LICENCIATURA EM

DESENVOLVIMENTO DE JOGOS DIGITAIS

Relatório – Zelda Link to the Past

Desenvolvimento de Jogos 2D

João Pedro Cavacos, Nº 36430

PORTO, MÊS DE ANO.

ÍNDICE GERAL

Introdução

Este projeto foi realizado no âmbito do Exame de Época Normal da cadeira de Desenvolvimento de Jogos 2D com o objetivo de recriar um jogo antigo (as suas mecânicas).

Neste caso, escolhi o jogo Zelda Link to the Past, originalmente lançado na SNES, em 1991.

Neste jogo, jogamos como o personagem Link numa aventura para salvar a princesa Zelda e o reino de Hyrule de um mago maléfico. O jogador vai percorrer pelo reino de Hyrule aventurando-se em dungeons, usando items mágicos, a espada e o arco.

As mecânicas propostas fazer foram: controlo do jogador (Link), como também o seu combate com a espada e o arco; o controlo do inimigo, incluindo o sistema de patrulha e ataque do próprio; um sistema de loot das folhagens; e por fim, um sistema de currency com gemas.

Mecânicas

Link:

- Movimento (topdown);
- Sistema de vida;
- Ataque básico da espada com knockback;
- Disparar setas de um arco;

Inimigo:

- Movimento;
- Ataque básico de espada com knockback;
- Sistema patrulha de um ponto a outro;
- Raio de perseguição e ataque;

Loot das folhagens:

- Ao destruirmos uma folhagem temos a chance de calhar um dos seguintes items: gemas, vida ou simplesmente nada.

Currency de Gemas:

- Quando apanhamos uma gema o nosso counter no HUD sobe por um valor.

As mecânicas desconstruídas

Movimento topdown

```
Event function
void FixedUpdate()
{
    if(Input.GetKeyDown(KeyCode.E) && playerState != PlayerState.ATTACK && playerState != PlayerState.STAGGER){
        StartCoroutine( routine: AttackCoroutine());
    }
    else if(Input.GetKeyDown(KeyCode.R) && playerState != PlayerState.ATTACK && playerState != PlayerState.STAGGER)
    {
        StartCoroutine( routine: ArrowCoroutine());
    }
    else if(playerState == PlayerState.IDLE || playerState == PlayerState.MOVE)
    {
        UpdateMove();
    }
}

Frequently called  1 usage
private void Move(){
    rb.MovePosition(rb.position + movement.normalized * speed * Time.fixedDeltaTime);
}

Frequently called  1 usage
private void UpdateMove(){
    if(movement != Vector2.zero){
        Move();

        animator.SetFloat( name: "Horizontal", movement.x);
        animator.SetFloat( name: "Vertical", movement.y);
        animator.SetBool( name: "Moving",  value: true);
    }
    else
    {
        animator.SetBool( name: "Moving",  value: false);
    }
}
```

Para o movimento, foi bastante simples a implementação. Foi utilizado um rigidbody para fazer o player andar com o MovePosition() e atualizar as animações da Blend Tree com o UpdateMove(). Inicializar no FixedUpdate() para física, como este movimento.

Sistema de vida

Com este sistema de vida decidi implementar Scriptable Objects para me facilitar na maneira em como o UI é atualizado. Com isto, optei por usar um observer pattern que envia um sinal do player (que vai ter a sua vida) que atualiza a UI de acordo com os corações que deve apresentar.

```
[CreateAssetMenu]
3 asset usages  6 usages  6 exposing APIs
public class FloatValue : ScriptableObject, ISerializationCallbackReceiver
{

    public float initialValue;  Changed in 3 assets
    [NonSerialized] public float runtimeValue;
    public void OnBeforeSerialize(){

    }
    public void OnAfterDeserialize(){
        runtimeValue = initialValue;
    }
```

