



STŘEDNÍ ŠKOLA PRŮMYSLOVÁ
A UMĚlecká, opava

ZÁVĚREČNÁ STUDIJNÍ PRÁCE

Dokumentace

Hranice prostoru a perspektivy ve videohrách

Lukáš Olbrecht

Obor: 18-20-M/01 INFORMAČNÍ TECHNOLOGIE
se zaměřením na počítačové sítě a programování

Třída: IT4

Školní rok: 2023/2024

Poděkování

Rád bych vyjádřil své poděkování panu Markovi Lučnému za cennou zpětnou vazbu a podporu, kterou mi poskytoval během tvorby této závěrečné práce.

Prohlašuji, že jsem závěrečnou práci vypracoval samostatně a uvedl veškeré použité informační zdroje.

Souhlasím, aby tato studijní práce byla použita k výukovým účelům na Střední průmyslové a umělecké škole v Opavě, Praskova 399/8.

V Opavě 31. 12. 2023

podpis autora práce

ABSTRAKT

Tento projekt se zabývá implementací optických klamů inspirovaných hrou Superliminal.

Projekt je realizován ve dvou různých herních enginech – Unity a Unreal Engine.

V Unity je vytvořena hlavní mechanika hry Superliminal, která je známá jako Forced Perspective. Tato technika využívá perspektivu hráče k vytvoření iluze změny velikosti objektů. Například při pohledu z určitého úhlu může objekt vypadat menší nebo větší, než ve skutečnosti je, a tento efekt je následně zachován i při interakci s objektem.

Unreal Engine byl naopak použit pro implementaci systému neviditelných portálů. Tyto portály umožňují plynulý pohyb hráče mezi prostory, aniž by zaznamenal, že byl ve skutečnosti přenesen na zcela jiné, vzájemně nenavazující místo. Portály lze dále využít pro změnu velikosti hráče či vytvoření tzv. nekonečné chodby, kde hráč prochází opakující se sekvencí prostorů.

ABSTRACT

This project deals with the implementation of optical illusions inspired by the game Superliminal. The project is implemented in two different game engines - Unity and Unreal Engine.

The main game mechanic of Superliminal is created in Unity, which is known as Forced Perspective. This technique uses the player's perspective to create the illusion of resizing objects. For example, when viewed from a certain angle, an object can appear smaller or larger than it actually is, and this effect is then maintained when interacting with the object. Unreal Engine, on the other hand, was used to implement the system of invisible portals. These portals allow the player to move seamlessly between spaces without noticing that they have actually been transported to a completely different, unrelated location. Portals can also be used to change the size of the player, which enables effects such as enlarging, shrinking or creating so-called endless corridors where the player goes through a repeating sequence of spaces.

OBSAH

1 TEORETICKÁ A METODICKÁ VÝCHODISKA	6
1.1 Pohyb hráče v Unity Engine	6
1.1.1 PlayerMovement.cs	6
Ukázka klíčových funkcí skriptu	6
1.1.2 MouseLook.cs	7
Ukázka klíčových funkcí skriptu	7
1.1.3 Spolupráce skriptů	8
2 Forced perspective	9
Ukázka klíčových funkcí skriptu	12
3 Pohyb hráče v Unreal Engine	15
4 Neviditelné portály	18
VYUŽITÉ TECHNOLOGIE	20
ZÁVĚR	21
SEZNAM POUŽITÝCH INFORMAČNÍCH ZDROJŮ	22

ÚVOD

- Tato práce se zabývá implementací inovativních herních mechanik inspirovaných optickými klamy a portálovými systémy, jak je známe například ze hry Superliminal. Cílem projektu je vytvořit funkční demonstrační aplikaci ve dvou předních herních enginech: Unity a Unreal Engine.

Projekt se zaměřuje na dvě specifické mechaniky: Forced Perspective v Unity

- a portálové systémy v Unreal Engine.
- Herní mechaniky založené na perspektivě a portálech představují zajímavé vývojářské výzvy, které vyžadují hluboké pochopení 3D prostoru, práce s kamerou a iluzemi. Tyto prvky nejen obohacují herní zážitek, ale také posouvají hranice technických možností herních enginů. Inspirací pro práci byla hra Superliminal, která ukazuje potenciál optických klamů a změny měřítka v herním designu.

Cíle práce

- Implementovat mechaniku Forced Perspective v Unity, která mění velikost objektů na základě hráčovy perspektivy.
- Vytvořit portálové systémy v Unreal Engine, které umožňují plynulý přechod mezi prostory, včetně efektů zvětšování, zmenšování a nekonečných chodeb.

1. Kapitola se věnuje teoretickým a metodickým východiskům, která objasňují základní principy použitých herních mechanik a postupů.
2. Kapitola popisuje technologie využité při implementaci, konkrétně enginy Unity a Unreal.
3. Kapitola se zaměřuje na způsoby řešení a detailně popisuje postupy použité při vývoji mechanik.
4. Kapitola prezentuje výsledky řešení, zahrnuje dosažené výstupy a obsahuje uživatelský manuál k vytvořeným mechanikám.

Závěr shrnuje hlavní poznatky práce a hodnotí dosažené cíle.

