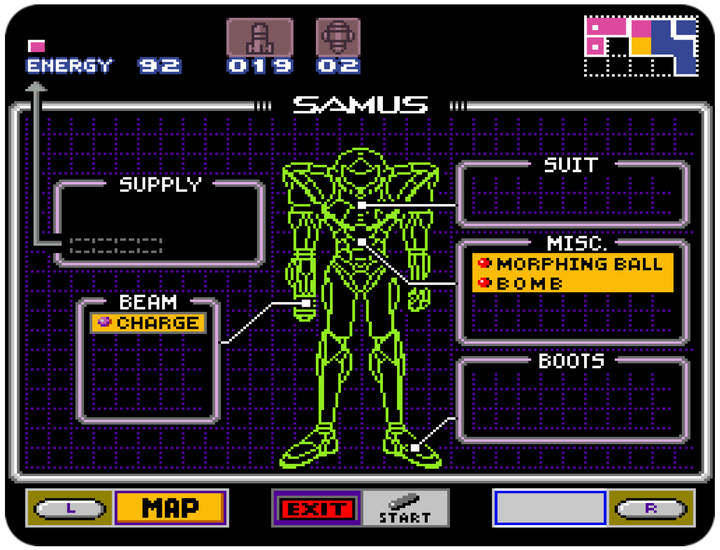

Super Mario World
Pausa sem menu

```
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();
    }
}
```

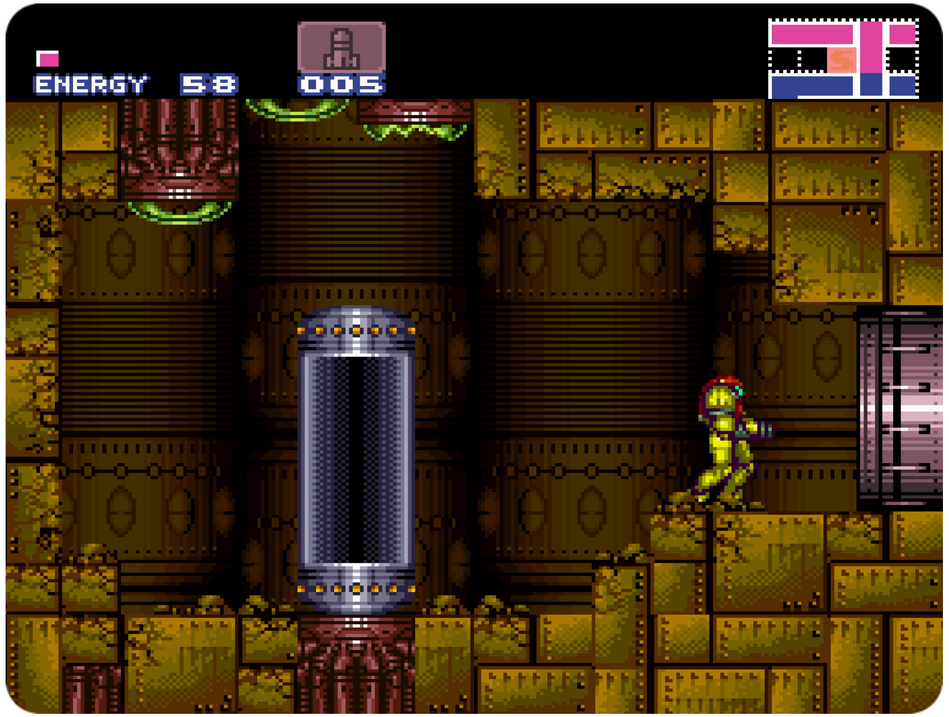
# Cenas

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:

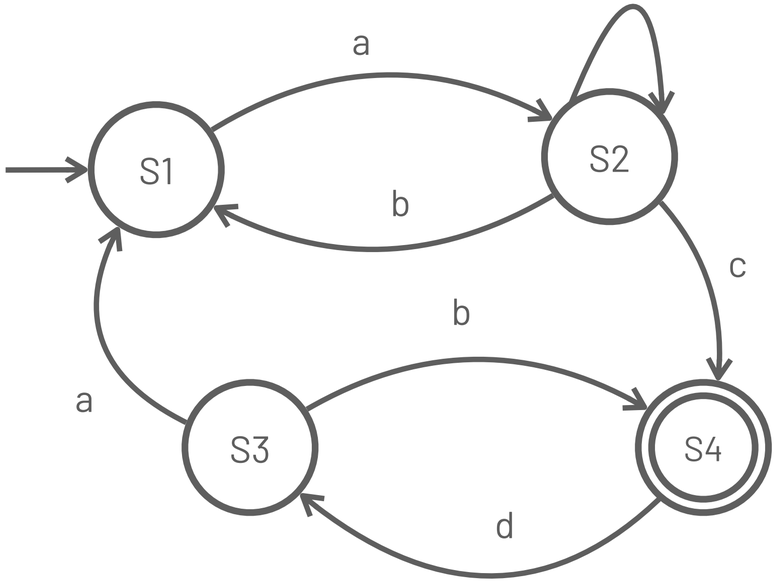



Super Metroid

- 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.



