2024

Křesťanské gymnázium

Kozinova 1000

102 00 Praha 10 – Hostivař

Pixel Carnage

KRYŠTOF KYSLÍK

Prohlášení

Prohlašuji, že jsem maturitní práci zpracoval samostatně a že jsem uvedl všechny použité informační zdroje a literaturu v seznamu použitých zdrojů.

Poděkování

Rád bych poděkoval vedoucí našich maturitních prací Mgr. Ivoně Spurné za její cenné rady, podporu a nápomocnost.

Dále bych rád poděkoval všem lidem, kteří mi pomohli s testováním a dali mi zpětnou vazbu.

Abstrakt

Tento maturitní projekt se zabývá průběhem vývoje, tvorby a funkcí mé 2D Hack and Slash videohry. Videohra pojmenovaná „Pixel Carnage“ je užitečná jako inspirace ostatním studentům. Poskytuje možnost strávení volného času hraním. Jde i o formu inspirace studentům a integrování učení prostřednictvím tvorby hry.

Klíčová slova

Videohra, Skript, 2D, Hack and Slash, Nepřítel, Collider, C#

Abstract

This graduation project explores the development, creation, and features of my 2D Hack and Slash video game. The video game, named "Pixel Carnage", is useful as an inspiration to other students. It provides an opportunity to spend free time playing. It is also a form of inspiring students and integrating learning through game creation.

Key words

Videogame, Script, 2D, Hack and Slash, Enemy, Collider, C#

Obsah

Souhrny objektů

Seznam obrázků

Úvod

Cílem této práce je vytvoření funkční Hack and Slash hry v enginu Unity, která bude vizuálně originální a nabídne odpovídající herní zážitek. Mezi mé cíle patří naučit se pracovat s programovacím jazykem C# a pochytit nové funkce v Unity. V práci se také zaměřím na technické aspekty vývoje a na programovací jazyk C#. Přínos této práce spočívá ve vytvoření hry, která může být následně prezentována, sdílena a použita jako inspirace pro budoucí projekty, a zároveň naučit se a zlepšit v programování v jazyce C# a pracovat v prostředí Unity.

Teoretická část

C# (C Sharp)

C Sharp, známý spíše jako C#, je programovací jazyk založený Microsoftem na základě jazyků C++ a Java. Tento programovací jazyk se běžně používá k tvorbě databázových programů, webových aplikací a stránek, aplikací ve Windows nebo právě her v Unity enginu. Jeho velkou výhodou je jednoduchost syntaxe. Společně s jednoduchostí Unity mohou vývojáři využívat C# k vytváření různých funkcí a systémů ve svých hrách, jako je ovládání postav, umělá inteligence, fyzikální simulace, animace, efekty a mnoho dalšího.

Používá objektově orientovaný přístup, kterým strukturuje kód pomocí tříd, objektů a dědičnosti. Objekt je instance třídy, což znamená konkrétní reprezentaci abstraktního návrhu definovaného ve třídě. Tyto objekty mohou obsahovat data (atributy) a metody (funkce), které jsou specifikovány ve třídě.

Pár důležitých charakteristik C#: neexistující vícenásobná dědičnost, žádné globální proměnné nebo metody a tzv. case sensitivita.

Neexistující vícenásobná dědičnost znamená, že každá třída má pouze jednu třídu jako rodiče.

Žádné globální proměnné nebo metody nás staví do situace, že v každé třídě je potřeba ty proměnné definovat nebo deklarovat individuálně. Lze ale použít systém zveřejnění pro používání proměnných či metod i mimo jejich definující třídu. Základní druhy proměnných jsou: int – celočíselný datový typ, float – ukládá jak pozitivní, tak negativní čísla i v desetinné formě, bool – dva stavy pravda / nepravda.

C# je case sensitive, což znamená, že rozlišuje mezi malými a velkými písmeny. Názvy se tudíž musí používat stále stejně, protože „Pojmenování“ a „pojmenování“ nejsou ekvivalentní výrazy.

Hack and Slash

Hack and slash, známý také jako hack and slay (H&S nebo HnS) nebo slash 'em up, označuje typ hry dávající důraz na boj zblízka zbraněmi. Mohou obsahovat i sekundární zbraně založené na projektilech (například pistole). Jedná se o podžánr beat 'em up her, který se zaměřuje na boj zblízka, obvykle s meči. Hack and slash hry z pohledu třetí osoby jsou někdy také známé jako akční hry s postavami a efektní bojovky.

Unity

Unity engine je multiplatformní herní engine od společnosti Unity Technologies. Poskytuje vývoj her v 2D a 3D libovolného žánru a zaměření. Podporuje jak grafickou tvorbu, tak psaní skriptů v jazyce C#. Unity má i poměrně velkou komunitu, vývojář budoucí hry má tudíž přístup k velkému množství dat skrz Unity Asset Store.

