Universidade Federal da Bahia
Curso: Engenharia de Computação
Disciplina: Programação Orientada a Objetos – MATA55
Professor: Rodrigo Rocha

GAME

NANO ATTACK

por Bianca Fernandes Silva

Tema: Nanomedicina.

Gênero: 2D, Side-Scroller.

Plataforma: PC Windows e Web.

Universo ou Contexto: Uma biomédica descobre que está infectada por um vírus mortal. Ao saber que tem pouco tempo de vida cria um nanorobô para atacar os vírus maliciosos e garantir sua sobrevivência.

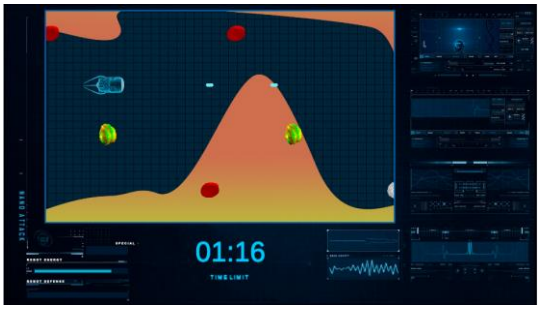
Objetivo do jogo: Controlar um nanorobô para atacar os vírus que estão no corpo de uma biomédica antes que o tempo acabe, além de desviar de qualquer obstáculo no caminho. Ganha o jogo quando derrota o principal vírus do jogo. Perde quando o tempo acaba ou quando perde todas as energias (life) ao se esbarrar nos obstáculos.

Telas:

Menu Principal: Play, Opções e Sair

Menu Opções: Volume, Controles e Créditos

Level:



Fase única:

Ação 1: Chegar até o foco dos vírus sem esbarrar nos obstáculos do caminho.

Ação 2: Atacar o máximo dos mini_vírus que conseguir.

Ação 3: Atacar o boss_vírus que é o principal disseminador da doença mortal.

Motor de jogo: Unity, versão 2021.1.12f1.

Linguagem de programação: C# (C Sharp).

Ambiente de desenvolvimento dos scripts: Microsoft Visual Studio 2019 community edition.

Scripts:

Todos os scripts devem herdar explicitamente da classe denominada **MonoBehaviour** que é a classe base do Unity. É através dela que podemos ter acesso a diversas funcionalidades, métodos e propriedades dos componentes da interface do Unity e configurar os elementos do jogo.

Por padrão os templates para os scripts já vem com o Monobehaviour e com os métodos mais comuns/usados nos projetos como:

Start() que executa o bloco somente uma vez ao iniciar a aplicação;

Update() que executa o bloco durante todo o tempo de execução atualizando a cada frame;

Abaixo estão as descrições das classes implementadas:

Classe HudControl: controla o carregamento de cenas e a possibilidade de sair do jogo.

Carregamento de cena: é feito através do método criado LoadScenes que usa o nome da cena como parâmetro. Exemplo: menus e level. Mas pose-se usar informações numéricas das cenas também.

```
public void LoadScenes(string name)
{
   SceneManager.LoadScene(name);
}
```

SceneManager: classe que é implementada pelo módulo core do Unity e que permite o gerenciamento de cena em tempo de execução.

LoadScene: método estático que carrega a cena por seu nome ou índice conforme a configuração feita no Unity.

Para sair do jogo: é possível através do método criado QuitGame.

```
public void QuitGame()
{
   Application.Quit();
}
```

Application: classe que permite acesso aos dados do jogo em tempo de execução.

Quit: método estático da classe Aplication que permite sair da aplicação (jogo).

Classe MusicVolume: controla a função do botão slider de volume presente no menu opções.

```
private AudioSource sound;
```

AudioSource: é uma classe implementada pelo módulo de áudio do Unity que permite adicionar aúdios ao jogo por meio de objetos e também controlar os áudios pelos métodos públicos de Play, Stop, entre outros.

```
public void UpdateVolume(float vol)
{
        musicVol = vol;
}
```

UpdateVolume: método criado para atualizar o volume do áudio ambiente colocado no menu do jogo. A variável está vinculada a propriedade volume que está sendo atualizada a cada quadro por estar dentro do método Update.

```
void Update()
{
    sound.volume = musicVol;
}
```

Classe SideScroller: controla a velocidade de movimentação da cena do jogo através de um rolamento lateral para a esquerda do objeto denominado mapa em direção ao jogador.

Awake() : é um método para carregar componentes e definir propriedades antes do método que são executados antes de iniciar o jogo. Esse método é executado antes do método Start().

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

FixedUpdate(): método que respeita os cálculo físicos dos componentes independente da taxa de quadros, pois executa num intervalo de tempo diferente na maioria das vezes do método Update. Importante pois considerando que

podemos definir por exemplo uma força para um objeto que se manterá fixa durante todo o jogo. Neste caso é usada para atualizar a velocidade de movimentação do objeto mapa do jogo.

```
private void FixedUpdate()
{
        sceneRb.velocity = new Vector2(-sceneSpeed * Time.deltaTime, 0);
}

public static float SetSceneSpeed(float val)
{
    sceneSpeed = val;
    return sceneSpeed;
}
```

SetSceneSpeed(): é um método estático que retorna um valor para definir a velocidade de movimentação do mapa que compõe a cena do jogo.