1 TEORETICKÁ A METODICKÁ VÝCHODISKA

1.1 Pohyb hráče v Unity Engine

Pohyb hráče je základním prvkem většiny her a jeho správná implementace ovlivňuje celkový zážitek ze hry. V tomto projektu je pohyb hráče implementován pomocí dvou skriptů: [PlayerMovement.cs](#) a [MouseLook.cs](#), které zajišťují ovládání pohybu postavy a pohledu hráče v prostředí Unity.

1.1.1 PlayerMovement.cs

Skript [PlayerMovement.cs](#) je odpovědný za horizontální pohyb hráče, skok a gravitaci. Využívá komponentu [CharacterController](#), která usnadňuje práci s pohybem v 3D prostoru bez nutnosti ruční implementace kolizí.

Horizontální pohyb: Pohyb hráče je řízen pomocí vstupů z klávesnice ([Horizontal](#) a [Vertical](#)) a je převeden na vektor pohybu ve světových osách [X](#) a [Z](#). Pohyb je normalizován, aby byla rychlosť konstantní i při pohybu úhlopříčně.

Skok je realizován na základě kontroly, zda se hráč nachází na zemi, což je ověřováno metodou [Physics.CheckSphere](#). Pokud hráč stiskne tlačítko pro skok a je na zemi, vypočítá se vertikální rychlosť na základě výšky skoku a gravitační síly.

Skript aplikuje konstantní gravitaci směrem dolů. Pokud hráč opustí povrch (např. po skoku), gravitace jej začne postupně zrychlovat směrem k zemi.

Ukázka klíčových funkcí skriptu

- Pohyb hráče na základě vstupu

```
float x = Input.GetAxisRaw("Horizontal");
float z = Input.GetAxisRaw("Vertical");
Vector3 move = transform.right * x + transform.forward * z;
controller.Move(move.normalized * speed * Time.deltaTime);
```

- Implementace skoku

```
if (Input.GetButtonDown("Jump") && isGrounded)
{
    velocity.y = Mathf.Sqrt(jumpHeight * -2f * gravity);
}
```

- Aplikace gravitace

```
velocity.y += gravity * Time.deltaTime;
controller.Move(velocity * Time.deltaTime);
```

1.1.2 MouseLook.cs

Skript MouseLook.cs zajišťuje ovládání pohledu hráče pomocí myši. Tento skript je klíčový pro vytvoření tzv. first-person perspektivy, která je typická pro hry z pohledu první osoby.

Ovládání pohledu: Pohyb myší ve vodorovném směru (osa X) otáčí celým tělem hráče, zatímco pohyb myší ve svislém směru (osa Y) ovládá otáčení kamery nahoru a dolů.

Clamping rotace: Rotace kamery ve vertikální ose je omezena metodou `Mathf.Clamp`, aby hráč nemohl otočit pohled příliš daleko za limity (např. za hlavu postavy).

Uzamknutí kurzoru: Pro plynulý pohyb pohledu je kurzor myši uzamčen do středu obrazovky pomocí `Cursor.lockState`.

Ukázka klíčových funkcí skriptu

- Ovládání pohledu myší

```
float mouseX = Input.GetAxis("Mouse X") * mouseSensitivity * Time.deltaTime;
float mouseY = Input.GetAxis("Mouse Y") * mouseSensitivity * Time.deltaTime;
```

- Clamping a vertikální rotace

```
xRotation -= mouseY;
xRotation = Mathf.Clamp(xRotation, -90f, 90f);
transform.localRotation = Quaternion.Euler(xRotation, 0f, 0f);
```

- Otáčení hráčova těla

```
playerBody.Rotate(Vector3.up * mouseX);
```

1.1.3 Spolupráce skriptů

Skripty [PlayerMovement.cs](#) a [MouseLook.cs](#) spolupracují na vytvoření komplexního systému pohybu a ovládání pohledu hráče:

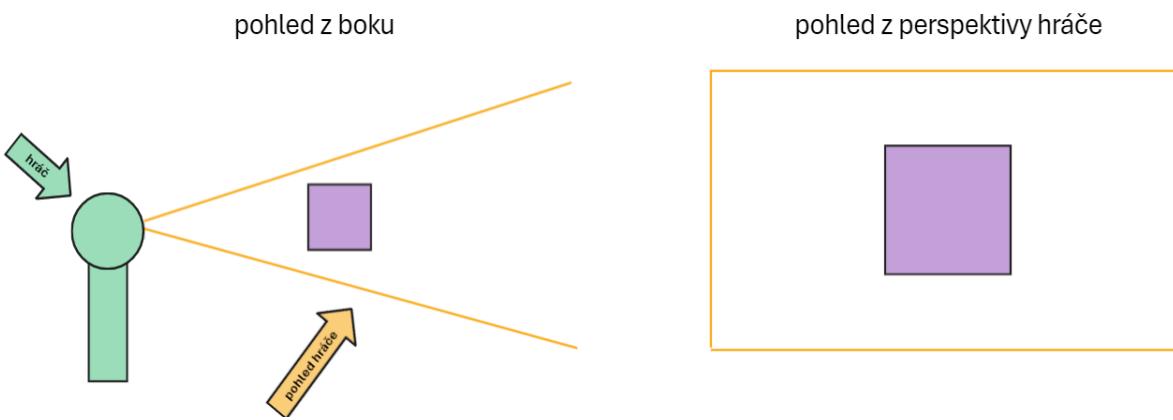
- [PlayerMovement.cs](#) zajišťuje pohyb a skákání hráče.
- [MouseLook.cs](#) ovládá pohled hráče a rotaci jeho těla podle pohybu myši.
- Díky kombinaci těchto skriptů je dosaženo plynulého ovládání postavy ve first-person perspektivě. Hráč může pohybovat postavou ve všech směrech, skákat a otáčet se po vertikální i horizontální ose.