Pro vytvoření 2D hry existuje mnoho unikátních funkcí, které Unity nabízí pro jednoduchost a snadnost. I přes to, že jsme v 2D, rozlišuje hloubku obrazu, co se a s čím jak překrývá.

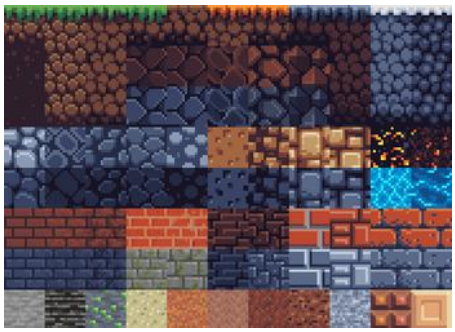
Unity 2D má také vlastní způsob na implementaci animací. Podporuje dva druhy. Sprite sheet animaci, která se skládá z obrázků (spritů) jdoucích po sobě, vytvářejících iluzi pohybu, nebo Skeletal animaci, kdy na obrázek dodá kostru, se kterou následně hýbeme.

Praktická část

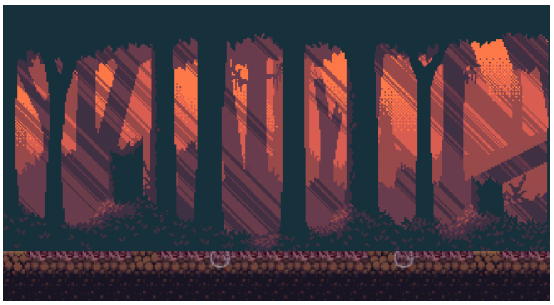
Mapa a Kamera

Aby měl hráč možnost kde a na čem hrát, bylo potřeba vytvořit mapu. Tu jsem vytvořil hlavně pomocí dvou objektů. Prvním bylo pozadí mapy, které jsem vložil ve formě spritu. Aby byla mapa omezená ohraničením, přidal jsem pomocí funkce Unity Tilemap. Ta využívá prostor rozdělený do dlaždic, do kterých lze vložit textury nebo obrázek a přidávat jim vlastnosti. Tu hlavní, kterou jsem využil, jsou kolize. Ty byly potřeba, aby mapa měla neprostupnou bariéru a hráč jednoduše nepropadl zemí.

Dále jsem připravil kameru. Ta zobrazuje to, co vidí hráč. Aby šla s hráčem a nevystoupila z mapy, musel jsem ji nejprve spojit s virtuální kamerou. Tu mám z Assetu Cinemachine z Unity Asset Storu. U ní jsem nastavil, aby následovala hráče a nemohla přesáhnout hranice mapy.



Img 2 Textury



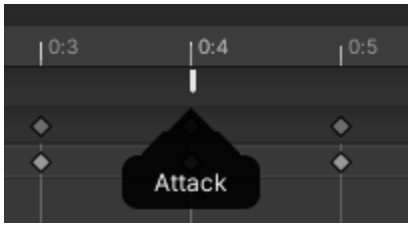
Img 1 Mapa

Postava hráče

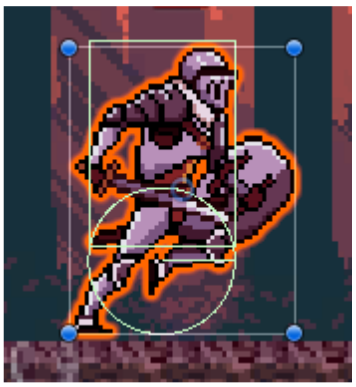
Po vytvoření mapy jsem se začal zabývat hráčem. Nejprve jsem zavedl nový game object a k tomu jsem přidal sprite s obrázkem. Dále jsem přidal potřebné komponenty, aby mohla být postava hráče ovládána, nepropadla mapou a detekovala kolize. To jsem zajistil přidáním Colliderů 2D. Právě ty kolize detekují. Dále jsem přidal Rigidbody 2D, který odpovídá za aplikování pohybu na objekt pomocí vektorů. Tím jsem zajistil základní funkce pro postavu. Poté zbývaly dvě věci: ovládání a animace. O skriptech budu mluvit podrobněji později. Pomocí několika skriptů lze ovládat postavu, útočit, získávat skóre, utrpět poškození a nakonec i zemřít. Aby nešlo jen o hýbající se statický obrázek, přidal jsem pro každou akci animace. Animace jsem vytvořil pomocí Sprite sheet. Je to klasický způsob animace, kdy nasázíme obrázky za sebou a vytvoříme pohyb. Obrázky jsem sice získal z internetu, ale bylo potřeba je ořezat a vycentrovat. Díky tomu vznikly poměrně kvalitní animace bez skoků. Pro spouštění animací pracujeme

s Animatorem, který funguje jako manažer. Pomocí podmínek určuje, kdy a jakou animaci spustit.