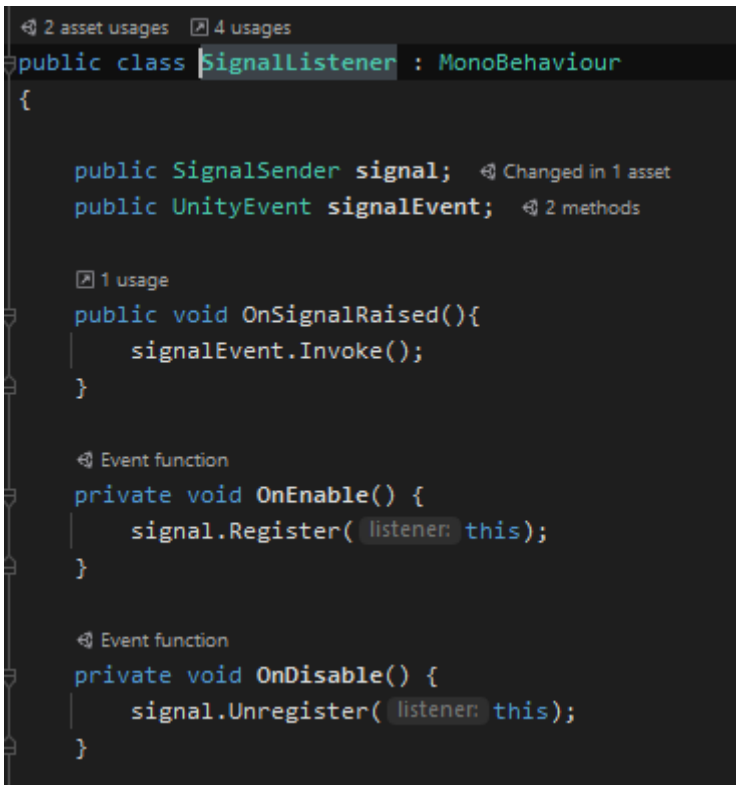
```
[CreateAssetMenu]
2 asset usages  3 usages  3 exposing APIs
public class SignalSender : ScriptableObject
{
    //Signal to send, observer pattern - player sends signal to hearts
    public List<SignalListener> listeners = new List<SignalListener>();

    4 usages
    public void Raise(){

        for (int i = listeners.Count - 1 ; i >= 0; i--)
        {
            listeners[i].OnSignalRaised();
        }
    }

    1 usage
    public void Register(SignalListener listener){
        listeners.Add(listener);
    }

    1 usage
    public void Unregister(SignalListener listener){
        listeners.Remove(listener);
    }
```



```
2 asset usages  4 usages
public class SignalListener : MonoBehaviour
{

    public SignalSender signal;  Changed in 1 asset
    public UnityEvent signalEvent;  2 methods

    1 usage
    public void OnSignalRaised(){
        signalEvent.Invoke();
    }

    Event function
    private void OnEnable() {
        signal.Register( listener: this);
    }

    Event function
    private void OnDisable() {
        signal.Unregister( listener: this);
    }
```

O script FloatValue foi criado para alterar os valores que precisamos de mudar para criar o sistema de vida. O initialValue é o valor máximo da nossa vida (ou outra mecânica, já que é reutilizável) e o runtimeValue é a vida durante o play mode.

Depois temos os dois scripts que comunicam entre eles, sendo o SignalSender o que cria o sinal e envia-o e, o SignalListener recebe e aplica.

```
public class HeartsManager : MonoBehaviour
{

    public Image[] hearts;  Serializable
    public Sprite fullHeart;  Serializable
    public Sprite halfHeart;  Serializable
    public Sprite emptyHeart;  Serializable

    public FloatValue heartsContainer;  HeartsContainer.asset
    public FloatValue playerCurrentHealth;  PlayerHealth.asset

    Event function
    void Start()
    {
        CreateHearts();
    }

    1 usage
    public void CreateHearts(){
        for (int i = 0; i < heartsContainer.initialValue; i++)
        {
            hearts[i].gameObject.SetActive(true);
            hearts[i].sprite = fullHeart;
        }
    }

    1 asset usage
    public void UpdateHearts(){
        float health = playerCurrentHealth.runtimeValue / 2; //divided by 2 because of half hearts

        for (int i = 0; i < heartsContainer.initialValue; i++)
        {
            if(i <= health - 1){
                //Full heart
                hearts[i].sprite = fullHeart;
            }
            else if(i >= health)
            {
                //Empty heart
                hearts[i].sprite = emptyHeart;
            }
            else
            {
                //Half heart
                hearts[i].sprite = halfHeart;
            }
        }
```

Neste script básico, recebemos a informação do valor da vida do sinal e baseado nisso atualizo os corações no HUD dependendo de quantos tiverem ativos.

