Curso Tecnológico em Jogos Digitais

Prática em Laboratório UNITY Comportamentos de Direção (Steering Behaviors)

Murilo Boratto

1 Resumo

Uma necessidade fundamental de quase todos os jogos digitais é a caracterização de personagens NPCs. Em mundos virtuais estes comportamentos são as conduções das habilidades de um personagem autônomo em navegar para um destino e/ou objetivo, através de um ambiente que inclua objetos estáticos e/ou dinâmicos. Sendo assim, esta prática tem como finalidade compor o comportamento de alguns personagens através dos Comportamentos de Movimentação (CM), ou conhecidos da língua inglesa como *Steering Behaviours*. A idéia base é adicionar a personagens comportamentos de direção (SEEK, ARRIVE e EVADE) e simular as caracterizações de um NPC. Este roteiro é baseado no tutorial do Livro *UNITY AI Programming* [1].

Parte 1 - Construção da cena do jogo *TheSteeringBehavior*

1. O ponto inicial será fazer o *download* do projeto no *google classroom* da disciplina. Após descompacta-lo, haverá uma pasta chamada *TheSteeringBehavior*.
2. Abrimos o UNITY HUB. Adicionamos o projeto *TheSteeringBehavior*, 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: *steering-behavior*.
4. Na hierarquia da cena *steering-behavior*, iremos inserir múltiplos objetos que a comporão nosso ambiente. Insiriremos os seguitnes elementos:
   - ◇ **Main Camera** - Que corresponde à câmera de perspectiva da cena;
   - ◇ **NPC1** - Que terá os comportamentos SEEK e ARRAY, e representará o personagem que persegue o Player Control;
   - ◇ **NPC2** - Que terá o comportamento EVADE, e representará o personagem que foge do Player Control;
   - ◇ **PlayerControl** - Personagem controlado pelas setas do teclado.
5. A base é criarmos estes personagem através GameObject > CreateEmpty. E associaremos nas próximas seções os comportamentos via script.

6. Apertamos o *play* e como não inserimos os comportamentos nada acontecerá. Assim completamos, a etapa básica do projeto de testar os comportamentos básicos dos personagens.

Parte 2 - Inserção do script *CameraController* da Câmera

1. Associaremos o script *CameraController.cs* a seguir ao elemento **Main Camera**.

```
using System.Collections.Generic;
using UnityEngine;

[RequireComponent(typeof(Camera))]
public class CameraController : MonoBehaviour
{
    [Header("Settings")]
    [SerializeField] float smoothFactor = 1f;
    [SerializeField] List<Transform> targets = null;

    Camera cam = null;

    void LateUpdate()
    {
        int count = targets.Count;
        Vector3 center = Vector3.zero;
        Bounds bounds = new Bounds();

        foreach (var t in targets)
        {
            center += t.position;
            bounds.Encapsulate(t.position);
        }
        center /= count;
        center = new Vector3
        (
            x: center.x,
            y: center.y,
            z: -10f
        );
        var cameraSize = Mathf.Max(bounds.size.x, bounds.size.y) / 2;

        transform.position = Vector3.Lerp(transform.position, center, smoothFactor);
        cam.orthographicSize = Mathf.Lerp(cam.orthographicSize, cameraSize, smoothFactor);
    }

    void Awake()
    {
        cam = GetComponent<Camera>();
    }
}
```

Parte 3 - Inserção do script do *PlayerController*

1. De forma similar faremos o mesmo que anterior só que para o script *PlayerController.cs* a seguir ao elemento **PlayerControl**. Este elemento será controlado pelas setas do teclado.

```
using UnityEngine;

[RequireComponent(typeof(Rigidbody2D))]
public class PlayerController : MonoBehaviour
{
    [Header("Settings")]
    [SerializeField] float maxVelocity = 6f;

    Rigidbody2D physics = null;

    void Update()
    {
        Vector3 normalizedInputDirection = new Vector3
        (
            x: Input.GetAxis("Horizontal"),
            y: Input.GetAxis("Vertical"),
            z: 0.0f
        ).normalized;

        physics.velocity = normalizedInputDirection * maxVelocity;
    }

    void Awake()
    {
        physics = GetComponent<Rigidbody2D>();
    }
}
```

2. As reações dos personagens **NPC1** e **NPC2** compreendem na movimentação dos distanciamentos e aproximações do **PlayerControl**.