Img 4 Postava hráče



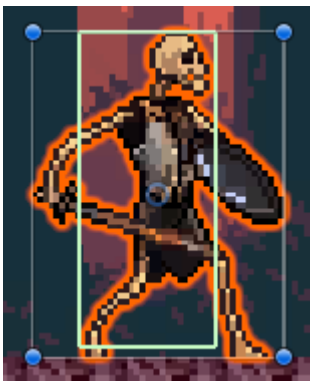
Img 3 Metoda v Animaci



Img 5 Collidery

Nepřátelé

První ze tří nepřátel byl Skeleton Warrior. Postup přidání byl podobný jako u postavy hráče. Začal jsem přidáním obrázku a k němu přiřadil další vlastnosti. Nepřátelé jsou ale narozdíl od hráče o něco jednodušší, nepočítaje skripta. Nepřítele totiž neovládá hráč, ale hra samotná. Takže stačí méně komponentů, zato vyžaduje více práce se skripty. Poté jsem přidal Animator a vytvořil animace.



Img 8 Collider

Img 7 Příklady komponenty

Img 6 Skeleton Warrior

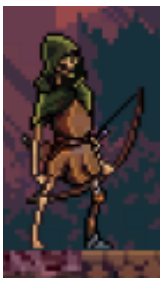
Druhý nepřítel byl Black Werewolf. Jeho nastavování se lišilo od Skeleton Warriora jen v pár hodnotách. Konkrétně jak je velký, rychlost pohybu a jak vysoké poškození způsobí. Zajímavější byl v pořadí třetí nepřítel.

Img 10 Poměr velikostí nepřátel

Img 9 Mrtvý Werewolf

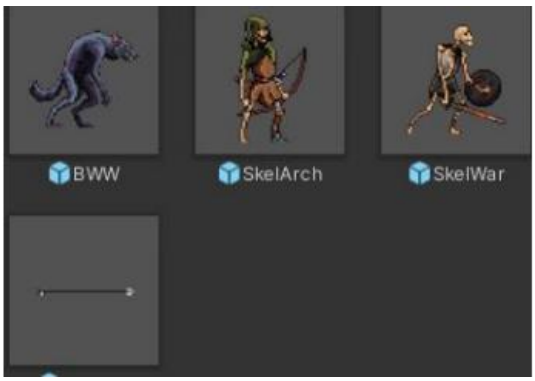
Skeleton Archer pro mě byl velký oříšek. Poprvé jsem pracoval se zbraní na dálku, a ještě k tomu s lukem. Měl jsem tušení, jak asi budu postupovat, ale nevěděl jsem, jak složité to bude. K běžnému postupu bylo potřeba zajistit detekci na větší vzdálenost a samotný objekt šípu, který pak bude z luku vystřelen. Abych mohl šíp vytvořit vícekrát se stejnou vlastností, využil jsem funkci Unity tzv. prefabu. Prefab je jakýkoliv GameObject, který zkopírujeme do Assetů (souborů) Unity. Tím se nám udělá kopie. Tu můžeme následně pomocí skriptů vkládat a klonovat do hry.

Img 11 Množství potřebných skriptů



Img 12 Skeleton Archer

Stejným způsobem jsem pak uložil všechny nepřátele jako Prefab. Chtěl jsem, aby se nepřátelé vytvářeli neustále, dokud hráč bude hrát. Pomocí skript jsem tedy vytvořil Spawnery, místa, kde jsem mohl vkládat nové klony nepřátel do hry.



Img 13 Prefaby

Skripta

Na začátku tohoto projektu jsem byl programováním a kódováním téměř nepolíbený. Měl jsem nějaké základy, proto jsem se rozhodl, že nepůjdu podle hotového tutoriálu na hru. Zvolil jsem metodu obecných tutoriálů, abych pochopil, proč to tak je. Většina kódu je tedy moje tvorba.

Pohyb hráče

Pohyb hráče mám zajištěný pomocí dvou skriptů. Character Controller 2D, který není můj, ale

velmi mi zjednodušil práci s postavou hráče. A dále PlayerMovement, ten jsem již psal já. Pomocí skriptu PlayerMovement detekuji vstup od hráče. Pokud hráč zmáčkne danou klávesu, Chracter Controller 2D vypočítá hodnotu vektoru (směr a sílu pohybu) a přidá vektor na Rigidbody 2D hráče. Při ovládání postavy mohl hráč vykonat několik akcí, například: vyskočit, chodit vlevo nebo vpravo, blokovat nebo zaútočit. Některé šlo však spustit najednou, což mohlo rozbít ovládání. To se vyřešilo podmínkami. Způsob referencí není běžný, ale byl to první způsob, jaký jsem uměl a který fungoval. V průběhu času jsem začal používat lepší a jednodušší metody.

```
// Move the character by finding the target velocity
Vector3 targetVelocity = new Vector2(move * 10f, m_Rigidbody2D.velocity.y);
// And then smoothing it out and applying it to the character
m_Rigidbody2D.velocity = Vector3.SmoothDamp(m_Rigidbody2D.velocity, targetVelocity, ref m_Velocity, m_MovementSmoothing);

// If the input is moving the player right and the player is facing left...
if (move > 0 && !m_FacingRight)
{
    // ... flip the player.
    Flip();
}
// Otherwise if the input is moving the player left and the player is facing right...
else if (move < 0 && m_FacingRight)
{
    // ... flip the player.
    Flip();
}
```