Ataque básico com espada

```
1 usage
public void Attack(float knockbackTime, float damage){

    currentHealth.runtimeValue -= damage;
    playerHealthSignal.Raise();

    if(currentHealth.runtimeValue > 0){
        StartCoroutine( routine: KnockbackCoroutine(knockbackTime));
    }
    else
    {
        //Game over
        Debug.Log( message: "Dead");
    }
}
Frequently called  1 usage
private IEnumerator AttackCoroutine(){

    animator.SetBool( name: "Attacking",  value: true);
    playerState = PlayerState.ATTACK;
    enemyHit.Play();
    yield return new WaitForEndOfFrame();
    animator.SetBool( name: "Attacking",  value: false);
    yield return new WaitForSeconds(0.5f);
    playerState = PlayerState.MOVE;
}

Frequently called  1 usage
private IEnumerator KnockbackCoroutine(float knockbackTime){

    if(rb != null){

        yield return new WaitForSeconds(knockbackTime);
        rb.velocity = Vector2.zero;
        playerState = PlayerState.IDLE;
        rb.velocity = Vector2.zero;
    }
}
```

Para ajudar na mudança de estados, criei uma State Machine que guarda todos os estados do player e do inimigo, separadamente. Então, no script de ataque mudo de estado dependente da ação que fizer. No caso mais concreto, o ataque, é uma junção da Coroutine de knockback que aplica um força sobre o inimigo e o envia para longe e a de Attack que dá dano e aplica a animação correta dependendo do eixo. Depois é inicializado no Update() de acordo com o input do player.

```
[Header ("Attack stats")]
public float thrustForce;  Changed in 3 assets
public float knockbackTime;  "0.2"
public float damage;  "1"
 Event function
private void Start() {
}

 Event function
private void OnTriggerEnter2D(Collider2D other) {

    if(other.gameObject.CompareTag("Enemy") || other.gameObject.CompareTag("Player")){
        Rigidbody2D rb = other.GetComponent<Rigidbody2D>();

        if(rb != null){

            Vector2 forceDirection = rb.transform.position - transform.position;
            Vector2 force = forceDirection.normalized * thrustForce;

            rb.AddForce(force, ForceMode2D.Impulse);

            if(other.gameObject.CompareTag("Enemy") && other.isTrigger)
            {
                rb.GetComponent<Enemy>().enemyState = EnemyState.STAGGER;
                other.GetComponent<Enemy>().Attack(rb, knockbackTime, damage);
            }

            if(other.gameObject.CompareTag("Player") && other.isTrigger){

                if(other.GetComponent<Link>().playerState != PlayerState.STAGGER)
                {
                    rb.GetComponent<Link>().playerState = PlayerState.STAGGER;
                    other.GetComponent<Link>().Attack(knockbackTime, damage);
                }

            }

        }
    }
}
```