Classe **Timer:** controla o tempo necessário para vencer e o exibe na tela de gameplay.

StartTimer(): método para iniciar o contador;

```
private void StartTimer()
{
    timerActive = true;
}
```

StopTimer(): método para parar o contador;

```
public static void StopTimer()
{
    timerActive = false;
    PlayerHealth.loadNextScene = true;
}
```

TimeSpan(): é uma estrutura do System usado para representar um intervalo de tempo.

```
TimeSpan time = TimeSpan.FromSeconds(currentTime);
textCurrentTime.text=time.Minutes.ToString("00")+":"+time.Seconds.ToString("00");
```

Classe **PlayerHealth:** controla os níveis de energia da nave do jogador.

DamageToPlayer(): decrementa a quantidade que representa o nível de energia do player e ativa o método de explosão caso não haja mais energia.

```
private void DamageToPlayer()
{
     energy -= 1;
     if (energy <= 0)
     {
          ExplosionPlayer();
     }
}
```

OnCollisionEnter2D(): usado para detecção de colisão e barramento da passagem de objetos. Usado para ativar métodos de dano e morte do jogador.

tag: permite definir palavras que representam ou ativam alguma ação, elementos e propriedades dos objetos dos jogos. Muito utilizada em gatilhos como colisores.

```
private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.tag == "Obstacle")
    {
            DamageToPlayer();
     }

     if (collision.gameObject.tag == "Explosion")
     {
            ExplosionPlayer();
     }
 }
```

Classe **PlayerMovement:** controla a movimentação da nave do jogador e ativa e desativa as ações para acelerar a velocidade do mapa e da geração de inimigos.

Input.GetAxisRaw: permite indicar os eixos padrões que já estão pré-configurados no unity para o controle de entrada, que vincula Horizontal às teclas A,D, Left e Right e Vertical às teclas W,S, Up e Down.

```
private void FixedUpdate()
{
    float h = Input.GetAxisRaw("Horizontal");
    float v = Input.GetAxisRaw("Vertical");

    Move(h,v);
}

private void Move(float h, float v)
{
     movement.Set(h, v);
     playerRb.velocity = playerSpeed * Time.deltaTime * movement;
}
```

OnCollisionExite2D(): indica que não está havendo colisão entre os objetos especificados: o objeto que possue esse método com o referenciado através da tag "Acceleration" que corresponde a um objeto de cena de jogo.

```
private void OnCollisionExit2D(Collision2D collision)
{
    if (collision.gameObject.tag == "Acceleration" && (regionBoss == false))
    {
         SideScroller.SetSceneSpeed(200f);
         Enemy.SetEnemySpeed(400f);
    }
}
```

Classe **PlayerAttack:** controla o ataque do jogador.

Attack(): método criado para instanciar objetos de ataque numa posição definida previamente por meio de outro objeto de jogo.

Instantiate(): usado para criar novos objetos em tempo de execução.

```
private void Attack()
{
```

```
    if (laserCoolDown <= 0)
    {
        laserCoolDown = laserRate;
        var laserTransform = Instantiate(laser) as Transform;
        laserTransform.position = originLaser.transform.position;
        sound.Play();
    }
}
```

Classe **Laser:** define as características do abjeto de dano usado no ataque aos inimigos do jogo.

Classe **Enemy:** define a movimentação e velocidade dos inimigos e a região em que poderão sofrer danos.

OnTriggerEnter2D(): usado para detecção de colisão e sem o barramento da passagem de objetos. Usado para ativar o método de destruição do objeto inimigo ao sair da cena de jogo.

```
private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.gameObject.tag == "Explosion")
    {
         Destroy(gameObject, 0.03f);
     }
}
```

Classe **SpawnEnemy:** controla o gerador de inimigos.

SerializeField: usado para manter visível e editável variáveis no inspector do Unity para que possam ser alterados manualmente.

InvokeRepeating: permite invocar um método repetidamente levando em conta intervalos de tempo definidos.

```
[SerializeField]
private GameObject[] enemies;

private void Start()
{
    InvokeRepeating("SpawnRandom", spawnTime, spawnDelay);
}

void SpawnRandom()
{
    random = Random.Range(0, enemies.Length);
    Instantiate(enemies[random], transform.position, transform.rotation);
}
```

Classe **EnemyHealth:** define a quantidade de energia dos inimigos e ativa o método que indica que o jogador venceu.

```
public void Damage(int damageCount)
{
    enemyHealth -= damageCount;

    if (enemyHealth <= 0 && (canBeDestroyed == true))
    {
        Destroy(gameObject);
          if (isBoss == true)
          {
                GameWin();   }   }  }
```

Links vinculados ao projeto:

Repositório Github: https://github.com/bifernandes/GAMENANOATTACK

Executável web: https://bifernandes.github.io/GAMENANOATTACK/

Demonstração: https://www.youtube.com/watch?v=EKuwm9azJs0

Referência:

https://docs.unity3d.com/2021.1/Documentation/Manual/index.html