Img 16 Část kódu Charcter Controlled 2D

```
if (Input.GetButtonDown("Jump"))
{
    jump = true;

    animator.SetBool("IsJumping", true);
}

if (animator.GetBool("IsJumping") == false)
{
    if (Input.GetButtonDown("Crouch"))
    {
        animator.SetBool("Block", true);
        crouch = true;
        Debug.Log("Player Blocking");
    }
    else if (Input.GetButtonUp("Crouch"))
    {
        animator.SetBool("Block", false);
        crouch = false;
        Debug.Log("Player stopped blocking");
    }
}
```

Img 15 Kód podmínek

```
public CharacterController2D controller;

public Animator animator;

public bool player = true;

float horizontalMove = 0f;

public float runSpeed = 40f;

bool jump = false;

bool crouch = false;
```

Img 14 Proměnné a reference komponentů

Životy hráče

Na životy používám skript PlayerHealth. Uvnitř tohoto skriptu mám tři hlavní funkce. První, na kterou se zaměřím, je ztráta životů. Ta je podmíněná detekcí přesahu dosahu útoku (attackRange) nepřítele a Collideru hráče. Při detekci se ubere hráči počet životů odpovídající hodnotě poškození útoku nepřítele. To se pak zobrazí i na ukazateli zdraví (health bar).

```
// zranění hráče
public void PlayerTakeDamage(int damage)
{
    if(animator.GetBool("Block") == true)
    {
        currentHealth -= (damage / 2);
        healthBar.SetHealth(currentHealth);
    } else
    {
        currentHealth -= damage;
        healthBar.SetHealth(currentHealth);
    }

    if (currentHealth <= 0)
    {
        healthBar.SetHealth(0);
        PlayerDie();
        score.GameOver();
    }
}
```

Img 19 Kód zranění hráče

```
void PlayerDie()
{
    playerIsDead = true;

    Debug.Log("Player Died");

    animator.SetBool("IsDead", true);

    //"Vypnutí" hráče

    bc2D.enabled = false;

    this.enabled = false;

    player.PlayerDied();

    rb.velocity = Vector3.zero;
}

void AnimatorDisable()
{
    animator.enabled = false;
}
```

Img 18 Kód smrti hráče

```
// Start is called before the first frame update
void Start()
{
    player = GetComponent<PlayerMoveScript>();
    currentHealth = startHealth;

    healthBar.SetMaxHealth(startHealth);
}
```

Img 17 Kód nastavení životů

S touto funkcí velmi úzce souvisí smrt hráče. Na začátku se životy hráče nastaví na určenou hodnotu. Celou dobu, co hráč ztrácí nebo získává životy, se tato hodnota upravuje. Pokud je hodnota rovna nule nebo menší, hráč je mrtvý. Je pak zapotřebí vypnout všechny komponenty, které by mohl hráč nebo hra stále používat.

Poslední funkcí je léčení hráče. Hráč se vyléčí, pokud způsobí poškození nepříteli. Protože je potřeba tuto funkci (metodu) provést při útoku, musí být nastavena na veřejnou, aby mohla být použitá při jiné funkci. Stejně tak je veřejná i funkce zranění, protože je používána při útoku nepřátel. K nastavením atributy veřejnosti přidáme před prvek „public“.

```
// léčení hráče
public void PlayerGetHealth(int healing)
{
    currentHealth += healing;

    healthBar.SetHealth(currentHealth);
}
```

Img 20 Kód léčení

```
public class PlayerHealth : MonoBehaviour
{
    public Animator animator;

    private PlayerMoveScript player;

    public Rigidbody2D rb;
    public BoxCollider2D bc2D;
    public CircleCollider2D cc2D;

    public HealthBarScript healthBar;
    public ScoreScript score;
    public int startHealth = 20;
    int currentHealth;

    public bool playerIsDead = false;
```

Img 21 Proměnné a reference komponentů

Útok hráče

Pro souboj hráče s nepřáteli je nutné mít možnost způsobit poškození. K tomu používám skript s názvem PlayerCombat. Zde se ptám, jestli hráč stiskl tlačítko pro útok. Pokud ano, nejprve se ujistím, že: není ve vzduchu anebo neblokuje. Pokud splním tyto podmínky, mohu se začít věnovat útoku samotnému. Hráči jsem nastavil několik důležitých proměnných, a to: dosah útoku, rychlost útoku a sílu poškození. Dosah je důležitý pro detekci přesahu. Hráč má na sobě určený bod, který je středem kruhu, dosah je poloměr. Pokaždé, když hráč zaútočí, ptám se, jestli se v tomto kruhu nachází nepřítel (přesah Colliderů s kruhem). Aby byla jistota, že se útočí jen do nepřátel, v podmínce pro detekci v přesahu se ptám, jestli je z vrstvy (layer) nepřátel. Pokud ano, způsobí hráč poškození každému nepříteli, který splňuje podmínky.

