Curso Tecnológico em Jogos Digitais

# Prática em Laboratório UNITY
# Máquina de Estado Finito (The Adventurer)

Murilo Boratto

## 1 Resumo

Esta prática tem como finalidade remodelar as habilidades de um personagem 2D, o qual denominaremos *The Adventurer*. A estrutura do personagem principal consiste em um herói o qual adicionaremos múltiplos comportamentos a partir dos conceitos de **Máquinas de Estado Finito (MEF)**. Para esta cena o *player* inicializa com alguns comportamentos, já pré-programados, como por exemplo: Parado (*IDLE*), correr (*RUN*), pular (*JUMP*) e cair (*GLIDE*). Sendo assim, a idéia básica desta prática é adicionar o maior número de comportamentos possíveis ao personagem. Este roteiro é baseado no tutorial do Livro *UNITY AI Programming* [1].

## Parte 1 - Construção da cena do jogo *The Players Tank*

1. O ponto inicial será fazer o download do projeto no *google classroom* da disciplina. Após descompactalo, haverá uma pasta chamada *TheAdventurer*.

2. Abrimos o UNITY HUB. Adicionamos o projeto *TheAdventurer*, o qual consta os assets básicos do nosso jogo. Abriremos o projeto 2D no motor de jogo UNITY.

3. Com o UNITY aberto na aba SCENES, abra uma cena chamada: *StateMachine*.

4. Na hierarquia da cena *StateMachine*, há múltiplos objetos que a compõem e o nosso *player*. Podemos explica-los a seguir:

   - ◇ Main Camera (Câmera de perspectiva da cena);
   - ◇ Ground (Objeto rígido que serve como chão);
   - ◇ LeftWall (Objeto rígido muro esquerdo);
   - ◇ RightWall (Objeto rígido muro direito);
   - ◇ Platform (Objeto rígido plataforma);
   - ◇ PlayerHeroPrefab (Prefab do *player* no estado inicial (*IDLE*)).

5. Na estrutura de diretórios a estrutura do conteúdo está estruturada da seguinte forma:

   **Animations** Diretório contendo as animações e o elemento *PlayerAnimatorControler*.

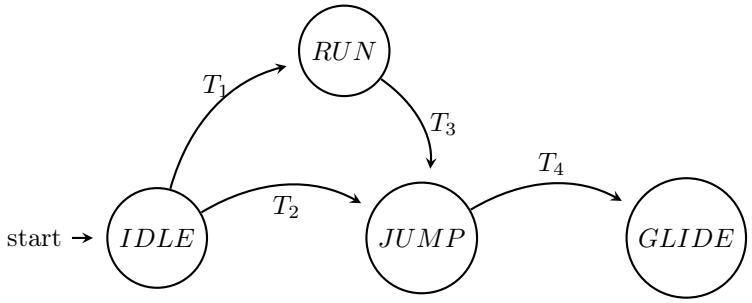
   **Prefabs** Diretório contendo o prefab do *player* com o comportamento inicial de *Idle*.

   **Scenes** Diretório contendo a cena inicial do jogo.

   **Scripts** Diretório contendo o script com a MEF do personagem denominado *PlayerStateMachine.cs*.

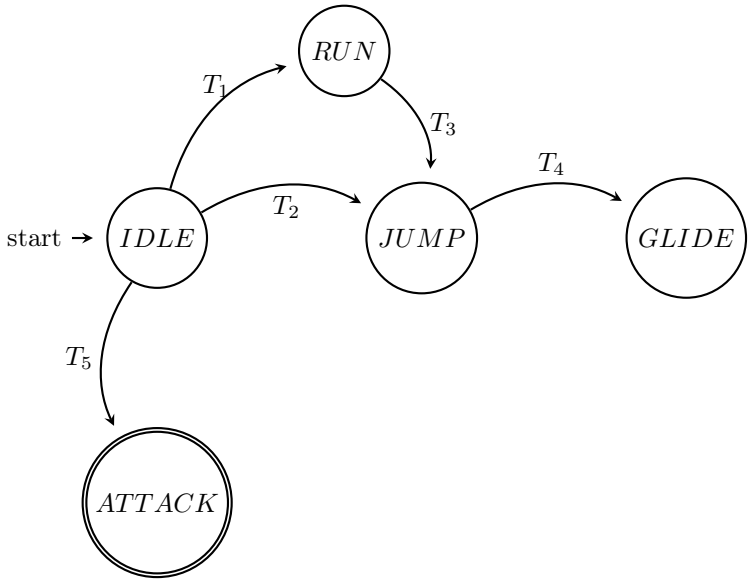
   **Sprites** Diretório contendo os sprites com os múltiplos comportamentos do *player*.

