FAKULTA MECHATRONIKY,
INFORMATIKY A MEZIOBOROVÝCH
STUDIÍ TUL

Bakalářská práce

Využití shaderů a procedurálního generování grafiky ve 2D pro herní engine Unity

Studijní program: B0613A140005 Informační technologie
Studijní obor: Aplikovaná informatika

Autor práce: **Mgr. Dominika Strečanská**
Vedoucí práce: Ing. Lukáš Zedek, Ph.D.
Ústav nových technologií a aplikované informatiky

Konzultant práce: doc. Mgr. Jan Březina, Ph.D.
Ústav nových technologií a aplikované informatiky

Liberec 2025

FAKULTA MECHATRONIKY,
INFORMATIKY A MEZIOBOROVÝCH
STUDIÍ TUL

Zadání bakalářské práce

Využití shaderů a procedurálního generování grafiky ve 2D pro herní engine Unity

Jméno a příjmení:	**Mgr. Dominika Strečanská**
Osobní číslo:	M22000186
Studijní program:	B0613A140005 Informační technologie
Specializace:	Aplikovaná informatika
Zadávající katedra:	Ústav nových technologií a aplikované informatiky
Akademický rok:	2024/2025

Zásady pro vypracování:

1. Proveďte rešerši technik tvorby 2D grafiky pro počítačové hry a nástrojů pro procedurální generování grafiky s možností editace zdrojového kódu.
2. Vytvořte grafiku ve formě UI elementů, postav, textur a pozadí pomocí zvolených technik a nástrojů.
3. Vytvořte shadery za pomocí vytvořené grafiky v herním enginu Unity.
4. Výsledné shadery a grafiku otestuje ve vybrané hře.
5. Vypracujte zprávu s analýzou a porovnáním metod z hlediska náročnosti a efektivity, doplněné vybranými ukázkami grafických výstupů a zdrojového kódu.

Rozsah grafických prací:	dle potřeby dokumentace
Rozsah pracovní zprávy:	30 – 40 stran
Forma zpracování práce:	tištěná/elektronická
Jazyk práce:	čeština

Seznam odborné literatury:

[1] BJÖRK, Staffan a HOLOPAINEN, Jussi. *Patterns in game design*. Hingham: Charles River Media, 2005. ISBN 1584503548.

[2] WATKINS, Ryan. *Procedural content generation for unity game development: harness the power of procedural content generation to design unique games with unity*. *Community experience distilled*. Birmingham: Packt Publishing, 2016. ISBN 978-1-78528-747-3.

[3] SILBER, Daniel. *Pixel art for Game Developers*. CRC Press, 2017. ISBN 978-1-138-41355-9.

[4] SOLARSKI, Chris. *Interactive Stories and Video Game Art*. CRC Press, 2017. ISBN 9781498781503.

Vedoucí práce:	Ing. Lukáš Zedek, Ph.D. Ústav nových technologií a aplikované informatiky
Konzultant práce:	doc. Mgr. Jan Březina, Ph.D. Ústav nových technologií a aplikované informatiky
Datum zadání práce:	15. října 2024
Předpokládaný termín odevzdání:	9. května 2025

L.S.

doc. Ing. Josef Černohorský, Ph.D.
děkan

doc. Ing. Josef Chaloupka, Ph.D.
garant studijního programu

V Liberci dne 18. října 2024

Prohlášení

Prohlašuji, že svou bakalářskou práci jsem vypracovala samostatně jako původní dílo s použitím uvedené literatury a na základě konzultací s vedoucím mé bakalářské práce a konzultantem.

Jsem si vědoma toho, že na mou bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci nezasahuje do mých autorských práv užitím mé bakalářské práce pro vnitřní potřebu Technické univerzity v Liberci.

Užiji-li bakalářskou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědoma povinnosti informovat o této skutečnosti Technickou univerzitu v Liberci; v tomto případě má Technická univerzita v Liberci právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Současně čestně prohlašuji, že text elektronické podoby práce vložený do IS/STAG se shoduje s textem tištěné podoby práce.

Beru na vědomí, že má bakalářská práce bude zveřejněna Technickou univerzitou v Liberci v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů.

Jsem si vědoma následků, které podle zákona o vysokých školách mohou vyplývat z porušení tohoto prohlášení.

4. května 2025 Mgr. Dominika Strečanská

Využití shaderů a procedurálního generování grafiky ve 2D pro herní engine Unity

Abstrakt

Tato bakalářská práce se zabývá využitím shaderů a procedurálního generování v prostředí vývoje 2D počítačových her s důrazem na praktickou implementaci v herním enginu Unity. Cílem práce je prozkoumat možnosti, které tyto technologie nabízejí pro efektivní tvorbu vizuálně atraktivního a dynamického herního obsahu, a demonstrovat jejich využití na konkrétním příkladu.
V teoretické části je představena problematika tvorby 2D grafiky, základní principy shaderů, možnosti jejich nasazení v Unity, a techniky procedurálního generování vizuálních prvků. Dále jsou popsány různé přístupy k tvorbě grafických assetů, včetně ruční kresby, algoritmického generování a využití nástrojů založených na umělé inteligenci.
Praktická část je zaměřena na vývoj vizuálních efektů pomocí shaderů, včetně dynamických změn prostředí, simulace ročních období a interaktivních grafických prvků. V rámci implementace byl použit hybridní grafický workflow zahrnující Adobe Illustrator, Python, Material Maker a generativní AI. Výsledkem je sada shaderů a grafických komponent, které lze nasadit v rámci vývoje 2D herního projektu.
Získané výsledky potvrzují, že využití shaderů a procedurálních technik přináší nejen vizuální přínos, ale i optimalizaci výkonu a paměťové náročnosti. Práce ukazuje potenciál těchto nástrojů pro vývojáře, kteří chtějí dosáhnout vysoké úrovně grafického zpracování bez nadměrného zatížení produkčního procesu.

Klíčová slova: shadery, procedurální generování, 2D grafika, Unity, vizuální efekty

Use of shaders and procedural graphic generation in 2D for the Unity game engine

Abstract

This bachelor's thesis deals with the use of shaders and procedural generation in the context of 2D computer game development, with an emphasis on practical implementation in the Unity game engine. The aim of the thesis is to explore the possibilities these technologies offer for the efficient creation of visually attractive and dynamic game content and to demonstrate their use through a concrete example.
The theoretical part introduces the issues of creating 2D graphics, the basic principles of shaders, their potential applications in Unity, and techniques for procedurally generating visual elements. It also describes various approaches to creating graphical assets, including hand drawing, algorithmic generation, and the use of AI-based tools.
The practical part focuses on the development of visual effects using shaders, including dynamic environmental changes, seasonal simulations, and interactive graphical elements. A hybrid graphical workflow was used during the implementation, incorporating Adobe Illustrator, Python, Material Maker, and generative AI. The result is a set of shaders and graphical components that can be deployed within a 2D game development project.
The findings confirm that the use of shaders and procedural techniques brings not only visual benefits but also performance and memory optimization. The thesis demonstrates the potential of these tools for developers aiming to achieve a high level of graphical quality without excessive strain on the production process.

Keywords: shaders, procedural generation, 2D graphics, Unity, visual effects

Poděkování

Ráda bych vyjádřila své upřímné poděkování panu Ing. Lukáši Zedkovi, Ph.D., za jeho vstřícný a odborný přístup, pravidelné konzultace a trpělivost, kterou mi věnoval po celý akademický rok. Velmi si vážím jeho aktivity, ochoty poskytovat zpětnou vazbu a především podnětů, které mi pomohly posouvat tuto práci dál a vnímat ji z nových perspektiv. Mé poděkování patří také Janu Smutnému za jeho neocenitelnou podporu, trpělivost a ochotu sdílet zkušenosti. Díky jeho nápadu na spolupráci při tvorbě 2D hry jsem měla možnost podílet se na tématu, které je mi blízké, a vytvořit práci, se kterou jsem spokojená. Jeho důvěra a pomoc pro mě byla velkou motivací. Dále bych chtěla poděkovat všem, kteří mi během zpracování této práce poskytli zpětnou vazbu, podněty nebo povzbuzení. Každý takový moment pro mě znamenal možnost posunout se dál, zamyslet se nad kvalitou své tvorby a zlepšit nejen výsledek, ale i vlastní přístup.

Obsah

Seznam zkratek

CRT	katodová obrazovka používaná ve starších monitorech (Cathode Ray Tube)
GLSL	jazyk pro psaní shaderů v rámci OpenGL (OpenGL Shading Language)
GPU	grafická karta (Graphics Processing Unit)
LOD	technika pro snižování detailů (Level Of Detail)
PBR	realistické vykreslování na základě fyzikálních principů (Physically Based Rendering)
PNG	formát rastrových obrázků s bezstrátovou kompresí (Portable Network Graphics)
UI	uživatelské rozhraní (User Interface)
URP	renderovací pipeline (Universal Render Pipeline)
WFC	algoritmus na procedurální generování vzorů (Wave Function Collapse)

Seznam pojmů

Asset Digitální obsah použitý ve hře, například textury, modely, zvuky nebo skripty.

Glitch efekt Vizuální efekt napodobující grafické chyby známé z analogových nebo digitálních zařízení, jako jsou CRT monitory, televizory nebo starší herní konzole.

Rigging Proces vytváření kostry (skeletu) pro 2D nebo 3D model, která umožňuje jeho animaci.

Shader Program určený pro běh na grafické kartě, který ovlivňuje způsob vykreslování obrazu. Umožňuje určovat vlastnosti povrchu objektů, světla, stíny, barvy nebo vizuální efekty.

Shader Graph Vizualizační nástroj Unity pro tvorbu shaderů pomocí funkčních bloků.

ShaderLab Speciální jazyk Unity pro popis struktury shaderu, jeho vlastností, průchodů a propojení s materiály.

Sprite Dvojrozměrný grafický objekt, který reprezentuje postavy, objekty nebo efekty ve hře. V 2D hrách je sprite označován za základní jednotku vizuální reprezentace.

Textura Dvourozměrný obraz sloužící k definici vizuálních vlastností povrchu objektu, jako jsou barva, vzor nebo struktura.

Tilemap Systém dlaždic pro vytváření 2D herních map.

Úběžník Bod v perspektivní projekci, do kterého se sbíhají rovnoběžné linie.

1 Úvod

V posledních letech se oblast vývoje počítačových her dynamicky rozvíjí, a to nejen po stránce technologické, ale i vizuální. Požadavky hráčů i vývojářů se neustále zvyšují, a proto kromě plynulého chodu a hratelnosti hraje stále důležitější roli také vizuální zpracování. Estetická stránka herního prostředí zásadně ovlivňuje celkový uživatelský zážitek a může rozhodovat o úspěchu nebo neúspěchu herního titulu. Na tuto skutečnost reagují i nástroje a techniky, které vývojářům umožňují vytvářet výrazné vizuální efekty a stylizované herní světy.

Jednou z oblastí, která v tomto kontextu nabývá na významu, jsou shadery a procedurální generování grafiky. Shadery jsou malé programy běžící na grafické kartě, které určují, jak budou objekty ve scéně vizuálně zobrazeny – například jejich barvu, osvětlení nebo texturu. Tyto technologie umožňují nejen dosažení atraktivního vizuálního výstupu, ale také optimalizované využití systémových prostředků, opakovatelnou tvorbu obsahu a snadnou úpravu pomocí parametrizace. Zatímco shadery slouží především ke zpracování vizuálních efektů na úrovni renderovacího procesu, procedurální přístupy umožňují generovat grafické prvky algoritmicky, často bez nutnosti manuálního kreslení.

Motivací k výběru tohoto tématu byl dlouhodobý zájem o propojení herního designu s grafickou tvorbou, a to především v oblasti, kde se technická stránka programování setkává s vizuální kreativitou. Práce se shadery a generováním grafiky mi umožnila tuto kombinaci prakticky rozvíjet a prohloubit si v ní znalosti.

Tato bakalářská práce se zaměřuje na využití výše zmíněných technik v prostředí 2D her, konkrétně s využitím herního enginu Unity. Teoretická část se soustředí na zmapování metod tvorby 2D grafiky v herním vývoji a nástrojů, které umožňují procedurální generování s možností zásahu do zdrojového kódu. Na základě této rešerše byly zvoleny konkrétní postupy a nástroje, které umožnily navrhnout a vytvořit grafické prvky zahrnující uživatelské rozhraní, postavy, textury i herní pozadí.

Praktická část práce byla realizována v rámci spolupráce na vývoji konkrétní 2D hry, jejíž návrh a herní mechaniky jsou detailně popsány v bakalářské práci Jana Smutného [46]. V rámci této práce byla vytvořena grafická stránka hry, včetně shaderů a procedurálně generovaných komponent. Implementované prvky byly otestovány v reálném herním prostředí, což umožnilo vyhodnotit jejich vizuální efekt, náročnost z hlediska vývoje i dopad na výkon. Výsledná hra je k dispozici ke stáhnutí na GitHubu: https://github.com/SmutnyJan/BP-Unity-Smutny/releases/tag/1.2.

Výsledkem práce je sada grafických výstupů, shaderů a procedurálních komponent, doplněná analýzou a porovnáním použitých metod, včetně praktických ukázek zdrojového kódu a výsledných obrázků. Mimo aplikování všech výstupů na zvole-

nou hru byl navíc vytvořen samostatný projekt v prostředí Unity, který slouží jako interaktivní galerie dosažených výstupů této práce.

2 Teoretické východiska pro tvorbu 2D grafiky v herních enginech

2.1 Úvod do problematiky 2D grafiky v herních enginech

Grafika je důležitou součástí herních zážitků a ve vývoji her zaujímá klíčové místo. I když se dnešní hry často zaměřují na realistickou 3D grafiku, 2D grafika si stále zachovává svůj význam. Její jednoduchost, estetická hodnota a široká škála kreativních možností z ní činí neodmyslitelnou součást vývoje her [29].

2D grafiku ve videohrách můžeme definovat jako vizuální prvky na dvourozměrné ploše. Tyto prvky, jako jsou postavy, pozadí, objekty či uživatelská rozhraní, postrádají hloubku, která je charakteristická pro 3D modely. V hrách s 2D grafikou jsou postavy a prostředí prezentovány jako ploché obrazy, které nazýváme sprite [1], přičemž kamera zpravidla používá ortografickou projekci [9] (podrobněji v následující podkapitole), která eliminuje perspektivu.

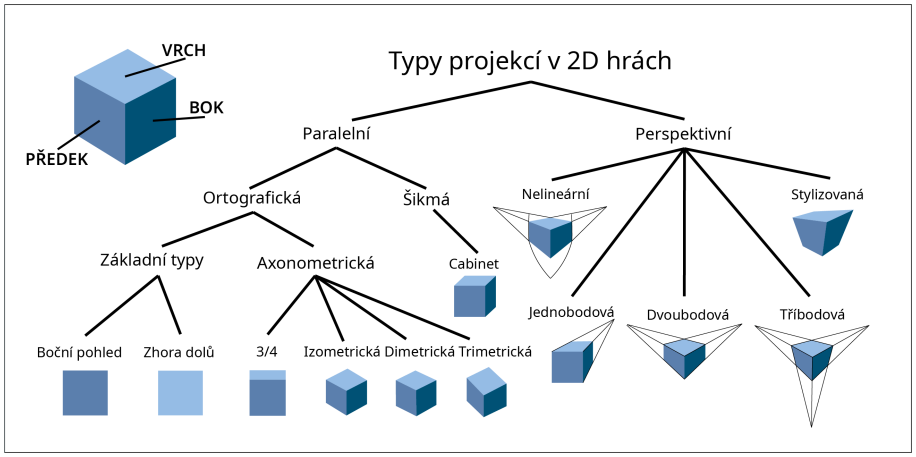
Vytváření a správné zobrazení těchto grafických prvků však vyžaduje specializované nástroje. Herní enginy představují základní platformu pro vývoj videoher a umožňují efektivní práci s grafikou i ostatními herními prvky. V případě 2D her nabízejí nástroje pro práci se sprity, animacemi, vrstvami a herními mechanismy.