Dále bylo potřeba omezit rychlost útoku. Pomocí jednoduché podmínky po každém útoku nastavím čas na nulu. Pokud je uplynulý čas větší nebo roven rychlosti útoku, může hráč zaútočit znovu. Pokud ne, útok se nespustí. Pak už jen zbývalo zajistit, aby poškození bude uděleno až při

```
if(Time.time >= nextAttackTime)
{
    if(animator.GetBool("IsJumping") == false && animator.GetBool("Block") == false)
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {
        animator.SetTrigger("Attack");
```

Img 22 Kód podmínky dalšího útoku

```
public float attackRange = 0.5f;
public int attackDamage = 40;

public float attackRate = 2f;
float nextAttackTime = 0f;
```

Img 25 Proměnné nastavující útok

```
nextAttackTime = Time.time + 1f / attackRate;
```

Img 24 Časovač

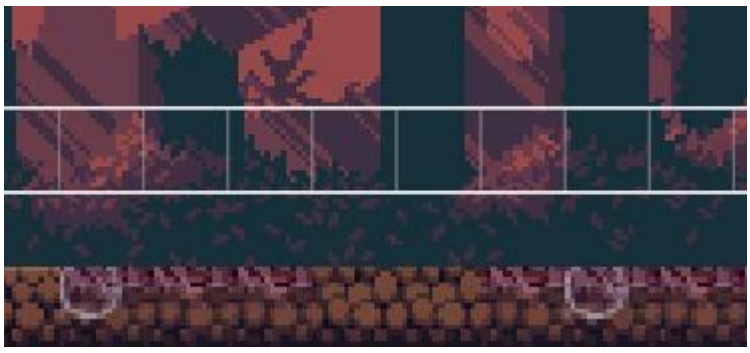
```
void Attack()
{
    Collider2D[] hitEnemies = Physics2D.OverlapCircleAll(attackPoint.position, attackRange, enemyLayers);

    foreach(BoxCollider2D enemy in hitEnemies)
    {
        enemy.GetComponent<enemyHealth>().TakeDamage(attackDamage);
        Debug.Log("Enemy Hit!");

        PlayerHealth playerHeal = GetComponent<PlayerHealth>();

        playerHeal.PlayerGetHealth(2);
    }
}
```

Img 23 Kód útoku na nepřátele a příklad reference jiného skriptu

správném snímku animace. To není těžké. Stačí si u daného snímku vytvořit událost (event) a přiřadit jí metodu (funkci). Aby to bylo možné, musí být funkce veřejná.

Pohyb nepřátel a Detekce hráče

Pohyb nepřátel představovalo složitější záležitost. Na pohyb hráče stačí vstup a přidání vektoru. Problém je, že u nepřátel vstup musí zajistit hra. Naštěstí se mi povedlo najít způsob, kde byla ta nejtěžší část, tj. orientace po mapě, vyřešena. V Unity Asset Storu jsem si stáhl projekt A*, což je komunitní balíček pro zjednodušení výroby „AI Pathfinding“. V něm jsem nastavil, co je mapa a kudy se mohou nepřátelé pohybovat. Stačilo už jen zajistit pohyb po vyhrazeném prostoru. Tím se dostáváme k detekci hráče.



Img 26 Vyhrazený prostor pro orientaci

Aby nepřítel věděl, kam a jak má jít, musel jsem mu nejprve určit cíl (hráč). Poté si načrtne cestu od sebe k hráči a rozdělí si ji na úseky pomocí Waypointů. Podle vzdálenosti od hráče aplikuje rychlost a jde po jednotlivých Waypointech. V tento moment jsem se naučil pracovat s metodou

InvokeRepeating. Aby se nestalo, že nepřítel dojde, kde hráč sice byl, ale mezitím už se přesunul jinam, musíme funkci opakovat. Šla by opakovat v metodě Update – opakuje se každý snímek, nebo Fixed Update – opakuje se po časovém úseku, který je fixní. Nevýhodou je, že bych opakoval všechny funkce, nikoli jen pouhé hledání cesty. Pomocí InvokeRepeating mohu určit, kterou funkci chci začít samostatně opakovat a jak často. Teď už nepřítel aktualizuje pozici hráče a ví kde se nachází.

