DCC192

2025/1

UFMG

Desenvolvimento de Jogos Digitais

A18: Heads-Up Display

Prof. Lucas N. Ferreira

Plano de aula

- Implementação de HUD como interface
  - UIScreen, UIText e UIImage
- Barra de Progresso
- Relógio (timer)
- Suportando múltiplas resoluções
- Localização

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.

UIScreen: Classe base de menus

Em aulas anteriores vimos que a interface gráfica do jogo é implementada através de *telas*, que não são vistas como game objects:

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

  virtual void Update(float deltaTime);
  virtual void Draw(class Shader* shader);
  virtual void ProcessInput(const uint8_t* keys);
  virtual void HandleKeyPress(int key);

  enum UIState { EActive, Eclosing };
  UIState GetState() const { return mState; }
  void Close();

  // Change the title text
  void SetTitle(const std::string& text,
                const Vector3& color,
                int pointSize = 40);

   void AddButton(const std::string& name,
   std::function<void()> onClick);
protected:
    // Helpers/member data...
};
```

Funções:

- `Update`: atualizar o estado da tela
- `Draw`: desenhar a tela
- `ProcessInput`: verificar quais teclas estão pressionadas
- `HandleKeyPress`: processar eventos individuais (up, down)
- `AddButton`: adicionar um botão na tela

Atributos:

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

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:
   bool Load(const std::string& text);
   void Unload();

   void SetActive(int index = 0);

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

private:
   std::string mText;
   SDL_Texture* mTextTexture;
   Vector2 mPosition;
   int mWidth;
   int mHeight;
};
```

Funções:

- `Load`: Cria uma textura a partir de um texto
- `Unload`: Descarrega textural aual

Atributos:

- `mText`: string contendo o texto atual
- `mPosition`: posição do texto na interface
- `mWidth`: largura do texto na inteface
- `mHeight`: altura do texto na interface

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);
```

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:
   std::string mFileName;
   Vector2 mPosition;
   int mWidth;
   int mHeight;
};
```

Funções:

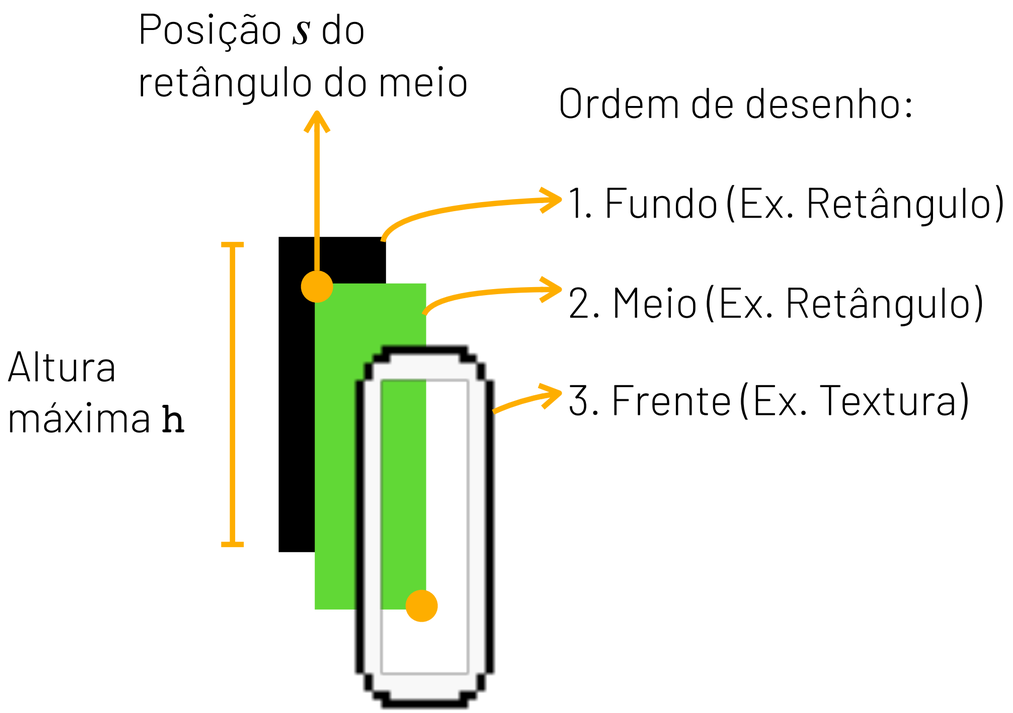
- `Load`: Cria uma textura a partir de um arquivo de imagem
- `Unload`: Descarrega textura aual