6. Apertamos o *play* e assim completamos, a etapa básica do projeto de testar os 4 estados básicos do personagem. O diagrama a seguir representa os estados e as transições da MEF do *player*:




## Parte 2 - Inserção do Comportamento (*ATTACK*) na Máquina de Estado

1. Uma vez testado e entendido o script fonte dos comportamentos do personagem principal, percebe-se que o (*player*) não possui o evento *ATTACK*, sendo assim se propõe a adição deste comportamento à MEF. Dando um duplo click no script *PlayerStateMachine.cs*, o editaremos para lograrmos esta adição. O novo diagrama da máquina de estado ficará assim:




2. Primeiro adicionaremos o comportamento privado `attackDuration`, o qual será exposto na aba do script, através da característica *SerializeField*. Adicionaremos também o novo estado dentro da lista de enumerados (*enum*) o comportamento *attack* que estará associado a animação *Attack.anim*, o qual

deverá ser editado e associado ao elemento `PlayerAnimatorController.controller`. Setaremos para inicializar este comportamento através da tecla `E`:

```
using UnityEngine;

[RequireComponent(typeof(Animator), typeof(Rigidbody2D), typeof(SpriteRenderer))]
public class PlayerStateMachine : MonoBehaviour
{
    [Header("Settings")]
    [SerializeField] float jumpYVelocity = 8f;
    [SerializeField] float runXVelocity = 4f;
    [SerializeField] float raycastDistance = 0.7f;
    [SerializeField] LayerMask collisionMask;
    [SerializeField] float attackDuration = 1f;

    Animator animator;
    Rigidbody2D physics;
    SpriteRenderer sprite;

    enum State { Idle, Run, Jump, Glide, Attack }

    State state = State.Idle;
    bool isGrounded = false;
    bool jumpInput = false;

    bool isAttack = false;

    float horizontalInput = 0f;

    void FixedUpdate()
    {
        // reset last frame input
        isGrounded = jumpInput = false;
        horizontalInput = 0f;

        // get latest player input
        isGrounded = Physics2D.Raycast(transform.position,
                     Vector2.down, raycastDistance, collisionMask).collider != null;
        jumpInput = Input.GetKeyDown(KeyCode.Space);
        horizontalInput = Input.GetAxisRaw("Horizontal");
        isAttack = Input.GetKeyDown(KeyCode.E);
...
```

3. A estrutura condicional *switch* também deverá conter um evento para o comportamento *attack*. Adicionaremos a linha "`case State.Attack:AttackState();break;`" ao script:

```
...
        if (horizontalInput > 0f)
            sprite.flipX = false;

        if (horizontalInput < 0f)
            sprite.flipX = true;

        switch (state)
        {
            case State.Idle:   IdleState();  break;
            case State.Run:    RunState();   break;
            case State.Jump:   JumpState();  break;
            case State.Glide:  GlideState(); break;
            case State.Attack: AttackState();break;
        }
...
```

4. No diagrama de estados sugerido o personagem somente poderá atacar quando estiver parado. Sendo assim, no módulo `IdleState`, adicionaremos a transição da ação, supondo que o personagem somente poderá atacar quando estiver no chão e a tecla `E` for acionada.

```
...
    void IdleState()
    {
        // actions
        animator.Play("Idle");

        // transitions
        if (isAttack && horizontalInput == 0f)
          state = State.Attack;

        else if (isGrounded)
        {
            if (jumpInput)
                state = State.Jump;
            else if (horizontalInput != 0f)
                state = State.Run;
        }
    }

...
```

## Parte 3 - Proposição de Novos Comportamentos à Máquina de Estado

1. A proposta da continuação desta prática é a partir dos assets contidos na pasta Sprites, inserir o maior número possível de estados para o *player*, como por exemplo:
   - ◇ *CLIMBER*
   - ◇ *CROUCH*
   - ◇ *FALL*
   - ◇ *STAND*
2. Estas adições de novos comportamentos serão traduzidos em um novo diagrama de estado e deverão ser traduzidos via codificação UNITY.
3. As transições dos comportamentos ficam livres de uma ordem pré-definida, estando à cargo do programador.

## Referências

[1] Aversa, David and Kyaw, August Sithu and Peters Clifford. *Unity AI Programming*. 2018.