Img 27 Werewolf běžící na hráče s načrtlou cestou

Životy a útoky nepřátel

Životy a útok jsem řešil velmi podobným způsobem. Nemohl jsem ale použít stejný skript jako u hráče. Nepřátelé mají pár jiných funkcí, než má hráč. Hlavní rozdíl je opět to, že postavu ovládá hra, ne hráč. Životy jsem vyřešil skoro identickým skriptem jako u hráče. Došlo jen k pár individuálním úpravám, především pak byly jiné reference komponentů.

```
public void TakeDamage(int damage)
{
    currentHealth -= damage;

    rb.velocity = Vector3.zero;

    animator.SetTrigger("Hurt");

    if(currentHealth <= 0)
    {
        Die();

        score.addScore(1);
    }
}
```

Img 28 Kód zranění nepřátel

```
void Die()
{

    Debug.Log("Enemy Died");

    cc2D.enabled = false;
    bc2D.enabled = false;

    animator.SetBool("IsDead", true);

    Debug.Log("Velocity set");
    rb.velocity = Vector3.zero;

    EnemyAI enemyAI = GetComponent<EnemyAI>();
    enemyAI.enabled = false;
```

Img 29 Kód smrti nepřátel

S útokem to bylo složitější. Využil jsem kolizí, abych věděl, kdy je nepřítel v dostatečné vzdálenosti. To řeším ve skriptu DetectionRange. Kolize jsem detekoval pomocí Colliderů, ale se zapnutou funkcí Trigger, tudíž detekovaly kolize, ale byly nehmotné a dalo se chodit skrz. Pokud dojde ke kolizi s hráčem, pozastaví se opakování funkce pohybu pomocí veřejné reference a spustí se InvokeRepeating na funkci Attacking. Tímto způsobem nezaútočí nepřítel jen jednou, ale každých několik sekund. Protože Skeleton Archer útočí na dálku, potřebuje větší vzdálenost. Proto používá skript velmi podobný DetctionRange, ale s důležitými rozdíly – ShootRange. Pokud hráč vyjde z Collideru, obnoví se funkce pohybu, aby nepřítel mohl znovu zaútočit.

```
void OnTriggerEnter2D(Collider2D collision)
{
    Debug.Log(collision.name);
    enemyai.PausePath();

    InvokeRepeating("Attacking", 0f, attackRate);
}
```

Img 31 Kód počátku útočení na hráče

```
private void OnTriggerExit2D(Collider2D collision)
{
    CancelInvoke();

    enemyai.ResumePath();
}
```

Img 30 Kód obnovení pohybu

Střílení z luku

Když útočí Skeleton Warrior nebo Black Werewolf, nejde o nic složitého, ale když útočí Skeleton Archer, je potřeba vytvořit klon šípu, který letí z luku, a sledovat jej. Tento klon se dá vytvořit pomocí metody Instantiate. Hned poté, co šíp vytvořím, přiřadím mu rychlost a směr pohybu. Šíp jsem nastavil tak, že koliduje jen s hráčem. Pokud se to stane, spustí funkci zranění u hráče a smaže se. Stejně tak se smaže, pokud se propadne pod stanovenou výšku. Je to kvůli tomu, aby se snížil nápor na techniku. Kdybychom šíp neodmazali, byl by stále načítán, a kdyby jich bylo dvacet stále načítaných, podepsalo by se to na výkonu a plynulosti hry.

```
void OnTriggerEnter2D (Collider2D hitInfo)
{
    PlayerHealth player = hitInfo.GetComponent<PlayerHealth>();

    if (player != null)
    {
        player.PlayerTakeDamage(attackDamage);
        Destroy(gameObject);
    }
```

Img 32 Kód útoku na hráče šípem

```
public void Shoot()
{
    Instantiate(arrowPrefab, firePoint.position, firePoint.rotation);
}
```

Img 34 Kód vytvoření klonu šípu

```
void Start()
{
    rb.velocity = transform.right * speed;
}

private void Update()
{
    if (transform.position.y < deadZone)
    {
        Destroy(gameObject);
    }
}
```

Img 33 Kód přidání pohybu a zničení šípu

Spawnování nepřátel

Jak jsem již řekl, chtěl jsem, aby nepřátelé mohli stále chodit na hráče. Vytvořil jsem tedy pár míst na mapě, které vytvářejí klony pomocí skriptu Spawn. Podobně jako u šípu využívám metodu Instantiate. Aby se ale neobjevili všichni nepřátelé najednou, využívám generátoru náhodného čísla mezi dvěma hodnotami. Ta hodnota, která se určí, je pak čas do vytvoření se nového klonu. Po jeho vytvoření se nastaví hodnota nová, stejným způsobem.

```
void Update()
{
    if (timer < nextSpawnTime)
    {
        timer += Time.deltaTime;
    }
    else
    {
        SpawnEnemy();
        timer = 0;
        nextSpawnTime = Random.Range(minimalNextSpawnTime, maxNextSpawnTime);
    }
}
```