Figura 1 - Attack system script

Ataque de arco e flechas

Para criar uma fleche tive de criar um prefab da própria e instanciá-la quando o player clicasse numa tecla. Para fazer com que a seta estivesse com a direção e rotação correta, usei outra vez o rigidbody que vai buscar os valores do animador do player para determinar a direção e o cálculo da tangente, transformando em graus para descobrir a rotação e aplicá-la.

```
No asset usages  2 usages
public class Arrow : MonoBehaviour
{

    public float speed;  Unchanged
    public Rigidbody2D rb;  Unchanged

    Frequently called  1 usage
    public void SetupArrow(Vector2 velocity, Vector3 direction){

        rb.velocity = velocity.normalized * speed;
        transform.rotation = Quaternion.Euler(direction);
    }

    Event function
    private void OnTriggerEnter2D(Collider2D other) {

        if(other.gameObject.CompareTag("Enemy")){
            Destroy(this.gameObject);
        }

    }
}
```

```
Frequently called  1 usage
private IEnumerator ArrowCoroutine(){

    playerState = PlayerState.ATTACK;
    arrowShot.Play();
    yield return new WaitForEndOfFrame();
    MakeArrow();
    yield return new WaitForSeconds(0.5f);
    playerState = PlayerState.MOVE;
}

Frequently called  1 usage
private void MakeArrow(){

    Vector2 direction = new Vector2( x: animator.GetFloat( name: "Horizontal"),  y: animator.GetFloat( name: "Vertical"));
    Arrow arrowGO = Instantiate(arrow, transform.position, Quaternion.identity).GetComponent<Arrow>();

    arrowGO.SetupArrow(direction,  direction: ArrowRotation());
}

Frequently called  1 usage
Vector3 ArrowRotation(){
    float direction = Mathf.Atan2( y: animator.GetFloat( name: "Horizontal"),  x: animator.GetFloat( name: "Vertical")) * Mathf.Rad2Deg;

    return new Vector3( x: 0,  y: 0,  z: direction);
}
```

Movimento do Inimigo

Para o movimento do inimigo, simplesmente criei um raio de perseguição e um raio de ataque que faz com que o inimigo reaja quando um player entra. Depois com o MoveTowards o inimigo vai contra o player e aplica dano e um knockback, reutilizando o knockback do jogador só que, agora no mesmo.

Sistema de ataque e patrulha dos inimigos

Como tenho dois inimigos diferentes presentes no jogo, fiz os dois criarem uma relação de herança com o próprio script de Enemy para poder criar mais inimigos mais facilmente apenas criando scripts dos mesmos para alterações de mecânica que queira adicionar.

O inimigo verde fica simplesmente parado num sítio até que o jogador aproxime dele ativando a perseguição e o ataque.

O azul patrulha uma zona de ponto a ponto até o jogador aproximar, e quando fica fora de raio volta ao mesmo sítio de patrulha.

```
//Check the distance of the player to the enemy to trigger the MoveTowards
Frequently called  1 usage  1 override
public virtual void CheckDistance(){ //can override with virtual

    if(Vector3.Distance( a: playerPos.position,  b: transform.position) <= chaseRadius && Vector3.Distance( a: playerPos.position,  b: transform.position) > attackRadius){

        if(enemyState == EnemyState.IDLE || enemyState == EnemyState.MOVE && enemyState != EnemyState.STAGGER){
            Vector3 moveTowards = Vector3.MoveTowards( current: transform.position,  target: playerPos.position,  maxDistanceDelta: speed * Time.fixedDeltaTime);

            ChangeAnimation( direction: moveTowards - transform.position);
            rb.MovePosition(moveTowards);
            ChangeState(EnemyState.MOVE);
            animator.SetBool( name: "inRange",  value: true);
        }
    }
    else if(Vector3.Distance( a: playerPos.position,  b: transform.position) > chaseRadius)
    {
        animator.SetBool( name: "inRange",  value: false);
    }
}

//Change animation based on direction
Frequently called  3 usages
public void ChangeAnimation(Vector2 direction){
    direction = direction.normalized;
    animator.SetFloat( name: "Horizontal", direction.x);
    animator.SetFloat( name: "Vertical", direction.y);
}

Frequently called  1 usage
void ChangeState(EnemyState state){
    if(enemyState != state){
        enemyState = state;
    }
}
```

```
Frequently called  0+1 usages
public override void CheckDistance(){

    //Check if is in range of player
    if(Vector3.Distance( a: playerPos.position,  b: transform.position) <= chaseRadius && Vector3.Distance( a: playerPos.position,  b: transform.position) > attackRadius){

        if(enemyState == EnemyState.IDLE || enemyState == EnemyState.MOVE && enemyState != EnemyState.STAGGER){
            Vector3 moveTowards = Vector3.MoveTowards( current: transform.position,  target: playerPos.position,  maxDistanceDelta: speed * Time.fixedDeltaTime);

            ChangeAnimation( direction: moveTowards - transform.position);
            rb.MovePosition(moveTowards);
            animator.SetBool( name: "inRange",  value: true);
        }
    }
    else if(Vector3.Distance( a: playerPos.position,  b: transform.position) > chaseRadius) // if not, continues patrol
    {
        if(Vector3.Distance( a: transform.position,  b: path[currentPoint].position) > roundedDistance){ //Check which point is closer, move there
            Vector3 moveTowards = Vector3.MoveTowards( current: transform.position,  target: path[currentPoint].position,  maxDistanceDelta: speed * Time.fixedDeltaTime);

            ChangeAnimation( direction: moveTowards - transform.position);
            rb.MovePosition(moveTowards);
        }
        else //When reached goal change to other point
        {
            ChangePoint();
        }

    }
}

Frequently called  1 usage
private void ChangePoint(){ //Check the point to follow

    if(currentPoint == path.Length - 1){
        currentPoint = 0;
        nextPoint = path[0];
    }
    else
    {
        currentPoint++;
        nextPoint = path[currentPoint];
    }

}
```