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

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);
    }
}
```

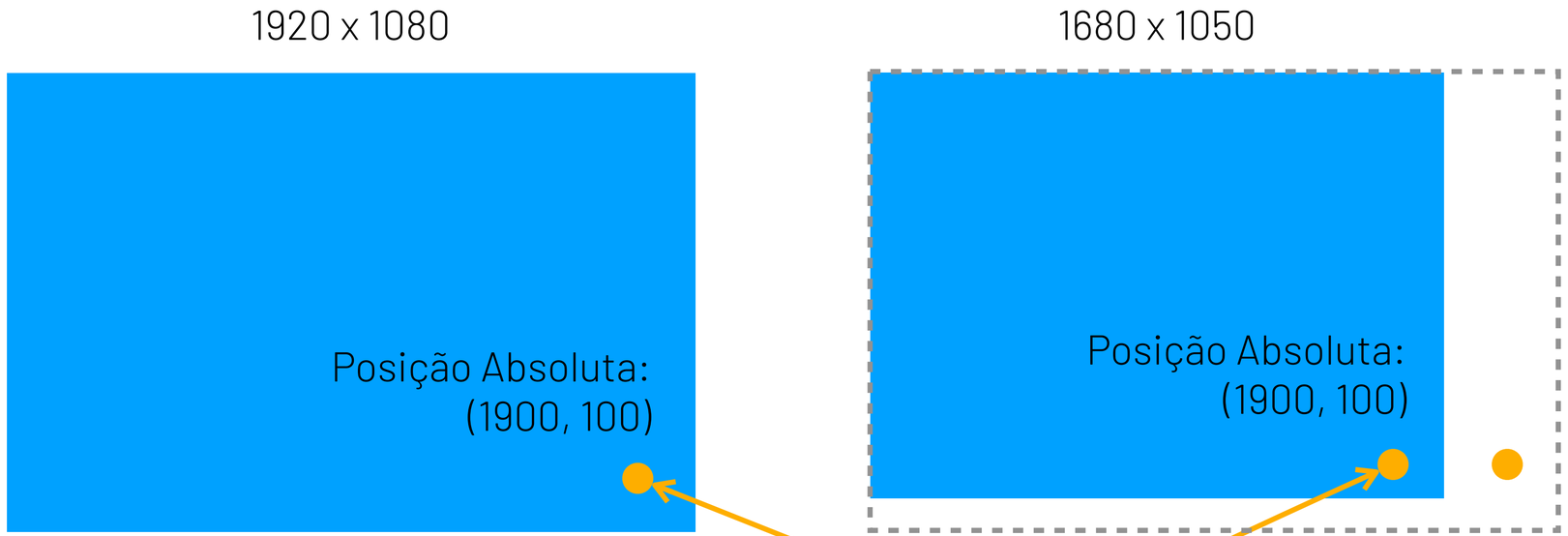
Relógio

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

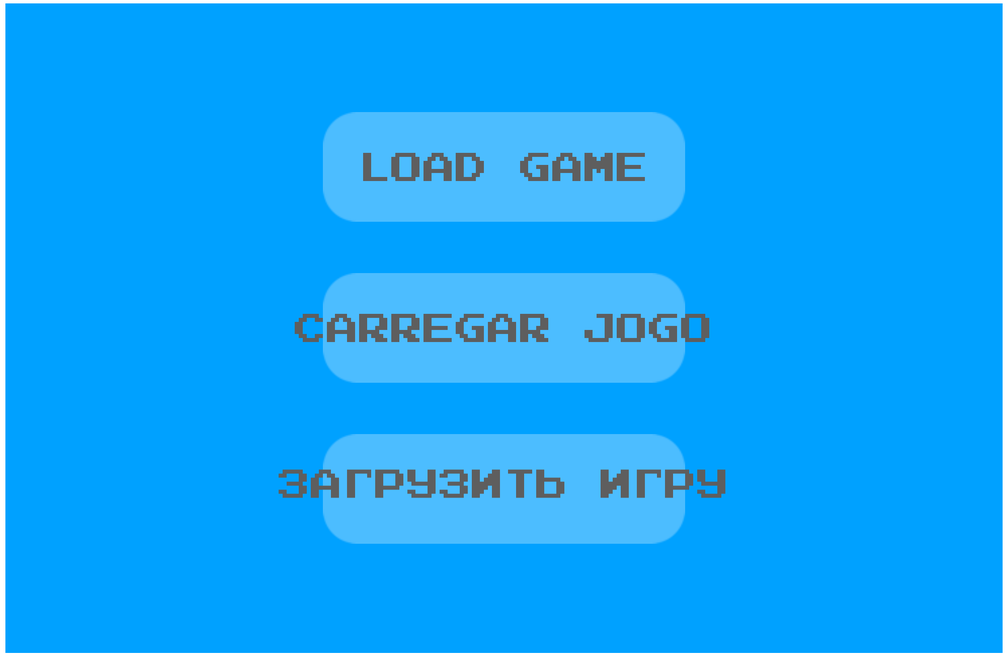
```
{
   “TextMap”: {
      “NewGame”: “Nuevo Juego”,
      “LoadGame”: “Cargar Juego”,
      “Options”: “Opciones”,
      ...
   }
}
```

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:



Próxima aula

A18: Inteligência Artificial I

- IA acadêmica vs. IA para jogos
- Máquina de Estados Finita para NPCs
- PacMac PostMortem
  - Estados
  - Transições
  - Ajuste de Dificuldade