Img 35 Kód náhodně spawnovaných nepřátel

Skóre a UI

Aby hra měla nějaký cíl, přičte se hráči za každé zabití nepřítele skóre. Skóre se zobrazuje jako číslo v levém horním rohu, ale na zobrazení těchto hodnot potřebujeme využít tzv. UI (User interface) – uživatelské rozhraní. To se promítá navrch scény hry a může obsahovat interaktivní prvky, např. tlačítka. Důležité je, že pokud chci ovládat prvky v UI ve skriptu ScoreScript, musím

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

Img 37 Rozšíření působnosti

```
public void addScore(int score)
{
    playerScore += score;

    scoreText.text = playerScore.ToString();
}
```

Img 36 Kód na zobrazení skóre

si rozšířit působnost o UnityEngine.UI. Bez toho bych neměl možnost referencí atypických komponentů (např. textů). Na této úrovni také zobrazuji ukazatel zdraví.

Zapnutí hry a scény

Poslední věc, kterou bylo potřeba vytvořit, bylo hlavní menu a vstup do hry. Mezi nimi přepínáme pomocí scén. Vytvořil jsem tedy dvě scény a pomocí interaktivních tlačítek a funkcí, které jsem jim přiřadil ze skriptu GameRestar, je možné hru vypnout, anebo začít hrát. Když hráč zemře, dostane možnost návratu do menu nebo hrát znovu. Pro přepínání scén je potřeba také rozšířit působnost ve skriptu, tentokrát pomocí UnityEngine.SceneManagment.

```
using UnityEngine.SceneManagement;
public class MainMenu : MonoBehaviour
{
    public void PlayGame()
    {
        SceneManager.LoadScene("Game");
    }

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

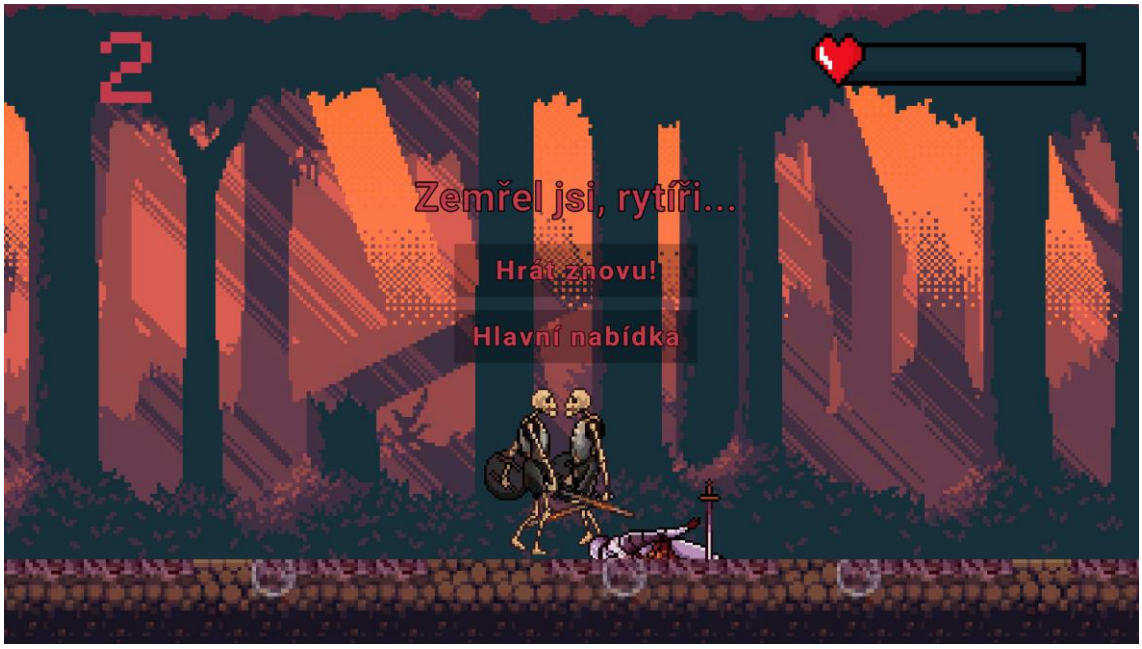
Img 38 Funkce tlačítek v menu