Parte 4 - Inserção do script *SteeringActor*

1. E por último dotaremos os personagens de comportamentos de direção. Primeiro associaremos o script *SteeringActor.cs* ao personagem **NPC1**, e setaremos no painel INSPECTOR o atributo *target* ao elemento **PlayerControll** o comportamento SEEK. E ao personagem **NPC2** setaremos no painel INSPECTOR o atributo *target* ao elemento **PlayerControl** o comportamento EVADE.

```
using UnityEngine.UI;
using UnityEngine;
enum Behavior { Idle, Seek, Evade }
enum State { Idle, Arrive, Seek, Evade }

[RequireComponent(typeof(Rigidbody2D))]
public class SteeringActor : MonoBehaviour{
    [Header("Settings")]
    [SerializeField] Behavior behavior = Behavior.Seek;
    [SerializeField] Transform target = null;
    [SerializeField] float maxVelocity = 4f;
    [SerializeField, Range(0.1f, 0.99f)] float decelerationFactor = 0.75f;
    [SerializeField] float arriveRadius = 1.2f;
    [SerializeField] float stopRadius = 0.5f;
    [SerializeField] float evadeRadius = 5f;

    Text behaviorDisplay = null;
    Rigidbody2D physics;
    State state = State.Idle;

    void FixedUpdate(){
        if (target != null){
            switch (behavior){
                case Behavior.Idle: IdleBehavior();   break;
                case Behavior.Seek: SeekBehavior();   break;
                case Behavior.Evade: EvadeBehavior(); break;
            }
        }
        behaviorDisplay.text = state.ToString().ToUpper();
    }

    void IdleBehavior(){
        physics.velocity = physics.velocity * decelerationFactor;
    }

    void SeekBehavior(){
        Vector3 delta = target.position - transform.position;
        float distance = delta.magnitude;
        Vector3 normalizedDirection = delta.normalized;

        if (distance < stopRadius)
            state = State.Idle;
        else if (distance < arriveRadius)
            state = State.Arrive;
        else
            state = State.Seek;

        switch (state){
        case State.Idle: IdleBehavior(); break;
```

```
        case State.Arrive:
          var arriveFactor = 0.01f + (distance - stopRadius) / (arriveRadius - stopRadius);
          physics.velocity = arriveFactor * normalizedDirection * maxVelocity;
          break;
        case State.Seek:
          physics.velocity = normalizedDirection * maxVelocity;
          break;
        }
    }

    void EvadeBehavior(){
        Vector3 delta = target.position - transform.position;
        float distance = delta.magnitude;
        Vector3 normalizedDirection = delta.normalized;

        if (distance > evadeRadius)
            state = State.Idle;
        else
            state = State.Evade;

        switch (state){
            case State.Idle: IdleBehavior(); break;
            case State.Evade: physics.velocity = -normalizedDirection * maxVelocity; break;
        }
    }

    void Awake(){
        physics = GetComponent<Rigidbody2D>();
        behaviorDisplay = GetComponentInChildren<Text>();
    }

    void OnDrawGizmos(){
        if (target == null)
            return;

        switch (behavior){
            case Behavior.Idle:
                break;
            case Behavior.Seek:
                Gizmos.color = Color.white;
                Gizmos.DrawWireSphere(transform.position, arriveRadius);
                Gizmos.color = Color.green;
                Gizmos.DrawWireSphere(transform.position, stopRadius);
                break;
            case Behavior.Evade:
                Gizmos.color = Color.red;
                Gizmos.DrawWireSphere(transform.position, evadeRadius);
                break;
        }
        Gizmos.color = Color.gray;
        Gizmos.DrawLine(transform.position, target.position);
    }
}
```

2. Algumas considerações desta associação. Primeiro o *target* associado ao elemento **PlayerControl** significa que qual o alvo que reagirá ao comportamento.

3. O cálculo das reações dos personagens nada mais é do que ação e reação entre objetos, utilizando elementos como velocidade e deslocamento no domínio do tempo.

Parte 5 - Proposição de Novos Comportamentos de Direção

1. A proposta da continuação desta prática é a partir do modelo construído, aplica-lo a um personagem qualquer em uma cena de um Jogo Digital.
2. As direcionalidades e os 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.