2 FORCED PERSPECTIVE

Jednou z klíčových mechanik hry *Superliminal* je tzv. **forced perspective** – efekt, který umožňuje měnit velikost objektů na základě perspektivy hráče. Tento princip spočívá v tom, že objekty zdánlivě zachovávají svou velikost i když jsou ve skutečnosti větší.

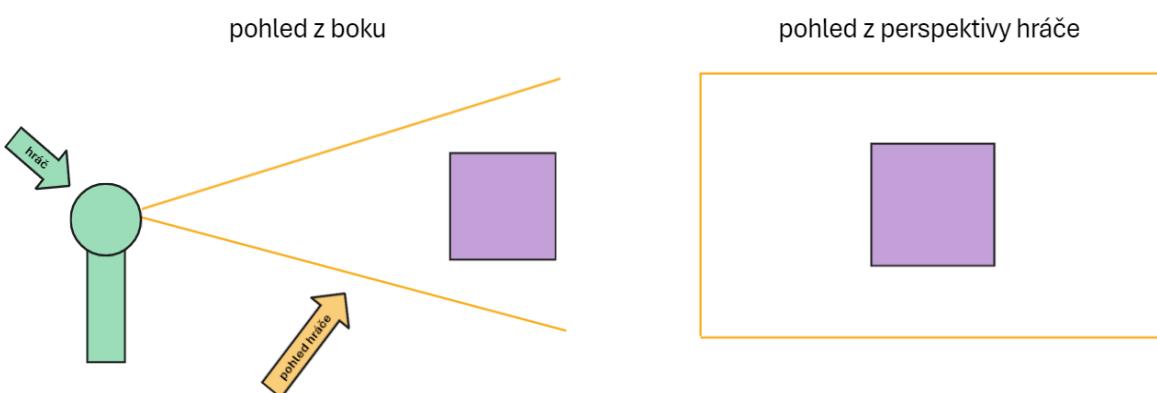
1. Startovní pozice objektu

- Představme si hráče, který drží objekt ve vzdálenosti 1 metru.
- Z této vzdálenosti objekt vypadá určitým způsobem – například vyplňuje určitou část zorného pole hráče.