```
using UnityEngine.SceneManagement;

public class GameRestar : MonoBehaviour
{
    public void restartGame()
    {
        SceneManager.LoadScene(SceneManager.GetActiveScene().name);
    }

    public void goToMenu()
    {
        SceneManager.LoadScene("Menu");
    }
```

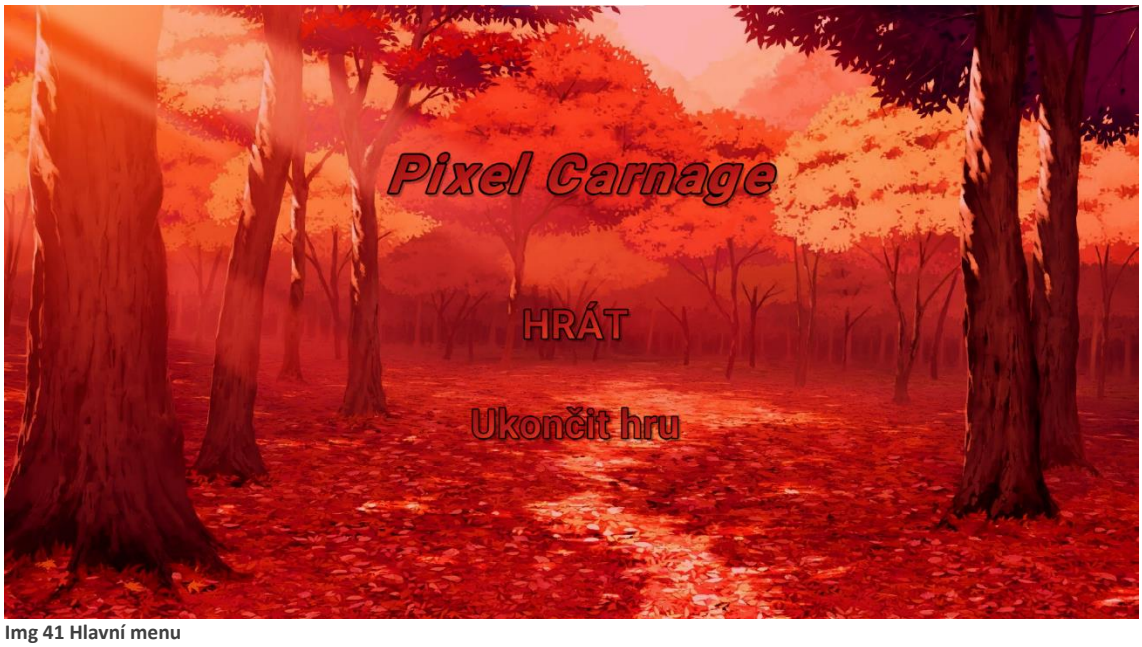
Img 39 Funkce tlačítek po smrti hráče

Výsledky

Výsledkem tohoto projektu je relativně funkční hra a spousta zkušeností. Povedlo se mi vytvořit hru, splnit žánr a nabídnout tři druhy nepřátel. Dva na blízko: Warrior – jednoduchý na souboj a Werewolf, který je rychlý a hodně silný. Třetí z nich, Archer, který je dokonce na dálku. Střílí z luku a je funkční a bez chyb. Hráč dostává skóre za každé zabití, ztrácí a získává životy, může blokovat, aby neztrácel životů tolik. Pokud zemře, může hrát znovu, pokud chce. Zároveň jsem se zlepšil s Unity, a především už rozumím základům v programovacím jazyce C#. Projekt pro mne byl formou cesty k porozumění kódu. Myslím, že zkušenost z vývoje této hry mi přinesla cenné dovednosti.



Img 40 Menu po smrti hráče



Img 41 Hlavní menu

Závěr

Myslím si, že práce ve formě projektu je skvělý způsob, jak udělat školu hrou. Téma vytvoření počítačové hry je velká výzva. Člověk si musí celý projekt promyslet a rozplánovat. Tento projekt zdaleka není dokončený a nabízí se možnosti pro jiné žáky, jak jej zlepšit, zdokonalit a dotáhnout. Ale asi nejpodstatnější pro mě je to, že se jedná o způsob, jak si najít cestu k programování. Programovací jazyk C# je hodně rozšířený a používaný. Je to jazyk, který je stále na vzestupu a po programátorech je poptávka.

Zdroje

1. MICROSOFT. *Prohlídka jazyka C#*. Online. Microsoft 2024. Dostupné z: https://learn.microsoft.com/cs-cz/dotnet/csharp/tour-of-csharp/ [cit. 2024-04-10]

2. *UNITY*. Webové sídlo. Dostupné z: https://unity.com [cit. 2024-04-10]

3. WIKIPEDIA. *C Sharp*. Online. Wikipedia 2024. Dostupné z: https://cs.wikipedia.org/wiki/C_Sharp [cit. 2024-04-10]

4. WIKIPEDIA. *Unity (herní engine)*. Online. Wikipedia 2024. Dostupné z: https://cs.wikipedia.org/wiki/Unity_(hern%C3%AD_engine) [cit 2024-04-10]

5. WIKIPEDIA. *Hack and slash*. Online Wikipedia 2024. Dostupné z: https://en.wikipedia.org/wiki/Hack_and_slash [cit. 2024-04-10]