Sistema de loot

Para este sistema optei de novo por um Scriptable Object que é responsável por definir a chance e o loot que pode dropar ao derrotar um inimigo ou destruir uma folhagem.

De momento, o loot disponível para drop é um coração que cura a vida do jogador por 1 coração ao apanhar e uma gema que serve de "moeda" do jogo em si.

Para a folhagem, apenas tenho um script que faz mudar o sprite e desativar o rigidbody para uma folhagem cortada, que pode dropar com 10% de chance uma gema, 15% uma vida e o restante nada.

```
1 asset usage   2 usages
public class Leaf : MonoBehaviour
{
    public Sprite destroyedSprite;  Serializable
    private SpriteRenderer spriteRenderer;
    private BoxCollider2D boxCollider2D;
    public LootTable thisLoot;  Changed in 1+ assets

    [Header ("Sounds")]
    public AudioSource grassSliced;  Changed in 0+ assets

    Event function
    void Start() {
        spriteRenderer = GetComponent<SpriteRenderer>();
        boxCollider2D = GetComponent<BoxCollider2D>();
    }

    //Replaces the sprite to destroyed and disables collider
    1 usage
    public void DestroyLeaf(){
        spriteRenderer.sprite = destroyedSprite;
        boxCollider2D.enabled = false;
        grassSliced.Play();
        MakeLoot();
    }

    1 usage
    public void MakeLoot(){
        if(thisLoot != null){
            Collectibles current = thisLoot.LootCollectible();

            if(current != null){
                Instantiate(current.gameObject, transform.position, Quaternion.identity);
            }
        }
    }
}
```

```
public class Destroyable : MonoBehaviour
{
    Event function
    private void OnTriggerEnter2D(Collider2D other) {

        if(other.CompareTag("Breakable")){

            //For the leaf
            if(other.GetComponent<Leaf>() != null){
                other.GetComponent<Leaf>().DestroyLeaf();
            }
        }
    }
```

Para os inimigos o loot teria que ser melhorado pois é algo que é dificil de derrotar, portanto com o mesmo scriptable object aumentei as chances de calhar algo melhor – 15% para vida e 25% para gemas.

```
public class Collectibles : MonoBehaviour
{
    public SignalSender collectibleSignal;  Changed in 0+ assets
}
```

Tenho também o script de collectibles que serve para qualquer tipo de drop, que envia um sinal para registar no UI, como a vida anteriormente.

```
[System.Serializable]
1 usage
public class Loot{
    public Collectibles loot;  Serializable
    public int lootChance;  Serializable
}

[CreateAssetMenu]
2 asset usages  2 usages  2 exposing APIs
public class LootTable : ScriptableObject
{
    public Loot[] loots;  Serializable

    2 usages
    public Collectibles LootCollectible(){

        int probability = 0;
        int currentProbability = Random.Range(0,100);

        for (int i = 0; i < loots.Length; i++)
        {
            probability += loots[i].lootChance;
            if(currentProbability <= probability){
                return loots[i].loot; //Drop loot
            }
        }

        return null; //Drop nothing
    }


}
```