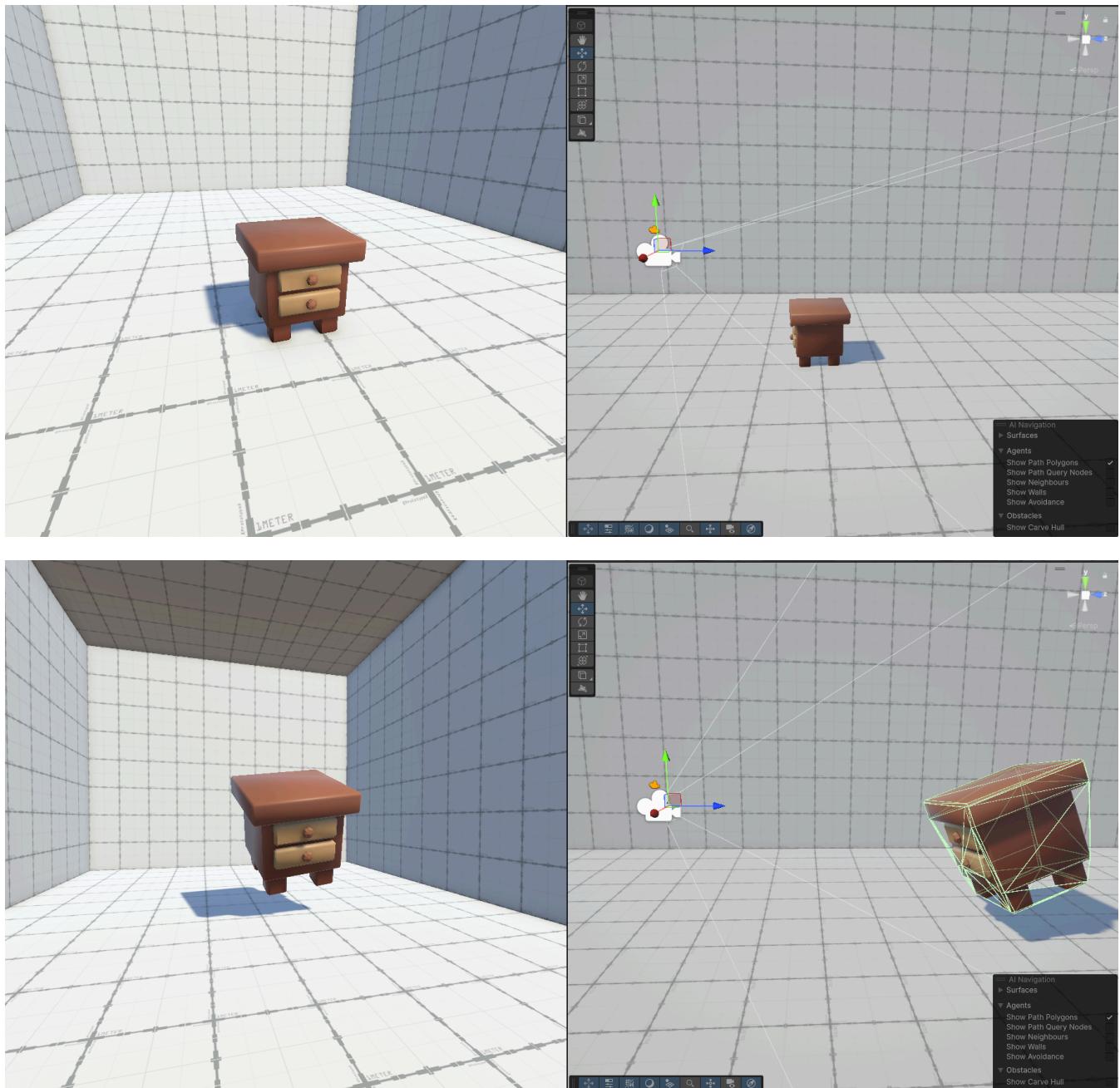
2. Změna vzdálenosti objektu

- Nyní hráč posune objekt dále od sebe, například do vzdálenosti 2 metrů.
- Z pohledu hráče objekt stále vyplňuje stejnou část zorného pole, ale jeho velikost ve světě je dvojnásobná



Jak to funguje?

Z pohledu kamery (hráče) je klíčem k tomuto efektu zachování **úhlové velikosti objektu** ve zorném poli. Pokud je objekt dál, jeho skutečná velikost se zvětší, aby se vizuálně jevil stejně velký. Tento princip lze simulovat pomocí přepočtu vzdálenosti objektu od kamery a přizpůsobení jeho měřítka.



Klíčové principy skriptu

- Skript umožňuje hráči vzít objekt pohledem (raycastem) a přiřadit jej jako držený. Objekt se přichytí k hráči, deaktivuje svou fyziku (Rigidbody) a jeho velikost i pozice se přizpůsobí hráčově perspektivě.
- Při držení objektu se jeho pozice dynamicky upravuje tak, aby byl vždy před překážkami, což zabraňuje tomu, aby objekt kolidoval se zdmi nebo jinými překážkami v prostředí.
- Objekt mění svou velikost podle vzdálenosti od hráče, čímž simuluje efekt perspektivy. Pokud je objekt dále od hráče, zvětšuje se; pokud je blíže, zmenšuje se. Velikost se vypočítává na základě poměru původní velikosti objektu a vzdálenosti od hráče.
- Skript vytváří mřížku bodů (grid) na povrchu objektu, které jsou používány pro detekci kolizí s okolním prostředím. Tato mřížka umožňuje přesné přizpůsobení pozice objektu při držení hráčem, aby byl vždy volný prostor před ním.
- Skript používá viewport kamery k výpočtu nových pozic a orientace objektu, což zajišťuje, že objekt zůstane správně zarovnaný z pohledu hráče, ať už je vzdálený nebo blízký.

Ukázka klíčových funkcí skriptu

Funkce GetBoundingBoxPoints()

```
private Vector3[] GetBoundingBoxPoints()
{
    Vector3 size = heldObject.GetComponent<Renderer>().localBounds.size;
    Vector3 x = new Vector3(size.x, 0, 0);
    Vector3 y = new Vector3(0, size.y, 0);
    Vector3 z = new Vector3(0, 0, size.z);
    Vector3 min = heldObject.GetComponent<Renderer>().localBounds.min;

    // Body bounding boxu
    Vector3[] bbPoints =
    {
        min,
        min + x,
        min + y,
        min + x + y,
        min + z,
        min + z + x,
        min + z + y,
        min + z + x + y
    };

    return bbPoints;
}
```

Skript vytvoří obdélník okolo viditelné části objektu

Funkce SetupGrid()

```

private Vector3[,] SetupGrid()
{
    // Vypočítáme délky hran ve vodorovném a svislém směru
    float rectHrLength = right.x - left.x;
    float rectVertLength = top.y - bottom.y;

    // Krok mřížky (horizontální a vertikální)
    Vector3 hrStep = new Vector2(rectHrLength / (NUMBER_OF_GRID_COLUMNS - 1), 0);
    Vector3 vertStep = new Vector2(0, rectVertLength / (NUMBER_OF_GRID_ROWS - 1));

    // Inicializace mřížky
    Vector3[,] grid = new Vector3[NUMBER_OF_GRID_ROWS, NUMBER_OF_GRID_COLUMNS];
    grid[0, 0] = new Vector3(left.x, bottom.y, left.z);

    // Generování bodů mřížky
    for (int i = 0; i < grid.GetLength(0); i++) // Řádky
    {
        for (int j = 0; j < grid.GetLength(1); j++) // Sloupce
        {
            if (i == 0 && j == 0) continue;

            if (j == 0) // První bod v řádku
            {
                grid[i, j] = grid[i - 1, 0] + vertStep;
            }
            else // Následující body v řádku
            {
                grid[i, j] = grid[i, j - 1] + hrStep;
            }
        }
    }

    return grid;
}

```

Vytvoří síť bodů skrze které se později posílají raycasty.