Některé populární herní enginy, jako Unity nebo Godot, podporují 2D i 3D grafiku a poskytují vývojářům specializované nástroje pro optimalizaci výkonu, správu paměti a interakci s fyzikálními systémy. Tyto systémy se týkají nejen simulace fyziky uvnitř herního prostředí (např. kolize, gravitace), ale mohou zahrnovat i interakce s fyzickými prvky mimo samotnou hru, jako jsou vibrace gamepadu či další haptická zpětná vazba, která zvyšuje herní zážitek.

2.1.1 Typy projekcí v 2D hrách

Vizuální podoba 2D her je výrazně ovlivněna typem použité projekce – tedy způsobem, jakým jsou trojrozměrné objekty nebo scény převedeny na dvourozměrnou plochu obrazovky. Projekce určují, jak se budou objekty zobrazovat vzhledem k pozici kamery, úběžníkům a perspektivě [28].

Existují různé typy projekcí, které se ve hrách používají. Strukturu a rozdíly mezi jednotlivými typy projekcí lze vidět na Obrázku 2.1.



Obrázek 2.1: Typy projekcí v 2D hrách. Převzato a upraveno podle Majewski(2022) [28]

- Paralelní projekce, někdy označovaná i jako souběžná projekce, je charakteristická tím, že rovnoběžné linie v prostoru zůstávají rovnoběžné i po promítnutí na obrazovku. Dělí se na na několik specifických kategorií, z nichž nejpoužívanější v kontextu 2D her jsou:
  - Ortografická projekce je technika používaná ve 2D hrách k zobrazování objektů bez perspektivního zkreslení. Při této projekci jsou všechny body zobrazeny na ploše tak, že jejich souřadnice zůstávají v poměru zachovány a nejsou ovlivněny vzdáleností od kamery. Díky tomu se velikost objektů nemění ani tehdy, když se nacházejí blíže nebo dále od kamery – jejich zobrazení zůstává stejné bez ohledu na prostorovou hloubku. Tento přístup usnadňuje orientaci hráčů a eliminuje potřebu škálování již vytvořených spritů. Ortografická projekce je v 2D hrách široce rozšířená právě díky své předvídatelnosti a technické nenáročnosti.
    - Základní typy ortografické projekce:
      - Side view (boční pohled): Kamera se dívá na herní svět z boku a často se pohybuje horizontálně spolu s postavou. Tento typ projekce je typický například pro plošinové hry. I když v ortografické projekci velikost objektů není ovlivněna vzdáleností od kamery, ve hrách se někdy pozadí záměrně škáluje nebo posouvá jinou rychlostí než popředí, aby se vytvořil vizuální dojem vrstveného prostředí a pohybu v prostoru.
      - Top-down (shora dolů): Kamera je umístěna nad herním světem a dívá se přímo dolů, přičemž objekty jsou zobrazeny z pohledu shora. Tento typ je často využíván ve strategických hrách a simulátorech. I když tato perspektiva poskytuje dobrý přehled o herním prostředí, není ideální pro zobrazení lidských postav kvůli zkreslení proporcí a omezené viditelnosti detailů.
    - Axonometrická projekce:

        - Izometrická projekce: Všechny tři osy svírají úhel 120 stupňů, což vytváří vizuálně vyvážené zobrazení světa, kde lze objekty vidět z více stran najednou.
        - Dimetrická projekce: Dvě osy svírají se svislou osou stejný úhel, zatímco třetí osa má jiný sklon. Tento typ je běžný v hrách s pixel art grafikou, protože umožňuje mírné perspektivní zkreslení při zachovávání jednoduchého vykreslování.
        - Trimetrická projekce: Každá osa svírá s horizontem jiný úhel a má jinou míru zkreslení, což umožňuje realističtější zobrazení objektů díky přesnějším proporcím a lepšímu rozložení perspektivy. Tento přístup poskytuje větší kontrolu nad vizuální prezentací scén, ale je náročnější na správné vykreslení.
        - ¾ projekce: Často označovaná jako pseudo-izometrická projekce. Kamera je obvykle nakloněná pod úhlem mezi 30° a 45°, aby zobrazovala část přední a horní strany objektů.
    - Šikmá projekce:
        - Cabinet projekce: Promítací paprsky jsou nakloněny pod šikmým úhlem, čímž se zdůrazňuje hloubka objektů bez perspektivního zkreslení. Tento typ se využívá například v hrách typu „beat 'em up“, kde se hráč pohybuje ve scéně nejen vodorovně, ale i do hloubky prostředí a bojuje proti protivníkům [51].
- Perspektivní projekce přibližují způsob, jakým lidské oko vnímá prostor. Charakteristickým rysem je, že s rostoucí vzdáleností od kamery se objekty jeví menší, čímž vzniká dojem hloubky. Typy perspektivních projekcí zahrnují:
    - Nelineární projekce: Objekty jsou zakřiveny na základě více úběžníků.
    - Jednobodová, dvoubodová a tříbodová projekce: Počet úběžníků ovlivňuje, jak realisticky bude zobrazen prostor. Perspektivní projekce se často využívají ve hrách s pevně daným úhlem kamery, kde se hráč nemůže volně rozhlížet. Umožňují vytvořit dramatické a filmové kompozice záběru, například s nadhledem nebo podhledem.

2.2 Proces a nástroje pro tvorbu grafiky pro 2D hry

Tvorba 2D herní grafiky sestává z několika fází, které jsou nezbytné k dosažení vizuálně konzistentního a funkčního prostředí ve hře. Každý krok ovlivňuje nejen estetickou stránku, ale také výkon a možnosti optimalizace a úprav grafiky v rámci herního enginu.

1. **Konceptuální design**

   Tato fáze zahrnuje návrh základní vizuální identity hry, včetně stylu, palety barev, postav, prostředí a prvků uživatelského rozhraní (UI – User Interface). Vytváří se série skic a digitálních konceptů, které slouží jako referenční podklady pro další tvorbu [44].

Zásadní součástí této fáze je volba výtvarného stylu, která ovlivňuje nejen atmosféru hry a způsob, jakým hráč vnímá herní svět, ale také technické nároky, cílovou platformu a celý průběh vývoje.

2. **Volba vizuálního stylu 2D grafiky**

   Výběr výtvarného stylu hraje v návrhu 2D herní grafiky zásadní roli. Různé grafické přístupy, jako je pixel art, digitální malba nebo vektorová grafika, mají svá specifika. Například pixel art umožňuje vytvářet vizuálně jednoduché, ale silně stylizované světy s minimálními nároky na výkon, zatímco digitální malba nabízí vysokou míru detailů a výrazné světelné efekty, které podtrhují atmosféru hry, avšak vyžaduje rozsáhlejší produkční zázemí, tedy více času, techniky a zkušeností..

   Některé styly, jako siluetová grafika, využívají kompozice založené na kontrastních tvarech bez vnitřního detailu, obvykle v podobě tmavých obrysů na světlém pozadí. Cutout grafika pracuje s vrstvenými tvary připomínající výstřižky z materiálu, které se skládají do výsledné scény. Flat design je vizuální styl redukovaný na základní barvy a tvary, bez použití prostorových efektů a stínování. Vektorová grafika pracuje s přesně definovanými geometrickými útvary, což umožňuje snadné škálování a čisté zobrazení na různých zařízeních. Díky těmto vlastnostem jsou oba přístupy často využívány v mobilních a logických hrách, kde je kladen důraz na přehlednost a nízkou technickou náročnost.

   Volba stylu tedy závisí nejen na vizuálních preferencích autorů, ale také na technických omezeních cílové platformy, cílové skupině hráčů, a na požadované náladě či tématu hry.

3. **Výběr nástrojů a softwaru**

   K tvorbě 2D grafiky se používají různé softwarové nástroje v závislosti na požadované estetice a technických omezeních:

   - Krita – open-source software určený pro digitální kresbu, vhodný pro vytváření rastrové grafiky a ilustrací [25].
   - GIMP – open-source alternativa k Adobe Photoshop, využívaná pro práci s rastrovou grafikou, včetně úprav textur a fotomontáží [15].
   - Adobe Photoshop – profesionální nástroj používaný k tvorbě detailních spritů, textur a UI prvků [4].

4. **Tvorba herních assetů**

   Na základě konceptů se vytvářejí finální grafické prvky, jako jsou pozadí, postavy, objekty a UI komponenty. Při tvorbě 2D grafiky se často využívá kombinace ručně kreslených prvků a generovaných textur [21]. Procedurální generování grafiky umožňuje automatizovanou tvorbu vizuálně rozmanitých prvků, čímž se šetří čas a snižuje náročnost manuální tvorby. Tato technika je využitelná zejména při generování prostředí, dlaždicových textur či dynamických efektů [43].

5. **Animace**

   Animace je důležitá pro oživení herních prvků. V 2D hrách se využívají různé metody animace:

   - Frame-by-frame animace – tradiční metoda, kde se každý snímek kreslí manuálně [61].
   - Skeletová animace – využívá kostru postavy pro manipulaci s jednotlivými částmi modelu [45].

   Mezi populární software pro animaci patří Adobe Animate [3] a Spine [47], které umožňují efektivní tvorbu animací s využitím kinematiky a interpolace pohybů. Herní enginy, jako Unity, pak poskytují nástroje pro implementaci a řízení animací v reálném čase.

6. **Integrace do herního enginu**

   Po dokončení grafiky je nutné ji správně implementovat do herního enginu, přičemž se řeší rendering, optimalizace a adaptace pro různá rozlišení. Mezi populární enginy pro 2D hry patří:

   - Unity [57] – univerzální engine, který nabízí pokročilou podporu pro zpracování shaderů a optimalizaci vykreslování.
   - Godot [16] – open-source engine, který obsahuje vlastní 2D renderer a podporu pro skriptování v GDScriptu, což je vlastní programovací jazyk Godotu inspirovaný Pythonem, určený pro jednoduchou a efektivní tvorbu herní logiky.
   - GameMaker Studio 2 [14] – engine vhodný pro 2D hry s jednoduchým implementačním workflow.

7. **Využití shaderů v 2D grafice**

   Moderní 2D hry využívají shadery ke zlepšení vizuálních efektů a interakce mezi objekty. Shadery umožňují implementaci efektů, jako jsou dynamické osvětlení, zobrazení povrchu objektů pomocí textur připomínajících materiály, rozmazání pohybu či post-processingové úpravy obrazu. Například v Unity se vytváří shadery v jazyce HLSL (High-Level Shader Language) [19] nebo Shader Graph, což umožňuje procedurální generování textur a složitější vizuální efekty [5] [41].

8. **Testování a optimalizace**

   Každá grafika musí být optimalizována pro efektivní využití výkonu zařízení. V 2D hrách se optimalizace zaměřuje na:

   - Redukci velikosti textur a jejich efektivní ukládání pomocí sprite atlasů.

- Snížení počtu draw calls, tedy jednotlivých požadavků na vykreslení objektů, které herní engine posílá grafické kartě. Každý takový požadavek představuje určitou výpočetní zátěž a jejich optimalizací lze výrazně zlepšit výkon hry.
- Použití techniky úrovně detailu, označované také jako Level of Detail (LOD), umožňuje vykreslování různých verzí spritů podle rozlišení obrazovky. Při vyšším rozlišení se zobrazují detailnější varianty, zatímco u nižších postačují jednodušší verze s menšími požadavky na výkon [17].
- Implementaci procedurálních efektů prostřednictvím shaderů namísto předem připravených animací, čímž lze snížit nároky na paměť.

2.3 Nástroje pro procedurální generování 2D grafiky s možností úpravy zdrojového kódu

Procedurální generování využívá matematické funkce a algoritmy k tvorbě komplexních struktur a vzorů, přičemž vychází z jednoduchých vstupních parametrů. Tento přístup je založen na několika klíčových principech.

Deterministický charakter algoritmů způsobuje, že při stejných vstupních parametrech vždy vznikne stejný výstup, což umožňuje přesnou reprodukovatelnost [11].

Na druhou stranu procedurální generování často pracuje také s pseudo-náhodností, která zajišťuje variabilitu výsledků, přičemž stále dodržuje stanovená pravidla a omezení [11]. Významnou roli při tvorbě procedurálních vzorů hrají i fraktální struktury a rekurzivní algoritmy, které umožňují generování složitých, vizuálně přirozených obrazců.

2.3.1 Výhody a nevýhody procedurálního generování

Procedurální generování nabízí mnoho výhod, díky nimž je široce využívané v oblasti tvorby počítačové grafiky a herního vývoje. Jednou z jeho největších předností je automatizace, která umožňuje vytváření obsahu bez nutnosti manuálního zásahu, čímž se výrazně šetří čas i výpočetní zdroje.

Dalším benefitem je vysoká variabilita generovaného obsahu, jelikož na základě různých vstupních parametrů může algoritmus generovat unikátní výstupy, což zajišťuje rozmanitost a v případě her i vyšší znovuhratelnost. Kromě toho procedurální generování výrazně přispívá k úspoře paměti – místo ukládání velkých objemů dat se uchovává pouze samotný algoritmus spolu s jeho parametry, což snižuje nároky na úložiště a umožňuje efektivnější zpracování dat [43].Tyto výhody jsou využívány například ve hře Terraria, která při každém spuštění generuje unikátní svět včetně biomů, jeskyní a struktur na základě algoritmu a jeho parametrů [53].

Navzdory těmto výhodám však tento přístup přináší i určité nevýhody. Jednou z nich je složitost ladění, protože u komplikovanějších algoritmů může být předvídání výsledného výstupu obtížné, což ztěžuje opravu chyb a úpravy systému [43].

Dalším problémem může být omezená kontrola nad finálním výstupem – i když algoritmus generuje obsah podle zadaných pravidel, ne vždy je možné dosáhnout přesně požadovaného designu, což může být nevýhodou u projektů, kde je důležitá vysoká míra detailnosti a umělecká kontrola.

2.3.2 Nástroje pro generování 2D grafiky

Existuje široká škála nástrojů, které umožňují automatizovanou tvorbu vizuálního obsahu na základě algoritmických přístupů. Tyto nástroje nabízejí různé funkcionality a jsou vhodné pro odlišné aspekty grafické tvorby – od generování textur až po vytváření komplexních herních prostředí. Následující softwarová řešení představují vybrané nástroje, které se běžně využívají při procedurální tvorbě grafiky:

Substance 3D Designer

Substance Designer [52] je profesionální nástroj primárně určený pro procedurální generování 2D textur a materiálů. Využívá uzlový (node-based) editor, který umožňuje uživatelům vytvářet komplexní grafické výstupy propojením jednotlivých funkčních modulů (uzlů). Díky tomu lze dosáhnout velmi detailních a vizuálně kvalitních textur, které jsou snadno modifikovatelné a opakovaně použitelné v různých projektech.

Hlavní výhody Substance Designer pro 2D projekty:

- Vysoká modularita – Snadné kombinování uzlů pro rychlou tvorbu a modifikaci 2D textur.
- Parametrická flexibilita – Možnost dynamicky měnit parametry uzlů, což umožňuje rychlé testování různých variant textur bez nutnosti manuálního překreslování.
- Podpora různých výstupů – Export vytvořených textur ve vysokém rozlišení a různých formátech, což zajišťuje kompatibilitu s herními enginy jako Unity nebo Unreal Engine.
- Podpora PBR (Physically Based Rendering) – Umožňuje realistické ztvárnění materiálů i v 2D projektech, například ve hrách s 2D grafikou, které využívají efekty světla a stínů pro dosažení realističtějšího vzhledu.

Na druhou stranu je práce se Substance Designer pro začátečníky náročnější, protože je nutné zvládnout principy tvorby uzlových sítí, což může představovat vysokou vstupní bariéru.

