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

Rok: 2024/2025

# 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.

[**Abstrakt 1**](#_mxl7t0qnf9nu)

[**Úvod 3**](#_yed9b9s5eh4y)

[Zdůvodnění volby tématu 3](#_5ltcariybkdm)

[Cíl práce 3](#_ssuz17iwoq8v)

[Stručný obsah práce 3](#_pzw41p9dhgh2)

[**Teoretická a metodická východiska 4**](#_pllxgs3q0euj)

[Pohyb hráče v Unity 4](#_to6qsn4jnhpq)

[PlayerMovement.cs - Pohyb hráče 4](#_vvjvy7jvsvxx)

[MouseLook.cs - Pohled Hráče 5](#_o1ngfaeujqqx)

[Forced perspective 6](#_x16nqxph4l1)

[PickupObjects.cs 6](#_dif87tqiuqpr)

[Pohyb hráče v Unreal Engine 16](#_tylirpdamcy)

# 

# 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.

## Zdůvodnění volby tématu

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.

## Cíl 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.

## Stručný obsah práce

Kapitola 1:

se věnuje teoretickým a metodickým východiskům, která objasňují základní principy použitých herních mechanik a postupů.

Kapitola 2:

popisuje technologie využité při implementaci, konkrétně enginy Unity a Unreal.

Kapitola 3:

se zaměřuje na způsoby řešení a detailně popisuje postupy použité při vývoji mechanik.

Kapitola 4:

prezentuje výsledky řešení, zahrnuje dosažené výstupy a obsahuje uživatelský manuál k vytvořeným mechanikám.

Závěr:

shrnuje hlavní poznatky práce a hodnotí dosažené cíle.

# Teoretická a metodická východiska

## Pohyb hráče v Unity

Pohyb hráče je základním prvkem většiny her a jeho správná implementace ovlivňuje celkový zážitek z 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.

### PlayerMovement.cs - Pohyb hráče

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 rychlost 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 rychlost 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); |
| --- |

### MouseLook.cs - Pohled Hráče

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); |
| --- |

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.

## Forced perspective

### PickupObjects.cs

Tento skript umožňuje hráči zvedat, přesouvat a měnit měřítko objektů na základě perspektivy v Unity. Funguje v kombinaci s kamerou a zajišťuje, že objekt je před překážkami a má odpovídající velikost vzhledem ke vzdálenosti od kamery.

#### Základní princip fungování

##### Raycasting

* Kamera hráče neustále „vysílá paprsek“ (Raycast) ve směru, kterým hráč hledí.
* Pokud paprsek narazí na objekt, který lze uchopit, skript umožní tento objekt vybrat a držet.

##### Připojení objektu ke kameře

* Po uchopení je objekt umístěn jako „potomek“ kamery, což znamená, že se pohybuje společně s hráčem a zůstává před ním ve fixní vzdálenosti.

#### Přizpůsobení pozice objektu před překážkami

* Při pohybu objektu skript kontroluje, zda objekt nenarazí na překážku (např. stěnu).
* Pomocí Raycastingu a mřížky bodů (grid) se detekuje nejbližší volná pozice, aby se objekt nepřekrýval s kolizemi.

##### Změna měřítka objektu na základě perspektivy

* Velikost objektu je dynamicky přepočítána na základě jeho vzdálenosti od kamery.
* Čím je objekt dál od hráče, tím je menší; čím blíže, tím je větší.
* Tento efekt se dosahuje přímým přepočtem měřítka objektu a jeho pozice ve světovém prostoru.

##### Uvolnění objektu

* Po uvolnění objekt ztrácí své propojení s kamerou a znovu se zapne jeho fyzikální simulace (gravitace a kolize).

##### Update()

Sleduje, zda hráč klikne myší na objekt (levé tlačítko) a volá metodu pro zvednutí objektu.

Pokud je objekt zvednutý:

* Aktualizuje jeho polohu, aby se nacházel před překážkami (MoveInFrontOfObstacles).
* Dynamicky mění měřítko objektu podle vzdálenosti od hráče/kamery (UpdateScale).

##### PickUpObject()

Zvedne objekt před hráčem, pokud je ve vrstvě určené pro zvedání.

Pokud je objekt již zvednutý:

* Položí ho zpět a obnoví fyzikální vlastnosti.

Pokud ne:

* Připojí objekt k hráči/kameře.
* Zamkne jeho fyziku (zmrazí Rigidbody).
* Nastaví základní měřítko a pozici objektu pro další úpravy.
* Vytvoří mřížku pro detekci překážek (SetupShapedGrid).

| void PickUpObject() {  // Pokud už držíme objekt, pusť ho  if (heldObject != null)  {  heldObjectTF.parent = null;  heldObjectRB.useGravity = true;  heldObjectRB.constraints = RigidbodyConstraints.None;  heldObject.layer = (int)Mathf.Log(pickUpLayer.value, 2);  heldObject = null;  return;  }   // Raycast pro detekci objektu před hráčem  RaycastHit hit;  if (Physics.Raycast(transform.position, transform.forward, out hit, 100))  {  if ((pickUpLayer.value & (1 << hit.collider.gameObject.layer)) == 0)  {  return; // Objekt není ve vrstvě pro zvedání  }  }  else  {  return;  }   // Přiřadíme objekt a jeho komponenty  heldObject = hit.collider.gameObject;  heldObjectTF = heldObject.transform;  heldObjectRB = heldObject.GetComponent<Rigidbody>();   // Nastavíme objekt k hráči/kameře  heldObject.layer = (int)Mathf.Log(heldObjectLayer.value, 2);  heldObjectRB.useGravity = false;  heldObjectTF.parent = transform;  heldObjectRB.constraints = RigidbodyConstraints.FreezeAll; } |
| --- |