Funkce MoveInFrontOfObstacles()

```
void MoveInFrontOfObstacles()
{
    float closestZ = 1000;

    // Projdeme body mřížky a detekujeme překážky
    foreach (Vector3 gridPoint in shapedGrid)
    {
        RaycastHit hit = CastTowardsGridPoint(gridPoint, wallLayer +
pickUpLayer);
        if (hit.collider == null) continue;

        // Získáme bod nejbližší překážky
        Vector3 wallPoint = transform.InverseTransformPoint(hit.point);
        if (wallPoint.z < closestZ)
        {
            closestZ = wallPoint.z;
        }
    }

    // Posun objektu o určitý offset od překážky
    float boundsMagnitude =
heldObject.GetComponent<Renderer>().bounds.extents.magnitude *
heldObjectTF.localScale.x;
    Vector3 newLocalPos = heldObjectTF.localPosition;
    newLocalPos.z = closestZ - boundsMagnitude;
    heldObjectTF.localPosition = newLocalPos;
}
```

Objekt posuneme před nejbližší překážku aby byl pořád v blízkosti hráče

3 POHYB HRÁČE V UNREAL ENGINE

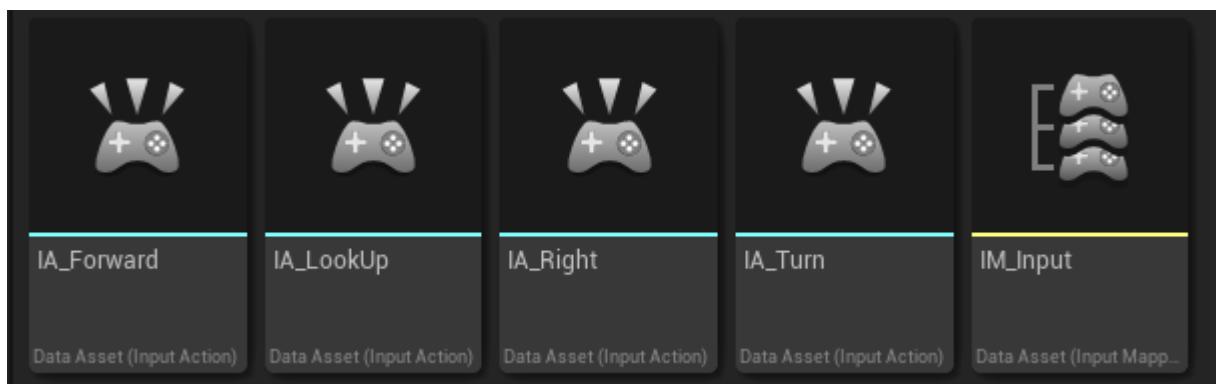
Pohyb hráče v Unreal Engine se výrazně liší od Unity především v přístupu k jeho implementaci. Zatímco v Unity je nutné pohyb hráče programovat skrze vlastní kód, v Unreal Engine jsou základní mechanismy, jako například chůze, přednastavené. Stačí je jednoduše nastavit a propojit.

Pro zajištění ovládání hráče je nutné vytvořit Input Mapping Context (IMC) a Input Action (IA).

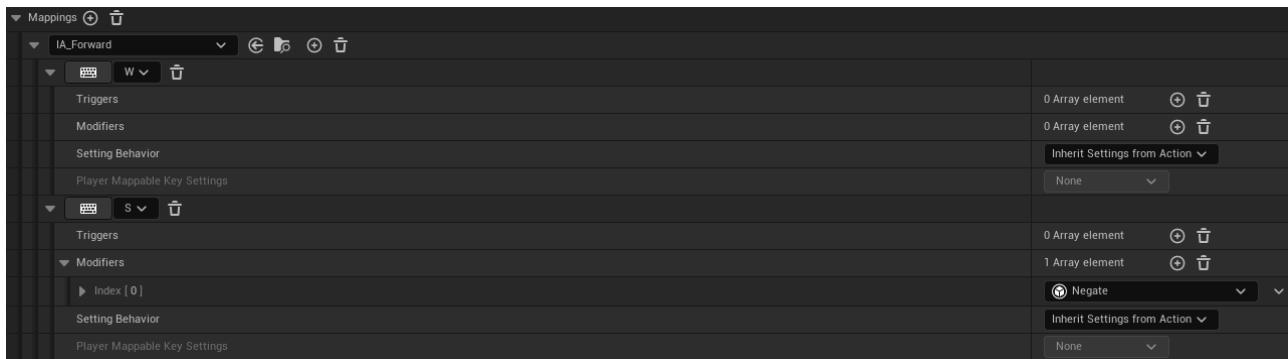
- **Input Action (IA)** slouží jako prostředník mezi vstupy (například pohyb myši nebo stisk klávesy) a tím, co se má stát ve hře.
- **Input Mapping Context (IMC)** následně umožňuje přiřadit jednotlivým vstupům konkrétní akce. V IMC tedy rozhodujeme, co má každý vstup vykonat – například přiřazení klávesy „W“ k pohybu vpřed nebo pohybu myší k otáčení kamery.