Houdini

Houdini [20] je výkonný procedurální software, který poskytuje nástroje nejen pro 3D, ale také pro pokročilé 2D procedurální generování. Jeho robustní uzlový

systém umožňuje vytváření komplexních scén a grafických prvků s důrazem na automatizaci, variabilitu a reprodukovatelnost. V oblasti 2D grafiky se využívá zejména při generování detailních krajin, terénních prvků, architektury či vegetace, které lze využít i ve dvourozměrných hrách nebo animacích.

Hlavní výhody Houdini pro 2D projekty:

- Komplexní kontrola – Díky uzlovému systému je možné detailně nastavovat a automatizovat tvorbu 2D grafiky a prostředí.
- Možnost skriptování – Podpora vlastních skriptů umožňuje vývojářům rozšířit základní funkcionalitu a upravit algoritmy podle potřeby.
- Integrace – Výborná kompatibilita s herními enginy jako Unity či Unreal Engine umožňuje snadný export výsledků do cílových aplikací.

Nevýhodou může být vyšší vstupní náročnost na pochopení principů uzlového workflow a zvládnutí pokročilých funkcí softwaru.

Processing / p5.js

Processing [36] a jeho webová knihovna p5.js [33] jsou open-source platformy zaměřené na kreativní programování a procedurální generování vizuálního obsahu pomocí jednoduchého, ale flexibilního programovacího jazyka založeného na Javě a JavaScriptu. Tyto nástroje jsou široce používány zejména v oblasti generativního umění, vizuální komunikace a vzdělávání.

Klíčové vlastnosti:

- Jednoduchost a dostupnost – Processing a p5.js mají jasnou a přehlednou syntaxi, což usnadňuje jejich používání i začínajícím programátorům nebo umělcům bez pokročilých programovacích zkušeností.
- Procedurální generování grafických prvků – Umožňují snadno generovat geometrické tvary, složité vzory, animace a interaktivní vizualizace, což je ideální pro rychlé prototypování a experimentální projekty.
- Interaktivita a flexibilita – Oba nástroje podporují interaktivní grafiku a reagují na vstup uživatele v reálném čase, což je činí vhodnými pro tvorbu interaktivních webových a multimediálních aplikací.
- Komunitní podpora a rozšiřitelnost – Existuje je rozsáhlá komunita uživatelů a vývojářů, kteří poskytují množství doplňkových knihoven, tutoriálů a dokumentace, čímž rozšiřují možnosti jejich využití.

GLSL Shadery

GLSL (OpenGL Shading Language) je vysoce specializovaný programovací jazyk určený k tvorbě shaderů, které umožňují generování a úpravu vizuálního obsahu

přímo na grafické kartě (GPU – Graphics Processing Unit). Shadery napsané v GLSL jsou klíčové pro efektivní procedurální generování textur, dynamických vizuálních efektů a animací v reálném čase, zejména v oblasti počítačových her, interaktivních aplikací a simulací [5].

Klíčové vlastnosti GLSL shaderů:

- Procedurální tvorba textur a efektů – GLSL umožňuje tvorbu dynamických textur a realistických efektů, jako jsou vodní plochy, mraky, plameny nebo různé šumové vzory, generované přímo za běhu aplikace.
- Vysoká rychlost a optimalizace – Protože výpočty probíhají přímo na GPU, GLSL poskytuje výrazně vyšší výkon a efektivitu oproti CPU-based řešením, což umožňuje vykreslování složitých scén a efektů v reálném čase.
- Flexibilita a přesnost – Uživatelé mohou detailně ovlivnit vizuální výstupy prostřednictvím nastavitelných parametrů, čímž dosahují vysoké úrovně kontroly nad výsledným vizuálním stylem a chováním grafických prvků.

Uživatelem definované skripty v Unity (C#)

Unity umožňuje procedurální generování 2D grafiky prostřednictvím vlastních skriptů psaných v programovacím jazyce C#. Tento způsob generování grafických prvků poskytuje tvůrcům vysokou míru flexibility a kontroly, protože skripty umožňují přesnou definici algoritmů a přesné řízení generovaného obsahu přímo v prostředí herního enginu.

Klíčové výhody procedurálního generování pomocí C# skriptů v Unity:

- Vysoká přizpůsobitelnost – Použití vlastních skriptů umožňuje vývojářům definovat algoritmy, pravidla a parametry generování, což vede k přesně požadovaným výsledkům, přizpůsobeným potřebám konkrétního projektu.
- Interaktivita a dynamická tvorba – Skripty lze propojit s různými herními událostmi, což umožňuje dynamickou změnu grafiky v reálném čase na základě hráčových aktivit nebo změn v herním prostředí.
- Integrace s Unity ekosystémem – Skripty napsané v C# lze snadno kombinovat s ostatními Unity komponentami, například s animacemi, fyzikálními simulacemi nebo shaderovými efekty vytvořenými pomocí Shader Graph, čímž se rozšiřují možnosti využití generované grafiky.
- Optimalizace výkonu – Protože skripty jsou kompilovány a optimalizovány přímo Unity enginem, generování grafiky může být velmi efektivní a rychlé, což je vhodné i pro mobilní platformy nebo zařízení s omezenými výpočetními zdroji.

2.3.3 Procedurální generování v Unity

Unity je široce využívaný herní engine, který poskytuje komplexní podporu procedurálního generování grafiky, včetně pokročilých možností pro 2D projekty. Díky flexibilitě svého skriptovacího systému a integrovaných nástrojů umožňuje vývojářům efektivně vytvářet různorodý obsah s minimálními manuálními zásahy. Unity podporuje několik technik procedurálního generování, které lze snadno přizpůsobit potřebám konkrétního projektu.

Nejčastěji používané metody procedurálního generování v Unity:

Noise-based metody

Noise-based metody využívají matematické funkce k procedurálnímu generování realisticky vypadajících struktur, textur nebo pozadí. Tyto metody jsou založené na algoritmech, které generují pseudo-náhodné, ale spojité a přirozeně vypadající vzory, vhodné pro různé aplikace – od terénů až po organické textury.

Mezi nejznámější noise-based metody patří:

- Perlin Noise – Navržený Kenem Perlinem, generuje spojité a hladké přechody, což ho činí ideálním pro tvorbu terénů, oblaků, kouře nebo organických textur. Jeho výhodou je schopnost generovat přirozeně působící výsledky s jednoduchou implementací [35].
- Simplex Noise – Optimalizovaná varianta algoritmu Perlin Noise s nižšími výpočetními nároky a menším množstvím artefaktů (tedy viditelných pravidelných vzorů, které snižují přirozenost generované grafiky). To umožňuje výhodnější využití při tvorbě složitých scén a efektů v reálném čase.
- Voroného diagramy – Matematická metoda rozdělení roviny na jednotlivé oblasti, přičemž každý bod roviny patří do oblasti podle toho, ke kterému ze zadaných bodů (seed points) má nejkratší vzdálenost. Hranice těchto oblastí jsou tvořeny úsečkami stejně vzdálenými od dvou nejbližších bodů. Voroného diagramy se často využívají při generování buněk, mozaikových vzorů, nerovnoměrného rozmístění objektů nebo při tvorbě realistických textur, jako jsou struktury kůže, dlažby či buněk [39].

Procedurální generování v Unity tak nabízí široké možnosti pro tvorbu dynamického a variabilního vizuálního obsahu s vysokou mírou automatizace a přizpůsobitelnosti.

Tilemap generátory

Tilemap generátory v Unity využívají systém dlaždic (tilemap) pro efektivní generování a správu herních prostředí, která sestávají z pravidelně se opakujících grafických prvků [54]. Tento přístup umožňuje vývojářům vytvářet komplexní a variabilní herní prostředí s minimálními nároky na paměť a výpočetní výkon.

Nejpoužívanější algoritmy pro tilemap generátory:

- Roguelike Dungeon Generator – Algoritmus, který na základě předem definovaných pravidel a omezení generuje náhodné, ale zároveň hratelné herní úrovně. Často se využívá pro hry typu roguelike, kde je důležitá náhodnost a variabilita jednotlivých herních úrovní [32].

- Wave Function Collapse (WFC) – pokročilý algoritmus, který generuje herní mapy na základě malých vzorků dlaždic a zachovává konzistenci prostředí, tedy dodržování pravidel a logických vazeb mezi prvky prostředí. Tato metoda umožňuje vytvářet detailní a koherentní mapy, které působí jako vizuálně ucelený celek bez náhodných přechodů, s přesně definovanými pravidly, které zachovávají vizuální logiku a strukturu prostředí [18].

Fraktálové algoritmy

Fraktálové algoritmy využívají matematické principy, které umožňují vytvářet struktury opakující se v různých měřítkách. Právě tato opakovatelnost a hierarchická složitost je činí vhodnými pro procedurální generování realistických přírodních útvarů, jako jsou hory, řeky, vegetace nebo oblaka. V Unity se tyto algoritmy implementují převážně pomocí rekurzivních funkcí nebo shaderů, což umožňuje efektivně generovat vizuálně složitá a přirozeně vypadající prostředí.

Nejpoužívanější fraktálové algoritmy:

- Midpoint Displacement Algorithm – Používá se k generování realistických terénů pomocí iterativního dělení a posouvání bodů, čímž vytváří nepravidelné a přirozeně vypadající krajiny. Tento algoritmus je vhodný pro 2D i 3D hry [27].

- L-systémy (Lindenmayerovy systémy) – Rekurzivní pravidla umožňují generovat složité vegetační struktury, jako jsou stromy, keře a rostliny. Algoritmus postupně iteruje přes jednoduchá pravidla, čímž dosahuje realistických a detailních organických tvarů [37].

Procedurální generování v Unity pomocí tilemap generátorů a fraktálních algoritmů nabízí široké možnosti pro automatizaci a optimalizaci tvorby herních světů, čímž šetří čas vývojářům a zajišťuje vysokou variabilitu herních prostředí.

2.4 Shadery ve 2D grafice a jejich využití

Shadery, které běží přímo na grafické kartě (GPU), hrají klíčovou roli při tvorbě vizuálních efektů. Zajišťují správné zobrazení textur, osvětlení, barevných úprav a dalších vizuálních vlastností objektů [5].

Typy shaderů relevantní pro 2D grafiku:

- Vertex shader – zpracovává pozice vrcholů spritů, umožňuje aplikaci transformací, jako jsou rotace, škálování a výpočty související s mapováním textur [42].

- Fragmentový shader, někdy nazývaný pixelový – zajišťuje finální barvení pixelů, včetně práce se světlem, stíny, šumem nebo postprocessingovými efekty [42].
- Compute shader – méně běžný v čistě 2D projektech, ale využitelný pro složité výpočty, jako je generování textur, simulace částicových efektů nebo pokročilá manipulace s obrazovými daty [5].

Zatímco tradiční vykreslování pomocí bitmap a spritů je založené na předpřipravených statických obrázcích, shadery umožňují dynamické generování a úpravu grafiky v reálném čase. To otevírá nové možnosti jak pro stylizaci prostředí, tak i pro efektivnější práci s pamětí a výkonem.

Ve 2D hrách se shadery využívají především pro tvorbu různých vizuálních efektů – od simulace světelných podmínek až po procedurální animace či barevné korekce.

Efekty dosažitelné pomocí shaderů ve 2D:

- Dynamické osvětlení a stíny – Shadery umožňují simulaci světelných zdrojů, které reagují na pohyb hráče nebo události ve hře. Díky tomu lze dosáhnout efektů jako blikající světla nebo realisticky vypadající stíny. Mezi časté efekty patří také ambientní záření, což je jemné rozptýlené světlo bez konkrétního směru, které vyplňuje celou scénu a vytváří dojem prostoru [5].
- Postprocessingové efekty – Používají se pro úpravu obrazu po jeho vykreslení. Patří mezi ně například motion blur - rozmazání pohybu, barevné filtry, vignette - efekt ztmavení okrajů. Častým efektem je také simulace starých CRT obrazovek, která napodobuje vlastnosti starších monitorů, například zakřivení obrazu, viditelné horizontální linky, jemné barevné odchylky nebo slabé blikání [11].
- Procedurální textury a šumové vzory – Pomocí fragmentových shaderů lze generovat opakovatelné a zároveň variabilní textury v reálném čase, např. vzory struktur (kámen, dřevo, voda) nebo animované efekty jako plameny či mlha. Běžně se používají algoritmy jako Perlin nebo Simplex noise, které jsou podrobněji popsány v kapitole Procedurální generování v Unity [11].
- Deformace a animace – Shadery umožňují aplikaci různých vizuálních deformací obrazu, jako jsou například vlnové efekty, zkreslení nebo tzv. glitch artefakty – ty napodobují grafické chyby známé ze starších televizorů, herních konzolí nebo počítačových monitorů, například náhodné posuny pixelů, barevné poruchy nebo rozpad obrazu. Takové efekty mohou být reakcí na vstupy uživatele, herní události nebo sloužit k vytvoření specifické atmosféry. Nahrazují náročné animace a mohou být využity k vytvoření specifických vizuálních efektů, které ovlivňují styl nebo vnímání hry. Ačkoliv původně mohly tyto artefakty působit rušivě, dnes jsou často využívány záměrně.
- Zvýraznění interakcí a stavů objektů – Například vizuální odezva při zranění hráče jako záblesk červené barvy, zvýraznění objektu při interakci nebo změna barevné teploty v závislosti na prostředí.

2.4.1 Implementace shaderů v Unity

Unity jako multiplatformní herní engine poskytuje komplexní podporu pro práci s technologiemi shaderů, které ve 2D projektech hrají klíčovou roli při tvorbě vizuálních efektů, postprocessingů a procedurálních textur. Shadery v Unity lze implementovat dvěma hlavními způsoby: pomocí vizuálního nástroje Shader Graph nebo skrze ručně psaný kód v jazyce HLSL v kombinaci s ShaderLabem.

Shader Graph

Shader Graph je uzlový editor, který umožňuje tvorbu shaderů bez potřeby psaní kódu [2]. Každý uzel představuje funkční blok, vstupní hodnotu nebo datový typ, např. texturu, barvu. Tento přístup je vhodný pro designéry, kteří chtějí experimentovat s efekty a interakcemi bez hlubší znalosti shaderových jazyků.

Shader Graph je plně kompatibilní s Universal Render Pipeline (URP – Universal Render Pipeline) [23], která nahrazuje starší renderovací systémy a přináší větší flexibilitu, optimalizaci a konzistentní chování napříč platformami.

Renderovací pipeline (neboli vykreslovací řetězec) je systém, který určuje, jak Unity převádí herní scénu do výsledného obrazu na obrazovce – tedy jak se z dat o objektech, materiálech, světlech a kamerách vytvoří finální vykreslení.

URP je moderní varianta tohoto systému. Je navržena pro vysoký výkon a zároveň vizuální kvalitu, a to jak pro 2D, tak 3D hry. URP funguje napříč platformami včetně mobilních zařízení, počítačů i konzolí. Zahrnuje také vlastní podpůrnou sadu shaderů, přístup ke kamerám, postprocessingovým efektům a dalším renderovacím funkcím, což rozšiřuje možnosti tvorby a úprav vizuálních stylů.

Ručně psané shadery (HLSL / ShaderLab)

Pro pokročilejší a optimalizovaná řešení je možné vytvářet shadery ručně v jazyce HLSL [19]. Tento jazyk se v Unity používá v rámci struktury *ShaderLab*, která definuje rozvržení shaderu, jeho parametry a průchody. *ShaderLab* zároveň určuje, jak bude shader propojen s materiály a jak se bude chovat při vykreslování [40].

HLSL umožňuje přímou manipulaci s grafickými daty. Lze například upravovat pozice vrcholů, měnit textury nebo počítat vlastní světelné efekty. Díky tomu má vývojář plnou kontrolu nad vzhledem a chováním grafiky. Tento přístup je vhodný pro tvorbu nestandardních efektů nebo optimalizaci výkonu.