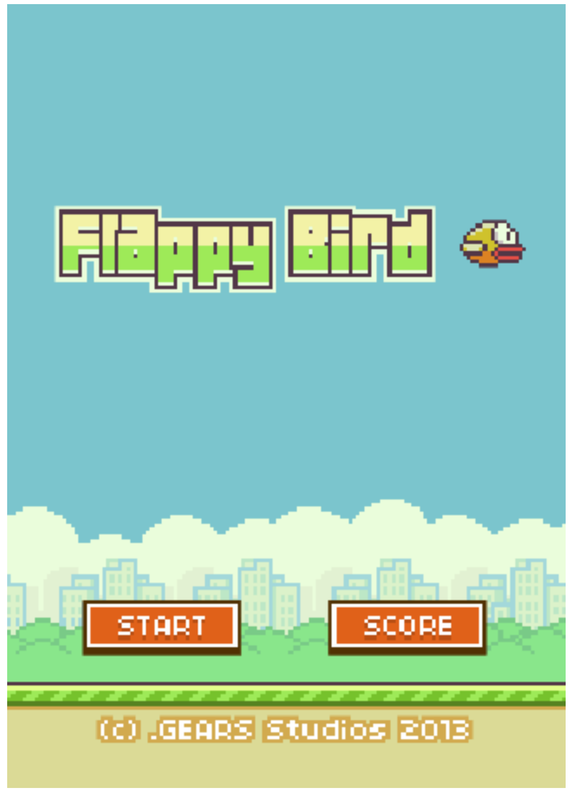

Uma FSM é definida por dois conjuntos:

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

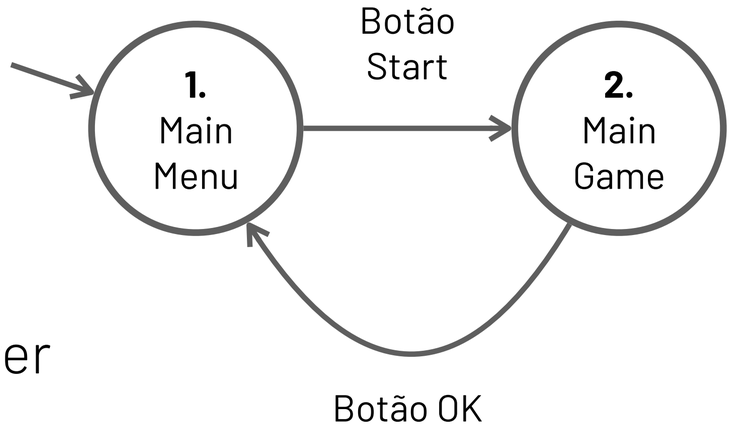
Um exemplo bastante simples de FSM para controle de cenas é o Flappy Bird:




**Estados**:

1. Menu Principal
2. Main Game

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

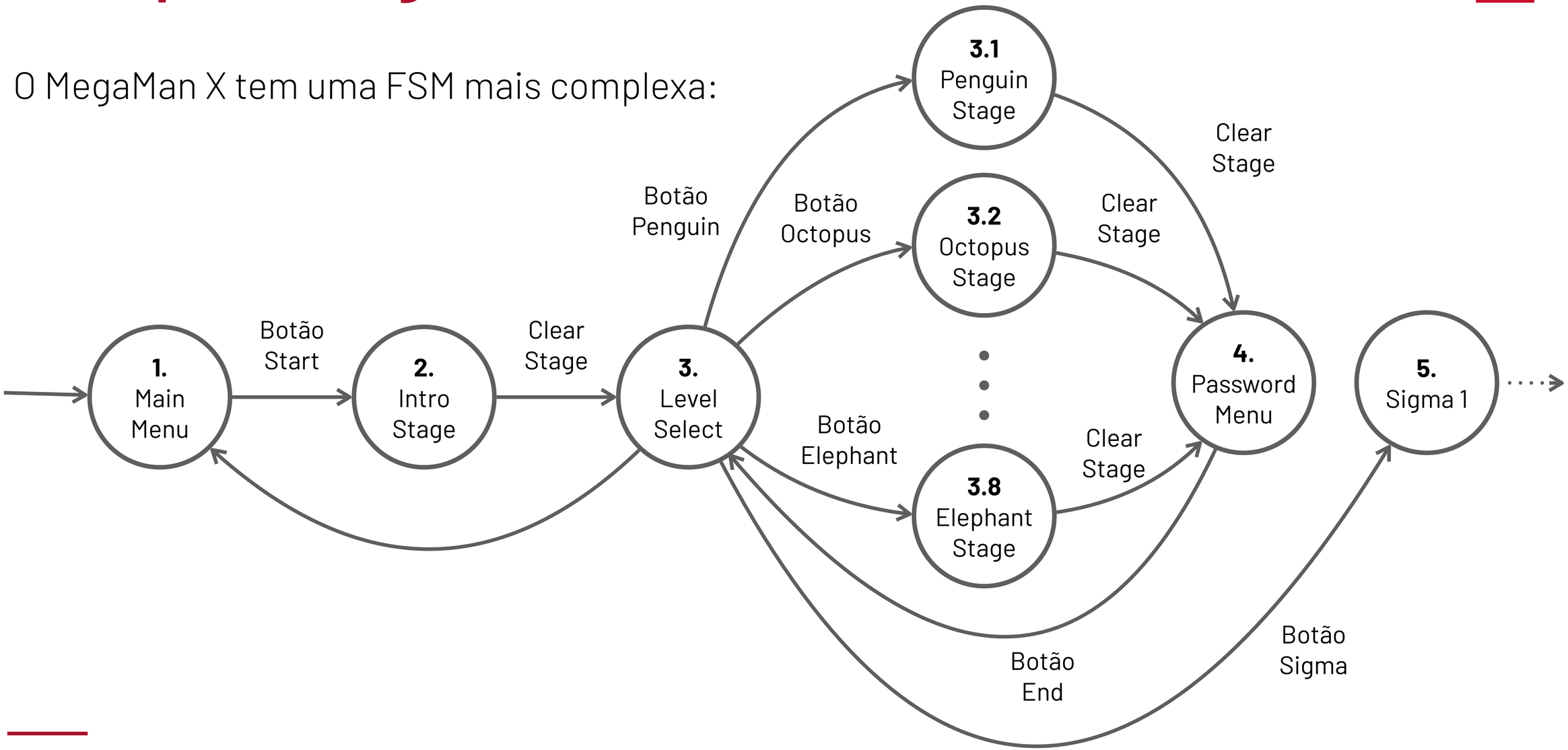



**Transições**:

- 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

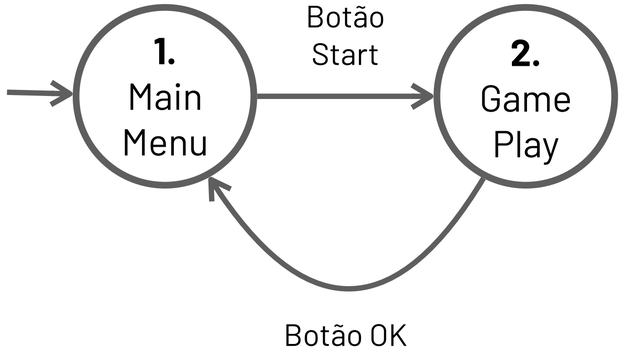
Todas as transições são codificadas em um só lugar, escritas como uma grande verificação condicional com múltiplas ramificações (instruções case em C++).

```
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

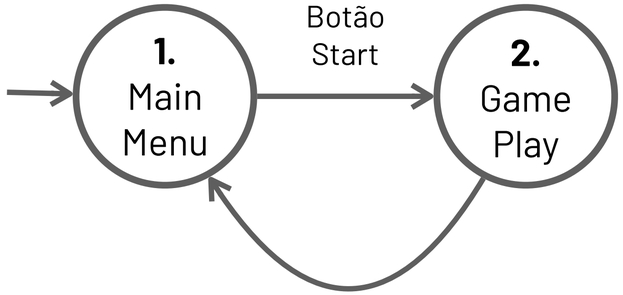
Cada estado é responsável por carregar e descarregar seus dados, além de determinar seu estado sucessor:

```
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

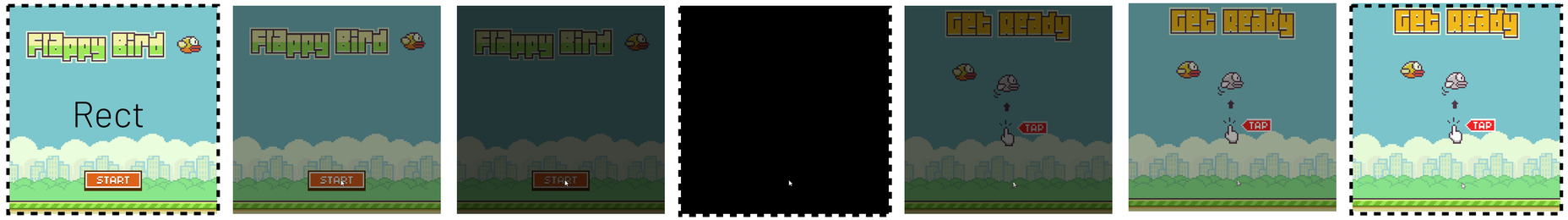
# Efeitos de Transição: CrossFade

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();
```

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




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

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

## A12: Fundamentos de Game Design

- Definições de Jogos
- Regras e Objetivos
- Mecânicas
- Dinâmicas
- Game Design Document
- Playtesting