##### MoveInFrontOfObstacles()

* Posune objekt tak, aby byl před nejbližší překážkou v prostoru.
* Používá mřížku bodů, ze které vrhá paprsky k detekci kolizí.

| 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; } |
| --- |

#### 

#### 

##### UpdateScale()

* Dynamicky mění velikost objektu v závislosti na vzdálenosti od hráče.
* Přepočítává pozici objektu, aby odpovídal změně měřítka a perspektivy.

| void UpdateScale() {  float newScale = (transform.position - heldObjectTF.position).magnitude / orgDistanceToScaleRatio;   // Pokud je změna měřítka příliš malá, neprováděj žádnou akci  if (Mathf.Abs(newScale - heldObjectTF.localScale.x) < SCALE\_MARGIN) return;   // Aktualizace měřítka  heldObjectTF.localScale = new Vector3(newScale, newScale, newScale);   // Udržení objektu na správném místě ve viewportu  Vector3 newPos = Camera.main.ViewportToWorldPoint(new Vector3(orgViewportPos.x, orgViewportPos.y,   (heldObjectTF.position - transform.position).magnitude));  heldObjectTF.position = newPos; } |
| --- |

##### SetupShapedGrid()

* Nastaví mřížku bodů kolem objektu.
* Využívá bounding box objektu k vytvoření mřížky pro detekci překážek.

| private void SetupShapedGrid(Vector3[] bbPoints) {  // Vypočítáme hranice objektu  left = right = top = bottom = Vector2.zero;  GetRectConfines(bbPoints);   // Vytvoříme mřížku bodů  Vector3[,] grid = SetupGrid();   // Detekujeme překážky pro body mřížky  GetShapedGrid(grid); } |
| --- |

##### CastTowardsGridPoint()

* Vrhá paprsky z kamery směrem k bodům mřížky.
* Detekuje kolize a určuje, které body jsou blokovány překážkami.

| private RaycastHit CastTowardsGridPoint(Vector3 gridPoint, LayerMask layers) {  Vector3 worldPoint = transform.TransformPoint(gridPoint);  Vector3 origin = Camera.main.WorldToViewportPoint(worldPoint);  origin.z = 0;  origin = Camera.main.ViewportToWorldPoint(origin);   // Směr paprsku  Vector3 direction = worldPoint - origin;   // Vrhání paprsku  RaycastHit hit;  Physics.Raycast(origin, direction, out hit, 1000, layers);   return hit; } |
| --- |

#### 

#### 

##### Pomocné metody

##### GetBoundingBoxPoints()

* Získává rohové body bounding boxu objektu.

| 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; } |
| --- |

##### 

##### GetRectConfines()

* Vypočítává ohraničení objektu ve viewportu kamery.

| private void GetRectConfines(Vector3[] bbPoints) {  Vector3 bbPoint;  Vector3 cameraPoint;  Vector2 viewportPoint;   // Najdeme nejbližší bod objektu vůči kameře  Vector3 closestPoint = heldObject.GetComponent<Renderer>().localBounds.ClosestPoint(transform.position);  float closestZ = transform.InverseTransformPoint(heldObjectTF.TransformPoint(closestPoint)).z;   // Zkontrolujeme, že bod neleží za hráčem  if (closestZ <= 0) throw new System.Exception("HeldObject's inside the player!");   // Procházíme všechny body bounding boxu  for (int i = 0; i < bbPoints.Length; i++)  {  bbPoint = heldObjectTF.TransformPoint(bbPoints[i]);  viewportPoint = Camera.main.WorldToViewportPoint(bbPoint);  cameraPoint = transform.InverseTransformPoint(bbPoint);  cameraPoint.z = closestZ;   // Kontrolujeme, zda bod leží uvnitř viewportu  if (viewportPoint.x < 0 || viewportPoint.x > 1 || viewportPoint.y < 0 || viewportPoint.y > 1) continue;   // Inicializace hraničních bodů  if (i == 0) left = right = top = bottom = cameraPoint;   // Aktualizace hranic  if (cameraPoint.x < left.x) left = cameraPoint;  if (cameraPoint.x > right.x) right = cameraPoint;  if (cameraPoint.y > top.y) top = cameraPoint;  if (cameraPoint.y < bottom.y) bottom = cameraPoint;  } } |
| --- |

##### SetupGrid()

* Vytváří rovnoměrnou mřížku bodů.

| 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; } |
| --- |

##### GetShapedGrid()

* Zpracovává body mřížky, které nejsou blokovány kolizemi.

| private void GetShapedGrid(Vector3[,] grid) {  shapedGrid.Clear(); // Vyčistíme seznam bodů   // Procházíme každý bod mřížky  foreach (Vector3 point in grid)  {  RaycastHit hit = CastTowardsGridPoint(point, heldObjectLayer);   // Přidáme bod do seznamu, pokud nedošlo ke kolizi  if (hit.collider != null)  {  shapedGrid.Add(point);  }  } } |
| --- |

#### Shrnutí klíčových konceptů:

* Raycasting: Pro detekci překážek mezi hráčem a objektem.
* Bounding Box: Slouží k určení okrajů objektu a vytvoření mřížky bodů.
* Viewport a perspektiva: Pozice objektu se upravuje podle kamery a viewportu.
* Fyzika (Rigidbody): Fyzika objektu je vypnuta během držení (zmrazení pozice a gravitace).
* Dynamické měřítko: Velikost objektu se mění podle vzdálenosti od hráče.

## Pohyb hráče v Unreal Engine