Ručně psané shadery se využívají například tam, kde je potřeba nízkoúrovňový přístup, přesné řízení výpočtů nebo komplexní logika, kterou není možné jednoduše sestavit v Shader Graphu.

Použití shaderů v Unity projektech

Ve 2D hrách se shadery nejčastěji využívají v kombinaci s komponentou *SpriteRenderer* [49]. Tato komponenta umožňuje přiřadit materiál, který obsahuje konkrétní shader, a tím ovlivnit způsob vykreslení spritu. Při práci se shadery je důleži-

té zohlednit cílovou platformu, protože výpočetně náročné efekty mohou negativně ovlivnit výkon na slabších zařízeních.

Shadery se často používají k implementaci specifických vizuálních prvků, jako jsou světelné odlesky, simulace ohně, kouře nebo mlhy, které přispívají k vizuální atmosféře hry. V praxi může jít například o zobrazení různých typů deformací, barevných přechodů, animací textur nebo napodobení světelných podmínek. Výhodou je možnost dynamicky měnit parametry shaderu v reálném čase a přizpůsobovat jeho chování různým herním situacím.

3 Návrh a tvorba grafických prvků

3.1 Metody tvorby grafiky

Před implementací shaderů, animací a dalších vizuálních prvků bylo nezbytné vytvořit grafické podklady, které mohly být dále použity, technicky zpracovány a vizuálně upravovány. Některé sprity nepředstavují finální vizuál, ale slouží jako vstupní data pro pozdější práci s implementací shaderů nebo animací.

3.1.1 Tvorba v Adobe Illustratoru

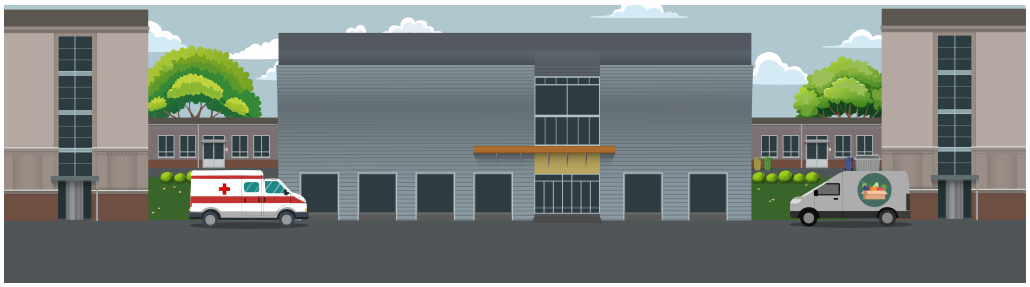
Před samotnou tvorbou finálních grafických prvků byly pomocí softwaru Adobe Photoshop vytvořeny počáteční grafické koncepty. V této fázi byly ručně načrtnuty základní vizuální návrhy jednotlivých assetů, především mapy herního prostředí a hlavní postavy. Návrhy byly vytvářeny pomocí grafického tabletu Wacom Intuos M, což umožnilo plynulou kresbu a jemnou práci s detaily.

Tyto náčrty sloužily jako vizuální reference pro následné zpracování ve vektorové formě. Díky tomu bylo možné zajistit konzistentní styl a zároveň zachovat původní charakter ručně kreslených návrhů (viz Obrázek 3.1).

Obrázek 3.1: Ručný náčrt herního prostředí sloužící jako vizuální reference

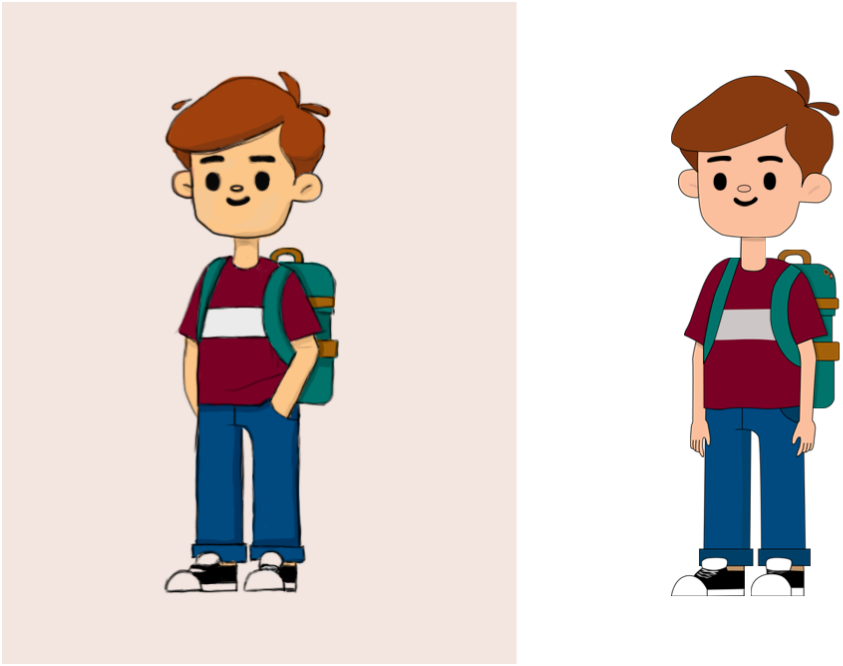
Po dokončení počátečních návrhů byl pro finální tvorbu grafiky zvolen software Adobe Illustrator, který nabízí rozsáhlé možnosti pro práci s vektorovou grafikou. Tento nástroj byl upřednostněn pro svou přesnost, flexibilitu při úpravách a možnost bezztrátového škálování, což je při vývoji 2D her klíčové.

Jednotlivé objekty byly v Illustratoru překresleny do čistě vektorové podoby, přičemž byly definovány tvarové kontury, barevné výplně a vrstvy. Tímto způsobem byly vytvořeny kompletní grafické podklady, které byly následně připraveny k implementaci do herního enginu (viz Obrázek 3.2).



Obrázek 3.2: Finální grafický návrh herního prostředí převedený do vektorové podoby

Podobným způsobem byla vytvořena také hlavní postava hry. Nejprve byl v Adobe Photoshopu vytvořen ručně kreslený návrh, který byl následně překreslen do vektorové podoby v Adobe Illustratoru. Díky tomuto postupu bylo možné připravit vizuálně konzistentní asset, který mohl být dále využit při různých animačních technikách, včetně frame-by-frame animace a riggingu. Obě verze postavy jsou uvedeny na Obrázku 3.3.



Obrázek 3.3: Hlavní postava převedená z ručně kresleného návrhu do vektorové podoby

3.1.2 Generování v Pythonu

V rámci praktické části práce je pozornost věnována možnostem procedurálního generování grafiky s využitím programovacího jazyka Python. Jsou zkoumány vybrané metody a algoritmy, které umožňují automatizované vytváření komplexních vizuálních struktur. Kapitola se zaměřuje na implementaci a analýzu tří odlišných přístupů: generování Voroného diagramů pro tvorbu textur, rekurzivní generování Kochovy vločky jako příkladu fraktální geometrie a vizualizaci Mandelbrotovy množiny demonstrující iterativní fraktální procesy.

Voronoi

Voroného diagram je geometrická struktura, která rozděluje rovinu na oblasti (tzv. Voroného buňky), přičemž každému bodu roviny je přiřazena ta oblast, jejíž generující bod (z množiny $P = \{p_1, p_2, \ldots, p_n\} \subset \mathbb{R}^2$) je mu nejblíže vzhledem k Euklidovské vzdálenosti.

Formálně je Voroného buňka pro bod p_i definována jako:

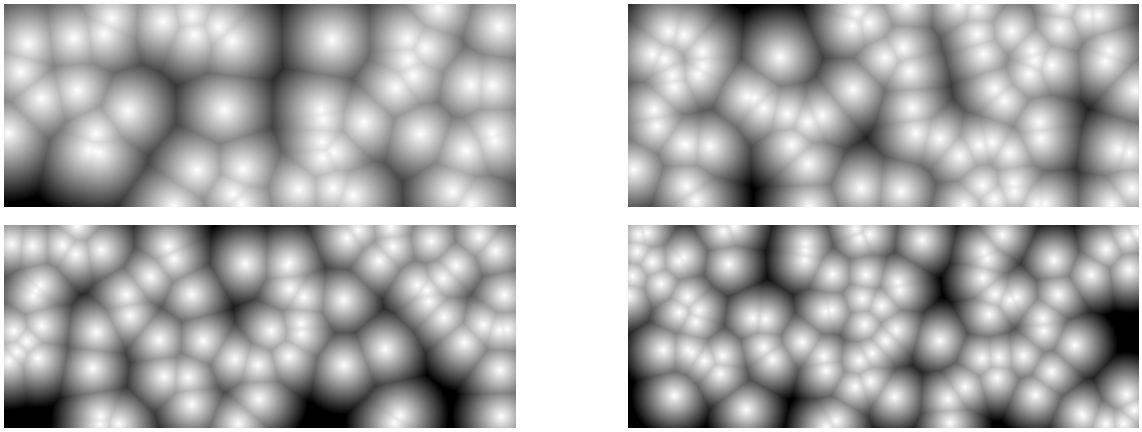
$$V(p_i) = \{x \in \mathbb{R}^2 \mid \forall j \neq i,\ \|x - p_i\| \leq \|x - p_j\|\}$$

K vizualizaci Voroného diagramu byla použita knihovna *scipy.spatial.Voronoi* [38] v jazyce Python. Implementace generování textury je uvedena v Příloze A

Na začátku algoritmu se definují základní parametry výstupního obrázku – konkrétně jeho šířka (*width*) a výška (*height*). Zvolí se počet generujících bodů Voroného diagramu. V tomto případě jde o 50 bodů, které tvoří množinu P. Každý bod p_i je náhodně rozmístěn v rámci obrázku, přičemž jeho souřadnice jsou vygenerovány pomocí funkce `np.random.rand(num_points, 2) * [width, height]`. Tímto způsobem je zajištěno rovnoměrné rozložení bodů v celém rozsahu obrázku.

Následně se pro každý pixel obrázku se souřadnicemi (x, y) vypočítá jeho Euklidovská vzdálenost ke všem generujícím bodům p_i. Tato vzdálenost se určuje pomocí `distance = np.min(np.linalg.norm(points - [x, y], axis=1))`, přičemž z celé množiny vzdáleností se vybere minimální hodnota. Tato minimální vzdálenost reprezentuje, jak daleko se daný pixel nachází od svého nejbližšího generujícího bodu.

Pro zobrazení výsledků v podobě rastru se tato minimální vzdálenost dále zpracovává. Hodnota se normalizuje do rozsahu 0–255 vzhledem k maximální očekávané vzdálenosti, následně se invertuje a použije jako hodnota jasu daného pixelu ve výsledném obrázku. Takto vzniká vizualizace, ve které jsou hranice mezi jednotlivými Voroného buňkami – tedy místa, kde se vzdálenosti ke dvěma nebo více bodům přibližně rovnají – znázorněny tmavšími liniemi.



Obrázek 3.4: Různé varianty Voroného diagramu

Výsledný obrázek tvoří procedurálně generovanou texturu, kterou je možné využít jako vstupní prvek pro shaderové efekty v prostředí Unity. Voroného diagram poskytuje vysokou míru variability při zachování jednoduché parametrické kontroly nad výstupem jak ukazují jednotlivé varianty na Obrázku 3.4.

Rekurzivní generování sněhových vloček

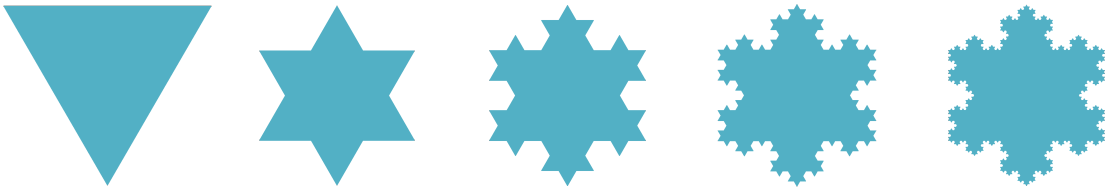
Pro další využití procedurálního generování grafiky byla zvolena implementace fraktálního útvaru známého jako Kochova vločka. Tato geometrická struktura vzniká rekurzivní úpravou úsečky pomocí tzv. Kochovy křivky, která patří mezi klasické příklady deterministických fraktálů [24].

Implementace byla provedena v jazyce Python s využitím knihovny `turtle` [56], která umožňuje jednoduché kreslení pomocí tzv. želví grafiky. Kompletní skript je uveden v Příloze B

Kochova křivka vzniká tak, že se každý úsek (segment) rekurzivně rozděluje na čtyři nové segmenty. Nejprve se původní úsek rozdělí na tři stejné části. Prostřední část je nahrazena dvojicí úseků, které tvoří výběžek ve tvaru rovnostranného trojúhelníku bez spodní základny. Tyto nové úseky vznikají pootočením o $60°$ a $-120°$. Tento transformační krok se následně rekurzivně aplikuje na každý nový segment podle nastavené hloubky (`depth`), která určuje počet úrovní rozdělení.

Při `depth = 0` se segment vykreslí přímo jako úsečka. Pro vyšší hodnoty `depth > 0` se aplikují transformační pravidla: segment se rozdělí na čtyři nové části, přičemž se využívají rotace pomocí příkazů `turtle.left(60)` a `turtle.right(120)`. Výsledkem je lomená čára s čím dál složitějším tvarem, která se s každou další úrovní rekurze stává detailnější.

Celá sněhová vločka je následně vytvořena ze tří po sobě jdoucích Kochových křivek, které dohromady tvoří uzavřený rovnostranný trojúhelník. Tato struktura vzniká v hlavní funkci `draw_snowflake()`, která vykreslí tři strany Kochovy křivky s rotací o $120°$ po každé straně. Výsledný útvar má vysoký stupeň symetrie a vizuálně připomíná sněhovou vločku.



Obrázek 3.5: Vygenerované obrázky s hloubkou 0 až 4

Výstup byl exportován ve formátu Encapsulated PostScript (.eps), který umožňuje jeho další vektorové zpracování v externích grafických editorech.

Výhodou tohoto přístupu je přesná kontrola nad mírou detailů a symetrií výsledného útvaru. Parametrickou změnou hloubky rekurze lze snadno měnit vizuální složitost výsledného tvaru bez potřeby jakéhokoliv manuálního zásahu. Vytvořená Kochova vločka tak slouží jako statický grafický prvek, který je možné dále využít jako vstup pro efekt sněžení nebo jako dekorativní prvek v herní grafice.

Mandelbrotova množina

Další metodou procedurálního generování grafiky byla vizualizace Mandelbrotovy množiny – matematického objektu, který vzniká iterativním vyhodnocováním komplexní kvadratické posloupnosti [7]. Každý bod c v komplexní rovině je testován, zda při dosazení do rekurentního vztahu

$$z_{n+1} = z_n^2 + c, \quad z_0 = 0$$

vede k posloupnosti, jejíž absolutní hodnota $|z_n|$ zůstává omezená i po velkém počtu iterací. Pokud hodnota $|z_n|$ překročí zvolenou mez (v tomto případě 2), považuje se bod c za nepatřící do Mandelbrotovy množiny (posloupnost diverguje). Naopak pokud i po maximálním počtu iterací $|z_n| \leq 2$, bod c je součástí množiny.

Každému bodu komplexní roviny (přesněji každému pixelu výsledného obrázku) odpovídá konkrétní komplexní číslo c. Algoritmus pak testuje, po kolika iteracích (pokud vůbec) daná posloupnost překročí mezní hodnotu. Počet iterací je použit k určení jasu daného pixelu – čím rychleji hodnota překročí mez, tím světlejší odstín dostane. Pokud nepřekročí nikdy, pixel je zcela černý.