Tento systém výrazně zjednoduší práci s ovládáním a poskytuje větší flexibilitu při nastavování interakcí ve hře.

K jednoduchému pohybu nám tedy stačí IA jen 4x:



Uvnitř IMC nastavení pro chůzi dopředu a dozadu vypadá takto:



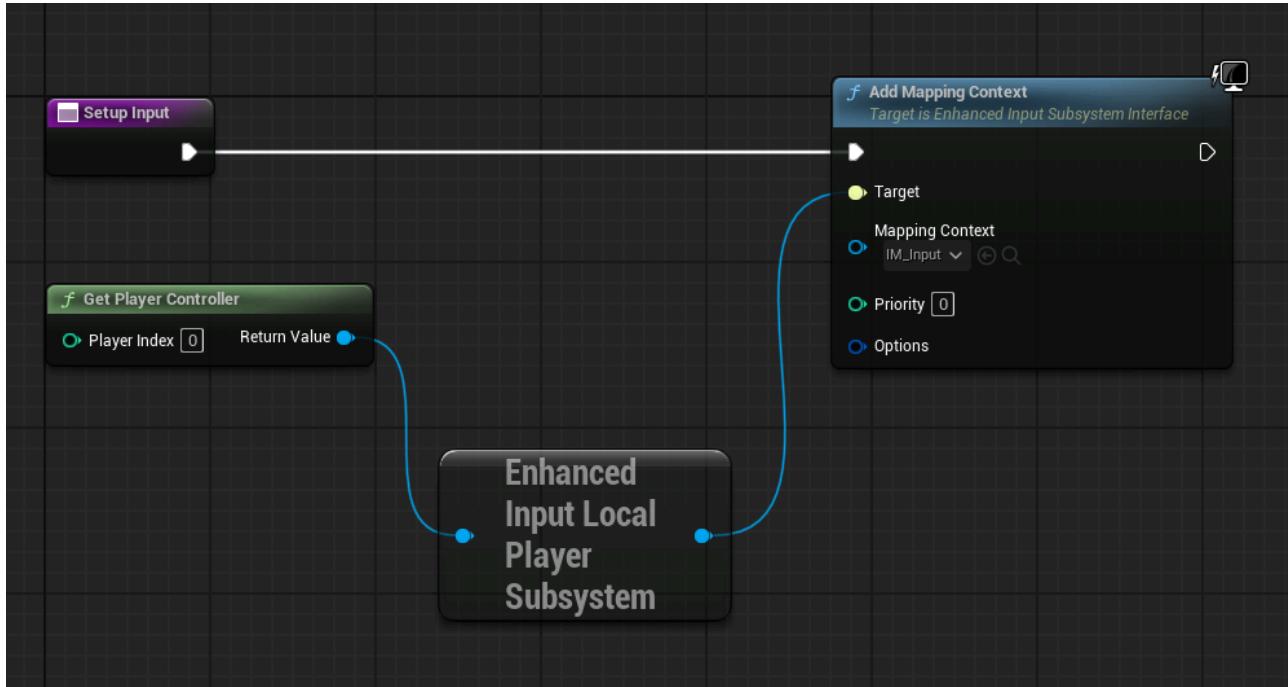
Jak je vidět na přiloženém obrázku, pohyb dopředu i dozadu využívá stejnou osu. Díky tomu stačí vytvořit pouze jedno IA, které obsluhuje oba směry pohybu. Pro zajištění pohybu dozadu je k této akci přiřazen modifikátor, který neguje hodnotu osy. Tento přístup zjednoduší implementaci ovládání a minimalizuje množství potřebných akcí, protože umožňuje efektivně využít jednu osu pro více směrů.

Stejný princip je použit i pro pohyb do stran, tedy doprava a doleva. Pohyb oběma směry využívá stejnou osu, přičemž pohyb doleva je realizován negací hodnoty této osy pomocí modifikátoru. Tento jednotný přístup umožňuje snadnou správu vstupů a snižuje množství potřebných IA, což vede k jednodušší a přehlednější konfiguraci ovládání.