Výsledný tvar Mandelbrotovy množiny je složitý, uzavřený a má fraktální obrys – to znamená, že jeho hranice je nekonečně detailní a vykazuje samopodobnost při různých měřítkách přiblížení. I malá výseč obrázku tak může při dostatečném zvětšení obsahovat nové struktury a vzory.

Použitý vzorec pro převod počtu iterací na barvu je následující:

$$\texttt{color} = 255 - \left\lfloor \frac{255 \cdot n}{\texttt{max\_iter}} \right\rfloor$$

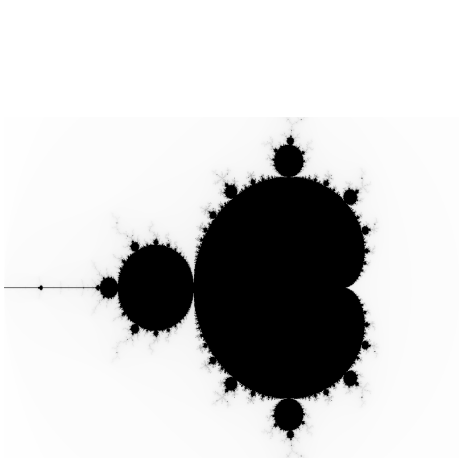
Parametr `max_iter` představuje maximální počet iterací (v testovaných variantách bylo použito 10, 250 a 1000), přičemž vyšší hodnoty umožňují jemnější rozlišení hranic množiny a přesnější vykreslení detailů. Proměnná n značí počet provedených

iterací, během nichž absolutní hodnota komplexního čísla z zůstala menší nebo rovna 2. Porovnání výstupů pro různé hodnoty iterací a rozlišení je znázorněno na Obrázku 3.6.

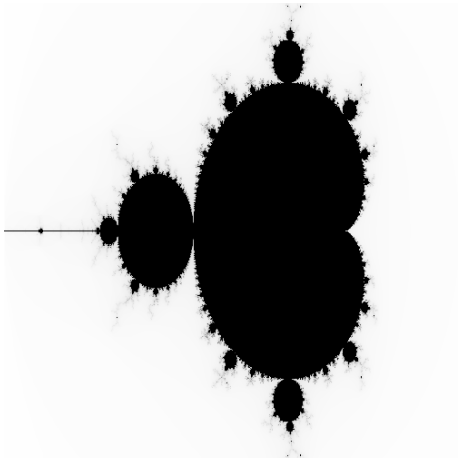
Pro maximální počet iterací 1000:

- Pokud absolutní hodnota $|z|$ překročí hodnotu 2 již po 10 iteracích (tj. posloupnost diverguje velmi brzy), pixel je téměř bílý (hodnota 253).
- Pokud k překročení nikdy nedojde ani po 1000 iteracích (tj. posloupnost zůstává ohraničená), pixel je černý (hodnota 0).

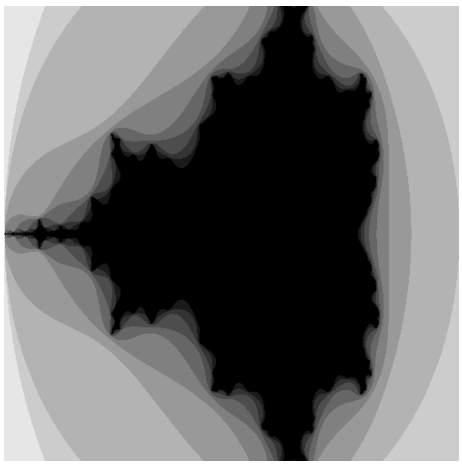
Výsledné obrázky slouží jako vizuální prvky, které mohou být dále využity jako textury nebo zdroj pro shaderové efekty v prostředí Unity. Fraktální charakter Mandelbrotovy množiny nabízí vysoký stupeň vizuální složitosti bez nutnosti ruční kresby a zároveň zajišťuje opakovatelnost výstupů při zachování parametrické kontroly. Implementace algoritmu pro vytvoření Mandelbrotovy množiny je uvedena v Příloze C.



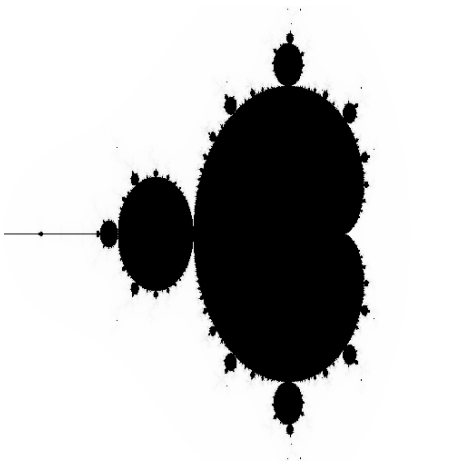
250 iterací, 800×600



250 iterací, 500×500



10 iterací – prstence



1000 iterací – detail

Obrázek 3.6: Porovnání výstupů Mandelbrotovy množiny při různém počtu iterací a rozlišení

3.1.3 Material Maker

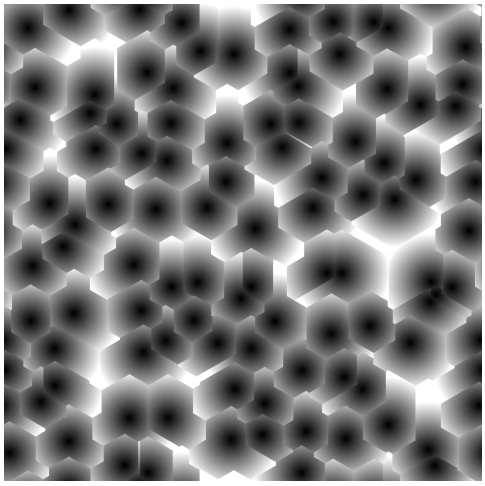
Pro vytváření materiálových podkladů pro následnou práci se shadery byl využit nástroj Material Maker [30], postavený na herním enginu Godot [16]. Jedná se o nástroj umožňující procedurální tvorbu textur pomocí uzlového systému. Výsledkem jsou textury, které mohou sloužit jako vstupní složky konkrétních shaderů.

Material Maker nabízí širokou škálu předdefinovaných uzlů pro generování šumu, geometrických vzorů, barevných gradientů i efektů simulujících přirozené materiály. Výsledné materiály lze exportovat jako obrázky nebo převést do GLSL kódu, který je možné dále upravit nebo integrovat do jiných systémů, například Unity.

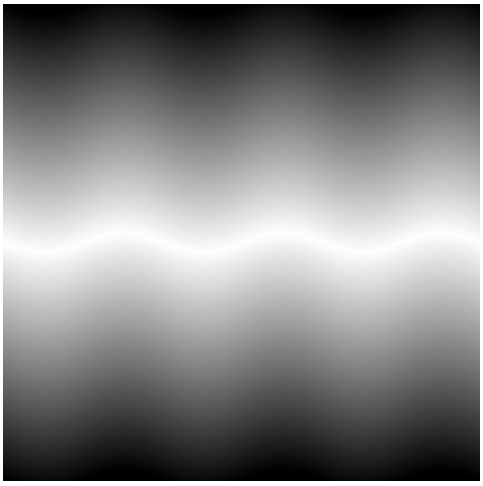
V rámci této bakalářské práce byly využity konkrétní uzly pro tvorbu základních textur a grafických motivů:

- Clouds – generuje hladký procedurální šum, založený na interpolaci náhodných hodnot. Výstupem je jemná textura s plynulými přechody, často využívaná jako základ pro další vrstvení nebo maskování.
- Dirt – vytváří texturu simulující nepravidelný a chaotický šum s vyšší mírou kontrastu. Výsledná mapa má charakteristické tmavé skvrny a zrnité přechody.
- Sine Wave – generuje pravidelný vlnový vzor na základě sinusové funkce. Lze nastavit frekvenci a směr vln, případně je kombinovat s dalšími generátory.
- Triangle Voronoi – varianta Voroného algoritmu, která používá trojúhelníkovou topologii pro výpočet buněk. Výstupem je struktura s ostře definovanými hranami a pravidelnými geometrickými vzory.

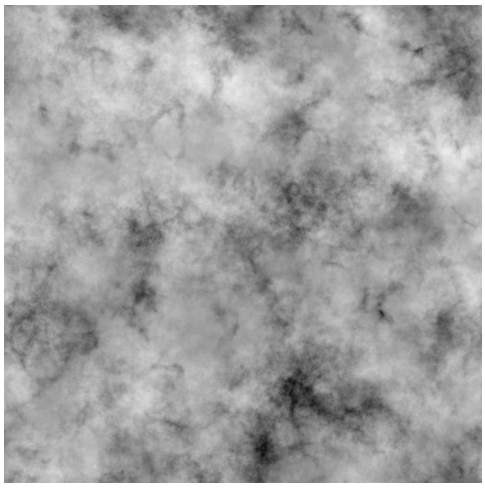
Přehled textur vytvořených pomocí jednotlivých uzlů lze vidět na Obrázku 3.7. Výsledné textury byly exportovány ve formátu PNG a následně použity jako vstup v Unity pro tvorbu shaderů.



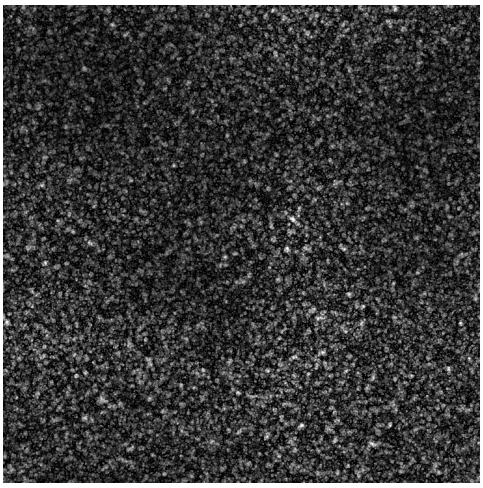
Triangle Voronoi



Sine Wave



Clouds



Dirt

Obrázek 3.7: Ukázky různých procedurálně generovaných textur vytvořených pomocí shaderů.

3.1.4 LeonardoAI a ChatGPT

V neposlední řadě byl pro tvorbu grafických podkladů využit generátor obrázků Leonardo AI [26] ve spojení s jazykovým modelem ChatGPT [22]. Umělá inteligence byla použita především při návrhu pozadí pro příběhové cutscény (filmová sekvence v rámci hry sloužící k vyprávění příběhu), kde by ruční tvorba zabrala značné množství času a zároveň by neodpovídala hlavní náplni této práce. ChatGPT byl využit k vytvoření textových zadání (popisů scén), která byla následně vložena do nástroje Leonardo, přičemž pro tvorbu výstupů byl použit model Phoenix 1.0.

Příklad dotazu pro LeonardoAI: *„A young male student with short, slightly messy light brown to auburn hair and a side part stands in front of a modern silver university building with large glass entrance doors, reflecting the bright morning sunlight. His deep brown eyes are filled with uncertainty as he adjusts the straps of a green backpack*

slung over his shoulders. He wears blue jeans and a red T-shirt with a white horizontal stripe across the chest, the fabric slightly wrinkled as if he had just finished dressing. His posture is tense, one hand still adjusting his backpack while the other rests loosely at his side, as if caught mid-question. The air is crisp, and the plaza around him is mostly empty, aside from the faint silhouettes of distant students walking toward the entrance. The highly photorealistic style captures the natural lighting, realistic textures of his clothing, and the depth of his expression as he grapples with the strange events leading up to this moment.“.

Výsledný Obrázek 3.8 generovaný na základě tohoto popisu zachycuje navrženou scénu ve fotorealistickém stylu a byl použit jako grafický podklad pro příběhovou část hry.

Obrázek 3.8: Výsledný výstup z Leonardo AI vytvořený na základě zadaného popisu scény

3.2 Tvorba shaderů

Pro tvorbu shaderů byl zvolen nástroj Shader Graph, který je součástí Unity a plně s ním integrovaný, což z něj činí ideální prostředí pro vizuální návrh shaderových efektů. V rámci práce bylo vytvořeno sedm shaderů, ze kterých vzniklo celkem osmnáct materiálů.

První skupinu tvoří celoobrazové shadery, které se neaplikují na konkrétní objekt ve scéně, ale přímo na kameru – tedy ovlivňují celý obraz výsledného výstupu. Dále byl vytvořen shader určený k vizuální podpoře herní mechaniky změny ročního období. Tento shader umožňuje plynulé prolínání mezi čtyřmi variantami jedné plošiny, čímž zajišťuje hladký přechod mezi jednotlivými ročními obdobími. Plošiny tvoří hlavní stavební blok ve hře, ve které byly vytvořené výstupy použity.

Poslední shader navazuje na předchozí a rozšiřuje ho o efekt praskání. Pomocí vstupního Voroného materiálu vytváří vizuální iluzi narušení povrchu plošiny a slouží

jako grafický základ pro herní mechaniku zničitelných plošin (tj. plošin, které se po určitém počtu kolizí zničí a po omezenou dobu na ně hráč nemůže znovu skočit), která je ve vybrané hře implementována.

3.2.1 Celoobrazové shadery

Celoobrazové shadery jsou speciální typy shaderů, které se neaplikují na jednotlivé objekty ve scéně, ale přímo na celý obrazový výstup kamery. Používají se k plošným efektům, jako jsou barevné filtry, šum, vinětace nebo jiné stylizace, které ovlivňují celý snímek bez ohledu na to, co je právě ve scéně zobrazeno.

Celoobrazové shadery v Unity se aplikují prostřednictvím tzv. *Renderer Feature*, což je rozšíření renderovací pipeline. Pro 2D projekty v URP se využívá *Renderer2D*, což je specifická konfigurace renderovacího procesu navržená přímo pro dvourozměrnou grafiku. V rámci nastavení URP se jedná o konkrétní typ vykreslovacího modulu, který určuje, jak bude scéna zpracována a zobrazena. *Renderer2D* umožňuje použití světel, průhlednosti nebo efektů typických pro 2D prostředí. Aby bylo možné použít celoobrazové shadery, je třeba v této konfiguraci aktivovat příslušnou funkci pomocí komponenty *Full Screen Pass Renderer Feature* (rozšíření, které umožňuje aplikovat efekt přes celý obrazový výstup kamery pomocí zvoleného shaderu).

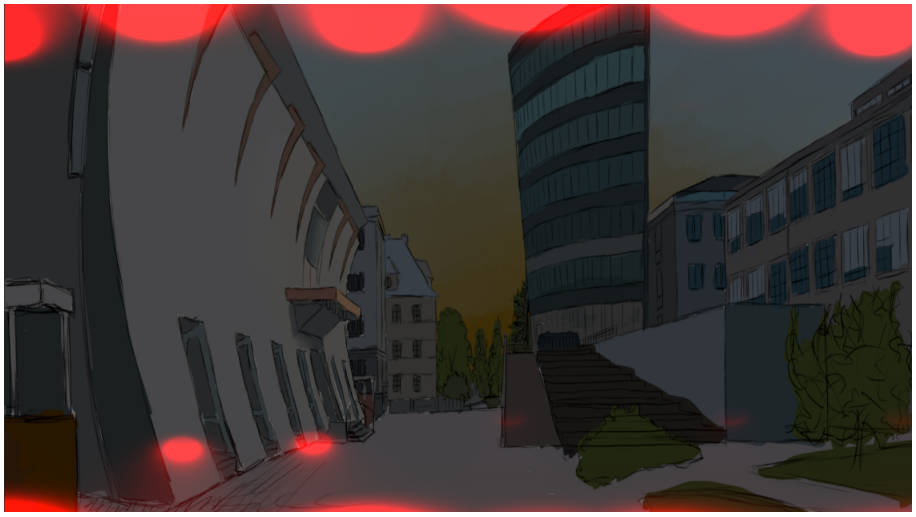
Po přidání komponenty *Full Screen Pass Renderer Feature* do nastavení *Renderer2D* je třeba nastavit tyto dvě klíčové vlastnosti, které komponenta *Full Screen Pass Renderer Feature* [13] obsahuje:

- *Pass Material* – sem se přiřazuje materiál, který vychází z vytvořeného shaderu a obsahuje jeho nastavení. Shader samotný v Unity není možné aplikovat přímo – vždy je nutné jej přiřadit materiálu, který následně slouží jako prostředník mezi shaderem a vykreslovaným objektem (v tomto případě celou obrazovkou). Pro fullscreen efekty se používají materiály založené na typech shaderů, které nezávisí na světelných podmínkách ve scéně (tzv. *Unlit shadery* [58]).
- *Injection Point* – určuje, kdy během vykreslování Unity aplikuje daný shader. V praxi to znamená, v jaké fázi renderovacího procesu se efekt přidá:
  - *Before Rendering Transparents* (před vykreslením průhledných objektů): Shader se aplikuje po vykreslení skyboxu (statického pozadí scény, například oblohy nebo panoramatu), ale ještě před tím, než se začnou zobrazovat průhledné objekty ve scéně.
  - *Before Rendering Post Processing* (mezi průhlednými objekty a dalšími obrazovými efekty): Shader se aplikuje po všech objektech včetně průhledných, ale ještě před postprocessingovými efekty, jako jsou rozmazání, barevné korekce apod.
  - *After Rendering Post Processing* (úplně na závěr, až po všech efektech): Shader se aplikuje jako poslední krok renderovacího procesu. Tato možnost je nejčastější a zároveň výchozí, protože zajistí, že shader ovlivní celý finální obraz a žádný další efekt ho už nepřepíše.

Shader poškození

Shadery pro zobrazení poškození obrazovky vytváří kontrastní červený efekt pomocí výrazné Voroného textury, která připomíná praskliny nebo poškozený povrch. Tato textura je generována přímo v Shader Graphu, kde je dále upravena z hlediska jasu a kontrastu a následně zabarvena dočervena, čímž vzniká efekt indikující zásah hráče. Příklad výsledného vizuálního efektu je uveden na Obrázku 3.9.

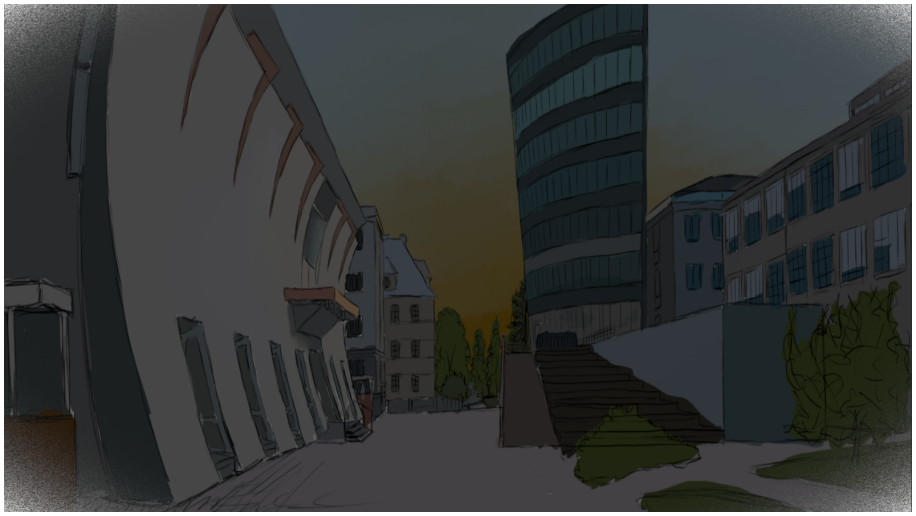
Výstup tohoto shaderu je omezen vertikální maskou, která zajistí, že se efekt neprojeví ve středu obrazovky, ale pouze na jejích horních a dolních okrajích. Maskování využívá gradient, jenž v centrální části pixelům nastavuje nulovou průhlednost a na krajích plynule přechází do plné intenzity. Výsledný efekt evokuje záblesk či poškození bez narušení viditelnosti hlavního dění ve hře.



Obrázek 3.9: Ukázka shaderu poškození obrazovky

Shader mrznutí

Shader s efektem mrznutí simuluje pomalé pokrytí obrazovky ledovou vrstvou, která začíná ve středu a postupně se rozšiřuje směrem k okrajům. Jeho základ tvoří tmavá radiální maska s nastavitelným poloměrem a jemným šumem, který dodává výslednému efektu organický a přirozený vzhled. Textura je následně obarvena bíle a výstup je aplikován jako celoobrazový shader. Shader byl použit v kombinaci s herní mechanikou změny ročního období, konkrétně pro zimu. Vizuální podoba tohoto efektu je znázorněna na Obrázku 3.10.

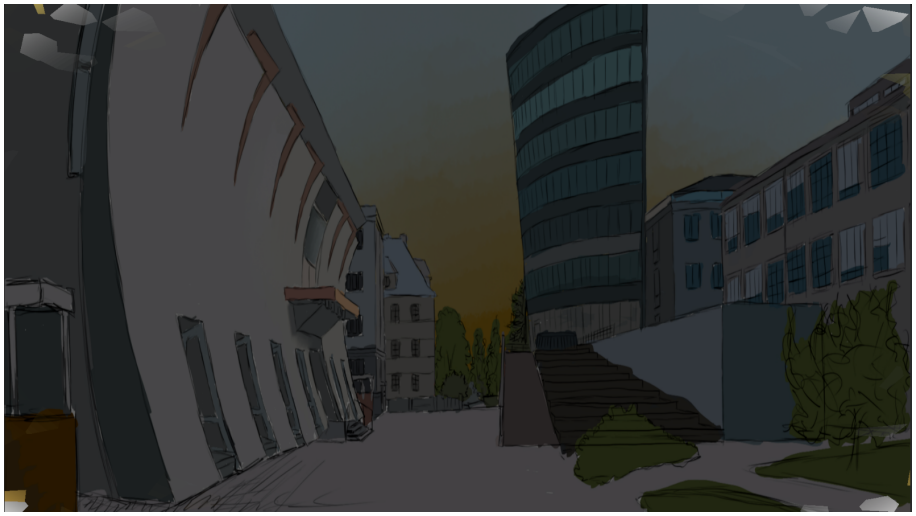


Obrázek 3.10: Ukázka shaderu mrznutí

Shader magie

Shader magie využívá Voroného diagram, který je opět generován přímo v Shader Graphu. Tentokrát však místo klasického výstupu `Out` používá výstup `Cells`, díky čemuž vzniká zcela odlišná vizuální struktura – ostřeji ohraničené buňky připomínající dlaždice nebo hexagonální rozdělení.

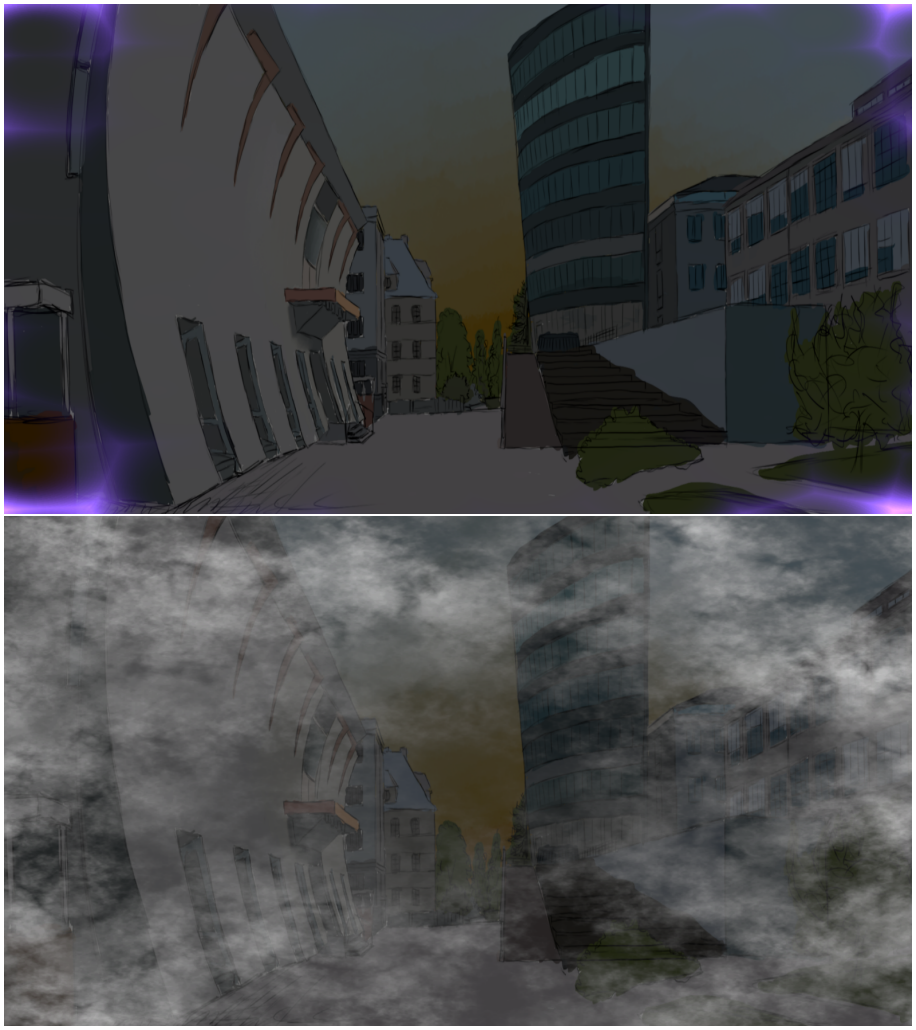
Tento efekt je navíc pohyblivý – pozice jednotlivých buněk se v čase mění, čímž vzniká dojem plynule se proměňující buněčné struktury, která navozuje efekt proudící magické energie. Materiál, který vychází z tohoto shaderu, je řízen parametrem `Speed`, který ovlivňuje rychlost pohybu vzoru a parametry `Brightness` a `Color`, které určují intenzitu a barvu výsledného efektu. Ukázka tohoto shaderu je uvedena na Obrázku 3.11.



Obrázek 3.11: Shader magie využívající Voroného buňky

Obecný texturový shader

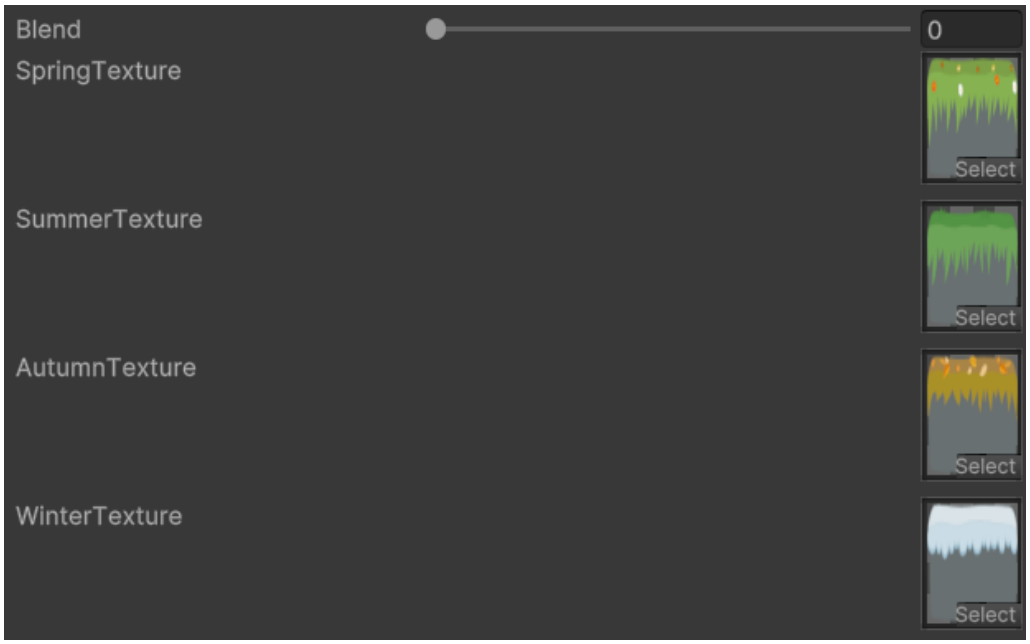
Jako poslední byl vytvořen texturový shader, jehož vizuální výstup je kompletně závislý na vstupní textuře. Pro tento účel byly vytvořeny speciální procedurální textury v nástroji Material Maker – například pro simulaci povrchů připomínajících kouř, zeminu nebo prachové vrstvy. Shader je navržen tak, aby jeho výsledný vzhled mohl být dále ovlivněn pomocí parametrů materiálu, jako je intenzita (tedy jak výrazně efekt zasahuje směrem ke středu obrazovky), průhlednost nebo barevný nádech. Díky tomu lze z jednoho shaderu snadno vytvořit více vizuálních variant bez nutnosti měnit samotný kód nebo strukturu shaderu. Dvě různé varianty výstupu tohoto shaderu jsou znázorněny na Obrázku 3.12.



Obrázek 3.12: Varianty výstupu texturového shaderu s použitím odlišných vstupních textur a parametrů

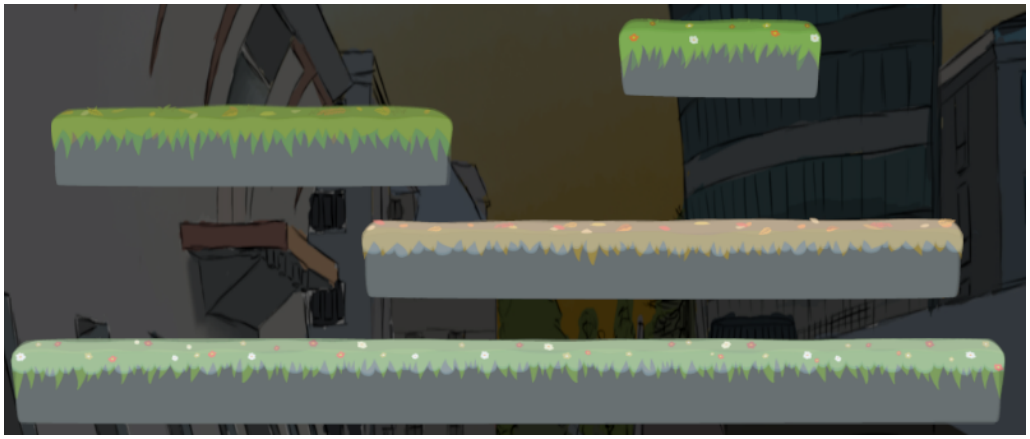
3.2.2 Shader pro změnu vizuálního stylu plošin

Dalším vytvořeným shaderem je shader pro plynulou vizuální změnu vzhledu plošin, po kterých se hráč ve hře pohybuje. Jeho vstupem jsou čtyři textury, z nichž každá reprezentuje vizuální podobu plošiny v jiném ročním období (jaro, léto, podzim, zima), a číselný parametr s `Blend`, jehož hodnoty se pohybují v rozsahu 0 až 1. Tento parametr určuje aktuální stav přechodu mezi jednotlivými obdobími. Vstupní hodnoty tohoto shaderu jsou znázorněny na Obrázku 3.13.