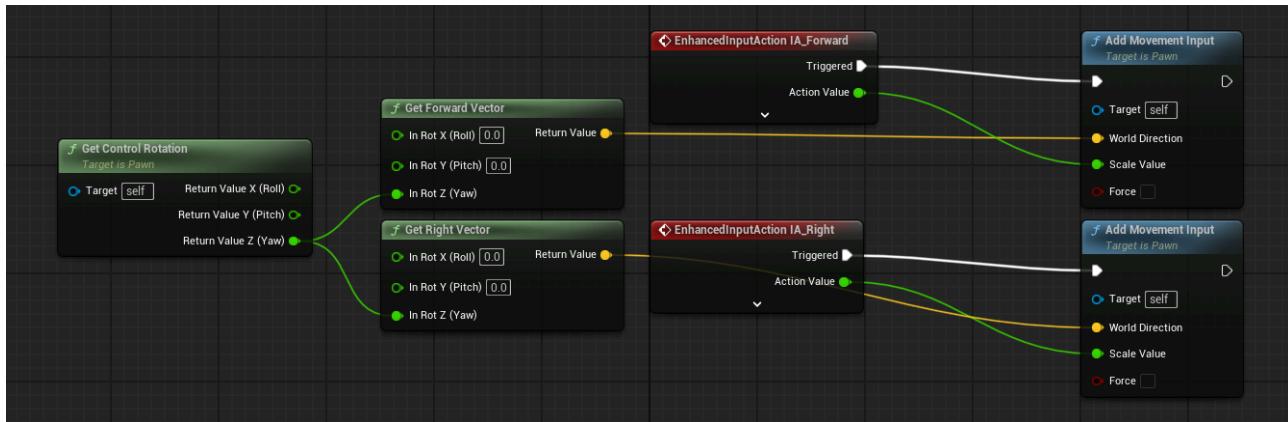
Toto ale není vše co je potřebné udělat proto abychom mohli pohybovat hráčem. V objektu hráče musíme vytvořit funkci která nám dovolí užívat náš IMC vytvoření této funkce je jednoduché.

To však není vše, co je potřeba udělat, aby bylo možné pohybovat hráčem. V blueprintu hráče je nutné vytvořit funkci, která umožní používat náš IMC. Tento krok je zásadní, protože právě tímto způsobem přiřazujeme náš IMC konkrétnímu hráči, aby mohl reagovat na vstupy.

Na následující straně je grafické znázornění této funkce, kde je jasně vidět propojení uzelů a logika přiřazení IMC hráči.



Nyní, když jsme mapping context úspěšně přiřadili hráči, můžeme definovat konkrétní akce, které se mají provést při aktivaci tohoto mapping contextu. To nám umožňuje propojit vstupy hráče, jako je pohyb, otáčení kamery nebo interakce, s odpovídajícími funkcemi nebo logikou ve hře.



4 NEVIDITELNÉ PORTÁLY

Pro vytvoření portálu v Unreal Engine je potřeba sestavit objekt, který bude obsahovat několik klíčových komponent. Základem je jednostranná plocha (Plane), která slouží jako "plátno" pro promítání obrazu. Aby portál vypadal vizuálně přirozeněji, doporučuji kolem této plochy vytvořit rám. Ten slouží nejen jako estetický prvek, ale také může pomoci s materiálovým odlišením portálů, pokud jich má hra více.

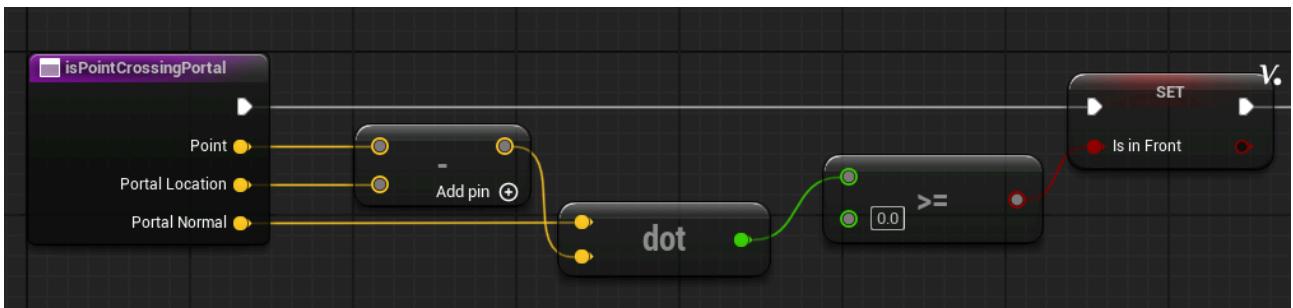
Důležitou součástí portálu je kamera, konkrétně Screen Capture Component 2D, která zachytává obraz scény z pohledu portálu. Tento obraz je následně promítán na druhý portál, což vytváří dojem, že hráč skutečně vidí skrz portál do jiného prostoru. Například portál_1 využívá svou kameru k zachycení obrazu scény za portálem_2. Obraz zachycený kamerou portálu_1 se promítá na plochu portálu_2, což je realizováno pomocí speciálního materiálu, který využívá texturu generovanou komponentou Render Target. Tímto způsobem dochází k dynamickému propojení obou portálů.

Aby bylo možné portálem plynule procházet, doporučuji upravit několik grafických nastavení Unreal Engine. Zaprvé je vhodné vypnout Lumen, který zajišťuje globální osvětlení, protože může způsobovat nežádoucí odrazy a nesprávné osvětlení uvnitř portálu. Dále je potřeba deaktivovat Global Illumination, aby se předešlo zbytečnému zatížení výkonu a nekonzistentním světelným efektům. Nakonec doporučuji vypnout Anti-Aliasing, čímž se eliminují možné vizuální artefakty na hranách portálu.