Obrázek 3.13: Vstupní hodnoty shaderu pro změnu ročního období

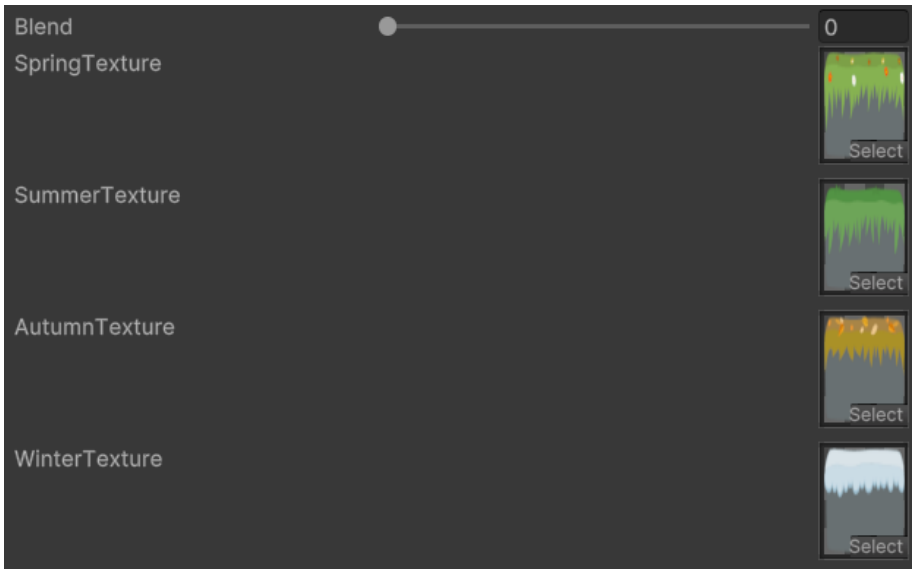
Rozsah hodnot `Blend` je rozdělen do pěti úseků, které zajišťují nejen přechody mezi čtyřmi ročními obdobími (jaro → léto → podzim → zima), ale i navazující návrat ze zimy zpět na jaro. První čtyři úseky slouží k plynulému přechodu mezi jednotlivými obdobími, pátý pak reprezentuje druhé jaro následující po zimě. Tento přechod je potřebný kvůli tomu, že není možné kontinuálně navázat hodnotu z konce rozsahu (1,0) zpět na začátek (0,0) bez skokové změny. Proto se po dokončení cyklu `Blend` jednorázově vrátí zpět na hodnotu odpovídající prvnímu jaru. Tímto způsobem je zajištěn nekonečný cyklus přechodů mezi ročními obdobími bez přerušení plynulosti přechodu. Ukázka výsledné podoby plošin během přechodu je uvedena na Obrázku 3.14.



Obrázek 3.14: Plošiny ve stádiu změny

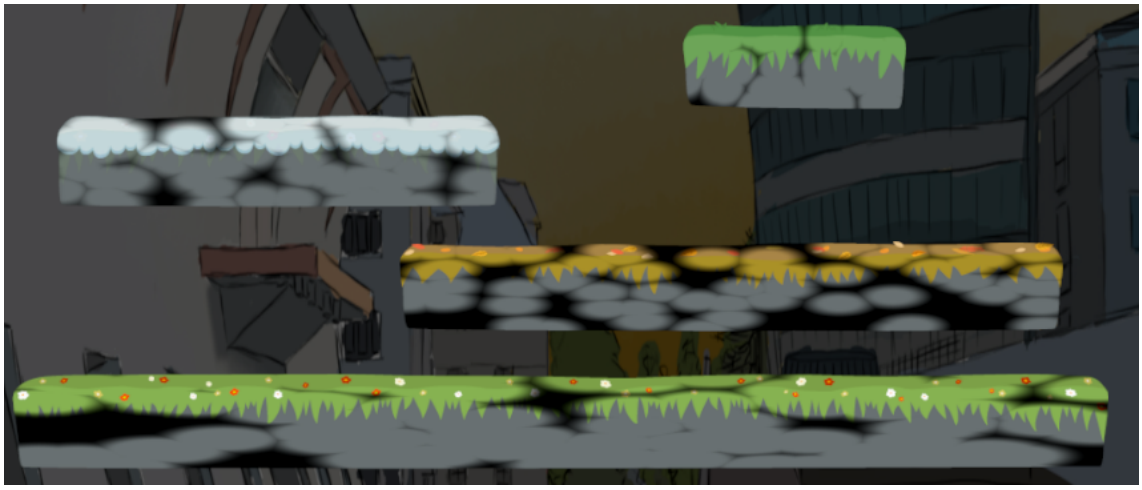
3.2.3 Shader pro praskající plošiny

Rozšířením předchozího shaderu je efekt praskání plošin, který je využit pro herní mechaniku zničitelných plošin. Tyto plošiny se chovají jako ty běžné (hráč se po nich může pohybovat), ale při kolizi dochází k vizuálnímu narušení jejich povrchu. Shader postupně mění vzhled plošiny podle jejího aktuálního stavu – s každou další kolizí se praskliny rozšiřují, až plošina nakonec zmizí. Po uplynutí určité doby se plošina znovu objeví v původním stavu, tedy bez prasklin. Díky vizuálnímu efektu praskání hráč snadno pozná, v jaké fázi se plošina nachází, a může tomu přizpůsobit svůj pohyb. Shader tak kromě estetické funkce plní i důležitou roli při čtení herní situace a orientaci v prostoru. Efekt je zároveň plně kompatibilní s vizuální stylizací jednotlivých ročních období. Parametry tohoto shaderu a nastavení jeho vstupních hodnot jsou znázorněny na Obrázku 3.15.



Obrázek 3.15: Vstupní hodnoty shaderu pro zobrazení prasklin plošin

Shader má všechny parametry předchozího shaderu pro změnu ročního období a je rozšířen o další vstupy, které ovlivňují vizuální podobu prasklin. Konkrétně se jedná o parametr barvy `BackgroundColor`, číselné parametry `CrackEdgeMin` a `CrackEdgeMax`, které určují rozsah a intenzitu prasklin, a vstupní texturu `CrackTexture`. Pro efekt prasklin byly využity Voroného diagramy vygenerované v Pythonu, které do shaderu vstupují jako textura. Kombinací těchto parametrů lze vizuálně kontrolovat nejen barvu, ale i tvar a rozsah poškození plošiny, a tím jasně indikovat její stav. Výsledná podoba vizuálního efektu praskání plošin je zobrazena na Obrázku 3.16. Velkou výhodou tohoto přístupu je, že celý efekt praskání je řešen čistě pomocí shaderu, bez nutnosti vytváření animačních klipů nebo komplexní logiky pro řízení průběhu animace.



Obrázek 3.16: Praskající platformy s aplikovanou texturou Voroného diagramu

3.3 Tvorba animací

Animace hrají klíčovou roli při vizuálním oživení herního světa a zprostředkování interakcí mezi hráčem a herním prostředím. V této kapitole jsou popsány různé přístupy k tvorbě animací využité v rámci této práce.

3.3.1 Procedurální animace

Procedurální animace nevyžadují práci s klasickými animovanými sprity ani s vícesnímkovými sekvencemi a většinou vycházejí pouze z jednoho základního obrázku nebo objektu. V Unity jejich největší výhodou bývá úspora výpočetních i paměťových nákladů a vysoká flexibilita – chování lze měnit skriptem v reálném čase bez nutnosti vytvářet složité animátory nebo předem připravené snímky.

Při použití klasické animace bylo nutné vytvořit a nakonfigurovat komponentu *Animator* [6], připojit k ní animaci vytvořenou v editoru (např. pomocí keyframe systému) a ručně určovat přechody mezi stavy. Naproti tomu procedurální přístup je zcela řízený kódem.

Nevýhodou procedurálních animací je jejich omezené využití – nejsou vhodné pro složitější pohyby jako je například chůze postavy, protože nepracují s předem definovanými fázemi pohybu. Jsou vhodné spíš pro jednoduché efekty jako pulsování, otáčení nebo změnu barvy, a proto jsou ve hře použity jen ve specifických případech, kde se tento typ animace dobře uplatní.

Animace zvednutelných předmětů

První z procedurálních animací byla využita pro sebratelné předměty, které se ve hře objevují. Cílem této animace je vytvořit jemný „plovoucí“ efekt, který zaujme hráče a zvýrazní sebratelné předměty ve scéně.

Každý předmět si při spuštění hry uloží svou výchozí pozici:

Každý předmět si při spuštění hry uloží svou výchozí pozici: `_startLocalPos = transform.localPosition;`.

Při každém snímku se pak vypočítá výškový posun pomocí funkce sinus: `float y = Mathf.Sin(Time.time * _frequency) * _amplitude;`.

Hodnota `Time.time` [55], kterou Unity poskytuje, udává, kolik sekund uplynulo od spuštění hry. Použitím této hodnoty v roli vstupu do funkce sinus vzniká plynulá a nekonečně opakující se vlna. Parametr `_frequency` určuje rychlost kmitání a `_amplitude` jeho výšku.

Nová pozice objektu se nastaví tak, že se ke svislé souřadnici výchozí pozice přičte aktuální výškový posun: `transform.localPosition = _startLocalPos + new Vector3(0, y, 0);`.

Díky tomu se objekt pravidelně pohybuje nahoru a dolů, zatímco zůstává na stejném místě v ose X a Z.

Animace papírového letadla

Další z procedurálních animací byla navržena pro nepřátelský objekt – papírové letadlo, které se ve scéně pohybuje tam a zpět mezi dvěma předem definovanými body. Během letu letadlo vykonává plynulý vertikální pohyb, jehož trajektorie je dána funkcí sinus: `float yOffset = Mathf.Sin(Time.time * _floatFrequency) * _floatAmplitude;`.

Parametr `_floatFrequency` určuje rychlost kmitání (tedy kolikrát za sekundu letadlo dokončí jednu vlnu), zatímco `_floatAmplitude` udává maximální výšku odchylky od původní výšky – tedy velikost výkyvu ve směru osy Y. Kombinací těchto parametrů vzniká efekt připomínající lehké klouzavé plachtění, typické pro papírové letadlo.

Současně je objekt v každém okamžiku mírně natočen kolem své osy Z podle funkce kosinus: `float rotationZ = Mathf.Cos(Time.time * _floatFrequency) * _maxRotationAngle * directionMultiplier;`.

Funkce kosinus zde odpovídá první derivaci sinu, a tedy udává okamžitou změnu výšky pohybu – čím prudší je změna směru, tím větší náklon. Parametr `_maxRotationAngle` definuje maximální možný úhel natočení letadla kolem osy Z. Hodnota `directionMultiplier`, která nabývá hodnoty ± 1 podle toho, na kterou

stranu letadlo právě letí, zajišťuje, že náklon respektuje směr pohybu (pomocí přepínání `SpriteRenderer.flipX` [50]).

O pohyb po ose X se stará metoda `Vector2.MoveTowards` [59], která je součástí Unity právě z důvodu plynulého pohybu k bodu ve scéně.

```
Vector2 basePosition = Vector2.MoveTowards(transform.position,
            _currentTarget, _speed * Time.deltaTime);
```

Metodě je předána aktuální pozice letadla `transform.position`, pozice objektu, ke kterému má letadlo „doletět“, a rychlost `_speed`. Tato metoda je volána každý snímek, což znamená, že rychlost letadla je také ovlivněna výkonem zařízení, na kterém je hra spuštěna (výkonnější zařízení budou dosahovat většího počtu snímků za sekundu). Protože se jedná o nežádoucí efekt, je rychlost `_speed` ještě vynásobena hodnotou `Time.deltaTime`, ve které engine uchovává, kolik sekund uběhlo od předchozího snímku. Tím je tento nežádoucí efekt odstraněn.

Jakmile letadlo dosáhne cílového bodu, dojde k přepnutí cílové pozice zpět na původní výchozí pozici, ze které letadlo původně vyletělo. Díky tomu se letadlo pohybuje zase zpět, čímž vzniká plynulý cyklus pohybu mezi dvěma body.

Animace nápisu v menu

Další procedurální animace byla použita v hlavním menu hry, kde slouží k jemnému pulsování názvu hry – tedy k efektu plynulého zmenšování a zvětšování textového objektu. Původně byla tato animace řešena pomocí klasické Unity animace (*Animator* a animace klíčových snímků), avšak kvůli snížení výpočetní náročnosti byla převedena do skriptové podoby.

Samotný efekt je dosažen periodickou změnou velikosti objektu pomocí funkce sinus:

```
float scale = 1f + Mathf.Sin(Time.time * _pulseSpeed) * _pulseAmount;
transform.localScale = _originalScale * scale;
```

Hodnota `_pulseSpeed` určuje rychlost oscilace a `_pulseAmount` maximální rozsah zvětšení nebo zmenšení. Výsledkem je vizuálně příjemný a nenápadný pohyb, který přitahuje pozornost hráče bez nutnosti využívat komponentu *Animator* nebo složité animace, čímž se zjednodušuje systém a snižuje režie vykreslování v hlavním menu.

3.3.2 Keyframe animace

Jako ukázka použití keyframe animace byla vytvořena jednoduchá sekvence animovaného křížku. Tento typ animace byl realizován ve formě frame-by-frame postupu pomocí časové osy v prostředí Adobe Photoshop.

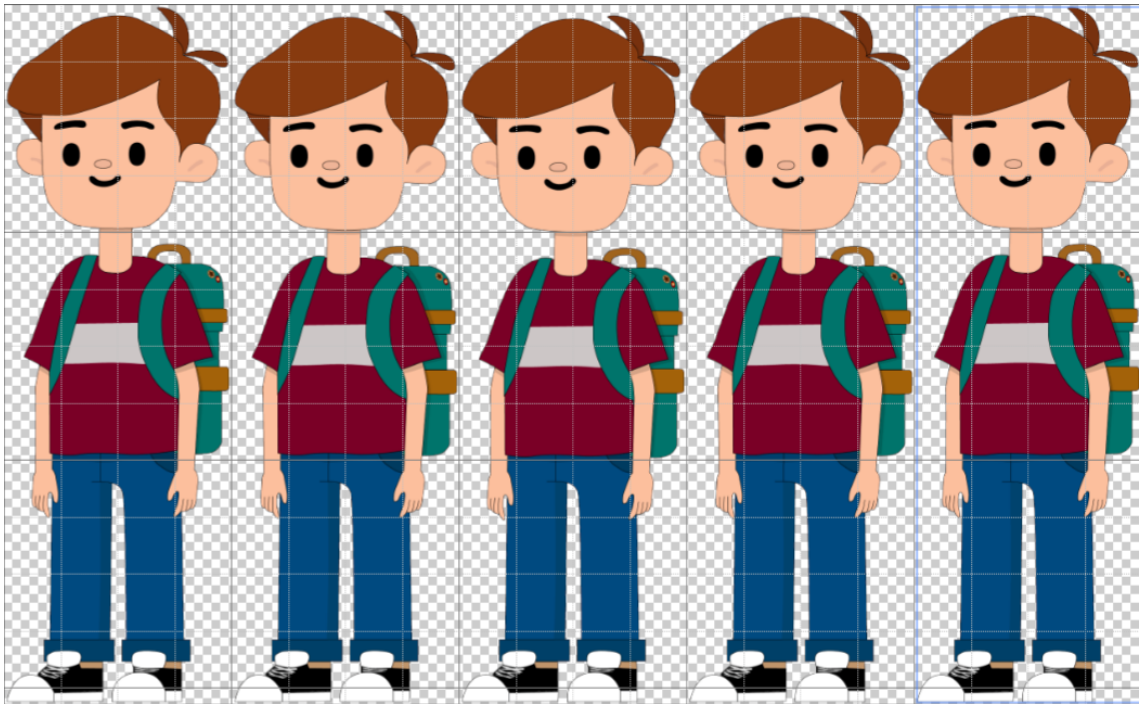
V jednotlivých snímcích byly postupně vykresleny dílčí části animace – nejprve se objevila jedna část křížku a následně byla doplněna druhá diagonála. Všechny

tyto kroky byly vytvářeny manuálně pomocí kreslícího nástroje, přičemž každý frame představoval samostatnou vrstvu nebo skupinu vrstev, které byly aktivovány v konkrétních časových bodech.