Výsledkem všech těchto kroků je portál, který vypadá jako skutečný průchod do jiného prostoru. Hráč při pohledu na portál vidí věrnou simulaci toho, co by se nacházelo na druhé straně, a při průchodu portálem je iluze zcela plynulá a bez vizuálních rušivých elementů. Tento postup zajišťuje, že portály nejen vypadají realisticky, ale také umožňují hráči bezproblémový přechod mezi různými částmi herního světa.



Pro teleportaci mezi portály použijeme skalární součin. Díky němu jsme schopni zjistit, na které straně portálu se hráč nachází, čímž se předejde nežádané teleportaci, pokud hráč projde stranou portálu, ze které není plátno viditelné.



Nakonec použijeme materiál pro plátno portálu, který vizuálně "prohne" plátno. Tento efekt je důležitý, protože při přímém pohledu na plátno během průchodu portálem by hráč mohl vidět druhou stranu, kterou právě prochází. Prohnutí plátna zakryje tuto nežádoucí scénu a zajistí, že hráč bude mít dojem plynulého přechodu do jiného prostoru. Tímto způsobem se udrží iluze, že portál je přímým průchodem mezi místy.

VYUŽITÉ TECHNOLOGIE

Při implementaci portálového systému a dalších funkcionalit jsem využil dvě hlavní herní platformy: Unity a Unreal Engine 5. Každá z těchto technologií byla vybrána na základě svých silných stránek a schopností splnit specifické požadavky projektu.

V Unity jsem pracoval na mechanice perspektivního růstu objektů inspirované hrou Superliminal. Klíčovými nástroji zde byly skriptovací možnosti Unity (C#) a využití komponenty Camera k získání hráčovy perspektivy. Pro výpočet velikosti objektů podle perspektivy jsem implementoval systém, který dynamicky upravuje měřítko objektu na základě vzdálenosti mezi hráčem a objektem. Toto řešení vyžadovalo precizní ladění, aby byla zachována konzistence herního zážitku. Výběr Unity byl motivován jeho jednoduchým a rychlým workflow pro prototypování a ladění specifických mechanik.

V Unreal Engine 5 jsem se zaměřil na vytvoření portálového systému. Základem bylo využití technologie Screen Capture Component 2D, která umožňuje zachytávat obraz scény v reálném čase a promítat jej na povrch portálu pomocí Render Target textury. Klíčovou součástí bylo vytvoření materiálu, který zajišťuje vizuální efekt průhledného portálu a zabraňuje nežádoucím vizuálním artefaktům, například při pohledu na zadní stranu portálu během průchodu.

Pro zajištění optimálního výkonu a vizuální kvality jsem upravil několik grafických nastavení Unreal Engine, včetně deaktivace Lumen a Anti-Aliasing, které mohou způsobovat nekonzistentní osvětlení nebo rušivé efekty na okrajích portálů.

Výběr těchto technologií byl založen na jejich schopnosti řešit specifické technické výzvy projektu. Unity mi umožnilo rychle implementovat a otestovat unikátní herní mechaniky, zatímco Unreal Engine poskytl nástroje pro tvorbu realistických vizuálních efektů a plynulého portálového přechodu. Obě platformy hrály klíčovou roli při realizaci projektu.

ZÁVĚR

Tento projekt ukazuje možnosti moderních herních enginů při vytváření pokročilých herních mechanik a vizuálních efektů. Kombinace práce v Unity a Unreal Engine mi umožnila experimentovat s různými přístupy k řešení technických výzev a vytvořit funkční systémy, které přispívají k pohlcujícímu hernímu zážitku.

V Unity jsem se zaměřil na implementaci perspektivního růstu objektů, což byla zajímavá výzva v oblasti přesnosti výpočtů a plynulosti mechaniky. Unreal Engine mi naopak poskytl prostředky pro vytvoření realistického portálového systému, včetně dynamických vizuálních efektů a plynulého přechodu mezi prostory.

Během realizace projektu jsem se naučil efektivně pracovat s oběma platformami, porozumět jejich silným a slabým stránkám a integrovat jejich technologie do funkčního celku. Projekt mě také naučil, jak důležité je detailní plánování, testování a iterace při vývoji komplexních herních systémů.

Výsledkem je hra, která nejenže demonstruje různé herní mechaniky, ale také ilustruje, jak mohou technologie přispět k vytvoření inovativních herních zážitků. Tento projekt je nejen ukázkou technického know-how, ale také zdrojem inspirace pro budoucí vývoj her a experimentování s herními enginy.

SEZNAM POUŽITÝCH INFORMAČNÍCH ZDROJŮ

[1] RumpledCode - Superliminal core scaling mechanic - Unity Tutorial

https://www.youtube.com/watch?v=BNYy6O_Uy2Y

- použit pro pochopení základu mechaniky forced perspective

[2] Sebastian Lague - Coding Adventure: Portals

<https://www.youtube.com/watch?v=cWpFZbjtSQg>

- použit pro pochopení funkce neviditelných portálů