Po dokončení byla animace exportována jako sprite sheet, ve kterém jsou všechny snímky animace umístěny vedle sebe v jedné řadě. Výsledný sprite sheet s jednotlivými fázemi animace je zobrazen na Obrázku 3.17. Tento výstup byl následně připraven k importu do herního enginu Unity, kde může být použit jako sériová animace pomocí *Sprite Animatoru*.

Obrázek 3.17: Sprite sheet animovaného křížku vytvořený pomocí keyframe animace

Podobným způsobem byla vytvořena i klidová (tzv. idle) animace hlavní postavy, při které byly jednotlivé pohyby navrženy jako samostatné snímky a následně sestaveny do plynulé smyčky. Idle animace se používá v momentech, kdy hráč postavou nehýbe – postava stojí na místě, ale přesto nevypadá staticky. Typicky jde o drobné pohyby, které navozují přirozený dojem, že postava dýchá nebo vnímá okolí. V tomto případě šlo o velmi jemné změny, které jsou na jednotlivých snímcích téměř nepostřehnutelné, ale při přehrání animace vytvářejí efekt lehkého pohybu. Snímky použité pro tuto animaci jsou zobrazeny na Obrázku 3.18.



Obrázek 3.18: Sekvence snímků pro idle animaci

3.3.3 Rigging

Pro potřeby animace hlavní postavy byla v Unity použita technika 2D skeletal rigging, která umožňuje animaci pomocí kostí přiřazených k částem spritu. Tento přístup je vhodný zejména při animaci postav složených z více samostatných segmentů (např. končetiny, trup, hlava), které je možné rozpohybovat a deformovat nezávisle pomocí kostí.

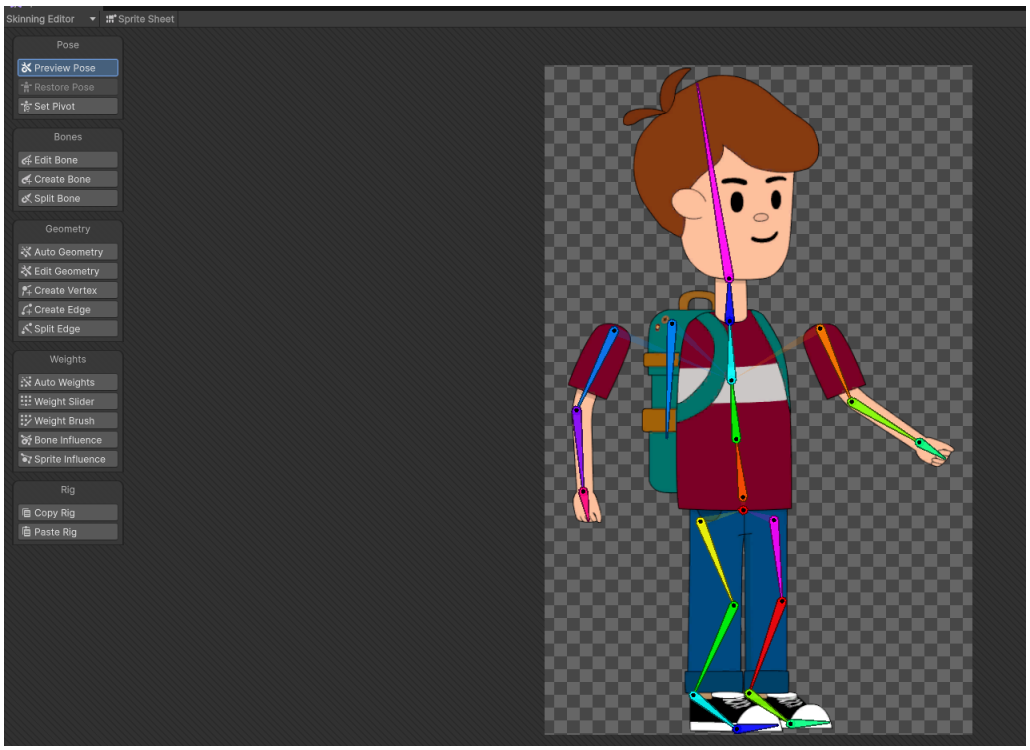
Průběh riggování postavy:

1. **Vytvoření kostry**

   Jako první krok byla ve *Sprite Editoru* [48] pomocí nástroje *Create Bone* vytvořena kompletní kostra postavy. Kostra byla navržena tak, aby odrážela přirozenou kloubovou strukturu těla – zahrnuje páteř, končetiny, hlavu a další části, které jsou hierarchicky propojené.

   Každá kost byla umisťována manuálně tak, aby odpovídala logickým bodům ohybu (např. loket, koleno, kotník), a tvořila tak základ pro následné animace. V tomto kroku bylo důležité dbát na to, aby byly kosti navázány v přirozeném sledu a správném směru, což výrazně ovlivňuje výsledný pohyb při deformaci.

   Výsledné rozložení a napojení kostí je znázorněno na Obrázku 3.19.



Obrázek 3.19: Vizualizace kompletní kostry

2. **Generování geometrie a vah**

   Po definování kostí byla použita funkce *Auto Geometry*, která automaticky

vygenerovala trojúhelníkovou síť (tzv. triangulaci) pokrývající vizuální oblast sprite postavy.

Tato síť se skládá z vrcholů (též uzlů), které jsou propojeny hranami a tvoří základ pro deformaci při pohybu kostí. Každému vrcholu byla následně přiřazena výchozí váha podle jeho vzdálenosti od jednotlivých kostí – tento krok určuje, jak silně která kost ovlivňuje daný bod obrazce. Tyto automaticky přiřazené váhy sloužily pouze jako základ pro další úpravy.

3. **Úprava vertexů**

   Pomocí nástroje *Edit Geometry* byly upraveny jednotlivé vertexy tak, aby lépe kopírovaly obrys postavy a zároveň umožňovaly přirozenější pohyb při rotaci kostí. Tato fáze je důležitá pro minimalizaci deformací postavy během animace.

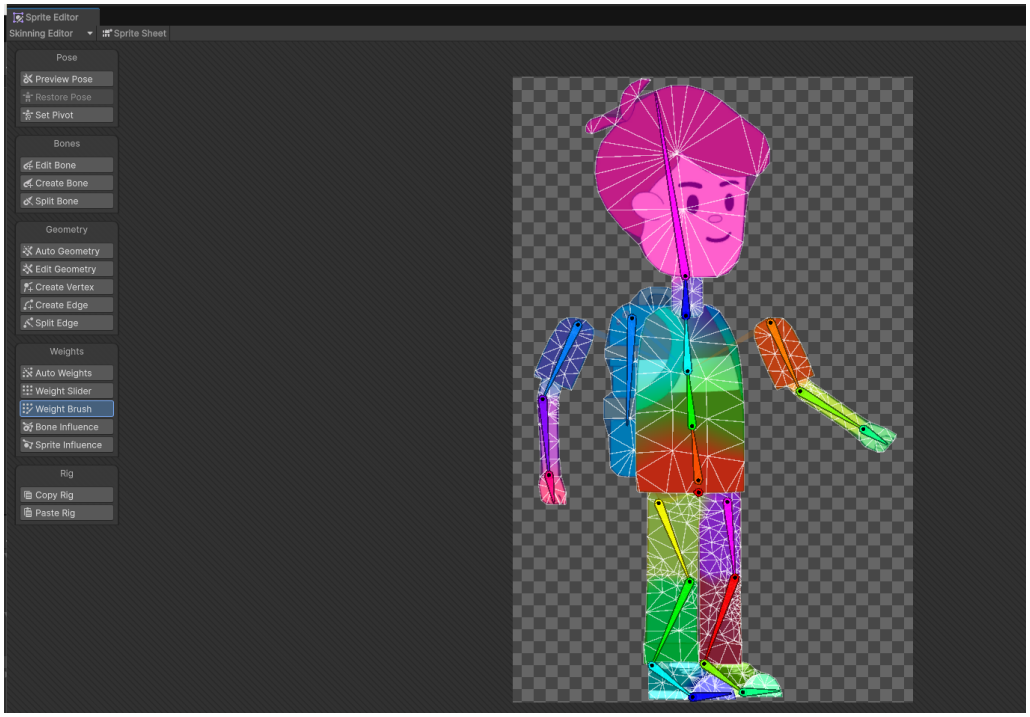
4. **Výběr ovlivňujících kostí**

   Následně byly v rámci nástroje *Bone Influence* pro každou část postavy určeny konkrétní kosti, které ji mohou ovlivňovat. Tento krok slouží k omezení vlivu vzdálených nebo nesouvisejících kostí – například k tomu, aby geometrii paže ovlivňovaly pouze kosti paže a ne například kosti dolních končetin.

   Tím se zajišťuje, že při pohybu nohou nedojde k nechtěnému protažení nebo deformaci částí, které s tímto pohybem nesouvisejí.

5. **Úprava vah**

   Pomocí nástroje *Weight Brush* byly manuálně upraveny váhy přiřazené jednotlivým vrcholům (vertexům) sítě. Tyto váhy určují, do jaké míry je daný vrchol ovlivňován pohybem konkrétní kosti – čím vyšší váha (a tedy i sytější barva příslušné kosti na vizualizaci), tím více se vrchol při animaci deformuje. Tento princip doplňuje hierarchii „parent–child“ mezi kostmi tím, že umožňuje jemně řídit deformaci povrchu postavy nezávisle na tom, zda je kost přímo nadřazená nebo podřízená jiné. Například oblast ramene je ovlivňována jak kostí paže, tak i kostí trupu – právě váhování umožňuje určit poměr tohoto ovlivnění a zajistit tak plynulý a realistický pohyb. Barevné zobrazení váhového rozložení v rámci postavy je znázorněno na Obrázku 3.20.



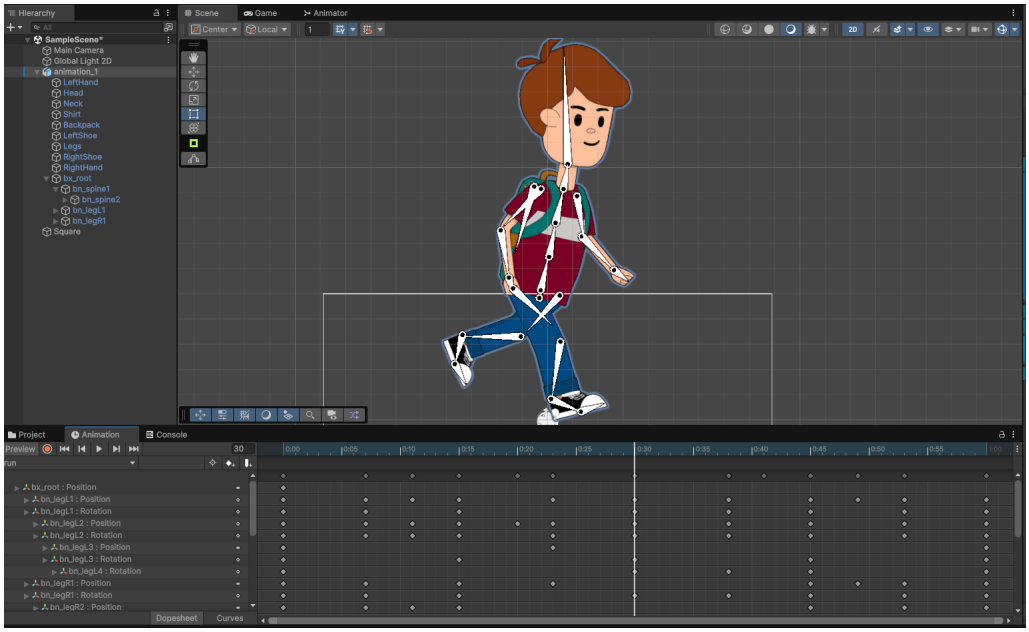
Obrázek 3.20: Barevné zobrazení váhového rozložení vertexů podle vlivu jednotlivých kostí.

6. **Tvorba animací**

   V Unity Animatoru byla vytvořena sada základních pohybových stavů:

   - *idle* – neutrální póza v klidu,
   - *walk* – chůze,
   - *run* – běh,
   - *jump* – výskok,
   - *apex* – fáze ve vrcholu výskoku,
   - *fall* – pád po výskoku,
   - *recovery* – dopad na zem.

   Tyto animace byly vytvořeny pomocí keyframe systému – pohyb kostí byl zaznamenán na časovou osu a interpolován mezi jednotlivými snímky. Tento způsob umožňuje plnou kontrolu nad každým pohybem a zároveň poskytuje jednoduchý způsob přechodu mezi stavy. Ukázka editace jedné z těchto animací v prostředí Unity Animator je zobrazena na Obrázku 3.21.



Obrázek 3.21: Ukázka animace běhu postavy pomocí keyframe systému.

4 Implementace grafiky, shaderů a animací ve vybrané hře

Tato kapitola popisuje, jak byly vytvořené grafické prvky, shadery a animace integrovány do finální podoby hry. Zaměřuje se na praktické využití jednotlivých komponent v herním enginu Unity a jejich propojení s herní logikou.

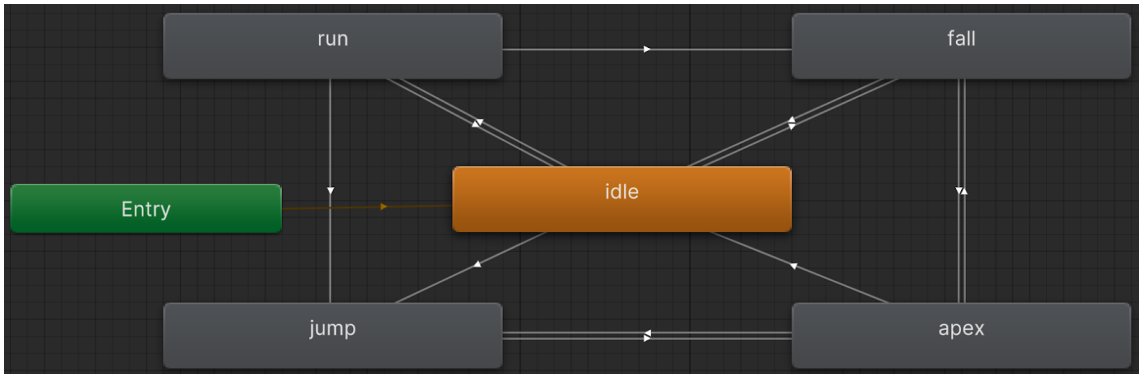
4.1 Animace hlavní postavy

Animace hlavní postavy byla ve hře řešena pomocí komponenty *Animator*, která slouží jako stavový automat pro řízení přechodů mezi jednotlivými animačními stavy. Přechody mezi animacemi jsou řízeny třemi parametry:

- `IsRunning` - indikuje, zda je hráč ve stavu běhu
- `VerticalVelocity` - aktuální vertikální rychlost postavy
- `IsGrounded` - označuje, zda je hráč v kontaktu se zemí (např. plošinou)

Pro porovnávání vertikální rychlosti se místo nuly používají hodnoty 0,01 a -0,01, které eliminují drobné odchylky způsobené nepřesnostmi výpočtů fyzikálního enginu.

Struktura stavového automatu v Unity Animatoru, která řídí přechody mezi jednotlivými animačními stavy postavy, je znázorněna na Obrázku 4.1.



Obrázek 4.1: Schéma stavového automatu v Unity Animatoru

Animator pracuje s pěti animačními stavy, mezi nimiž se postava přirozeně přepíná v závislosti na herním kontextu:

- Idle: výchozí (klidová) animace. Přechází do:
  - Run, pokud je `IsRunning` nastaveno na `true`
  - Fall, pokud je `VerticalVelocity` < -0,01
  - Jump, pokud je `VerticalVelocity` > 0,01
- Run: animace běhu. Přechází do:
  - Fall, pokud `VerticalVelocity` < -0,01
  - Idle, pokud je `IsRunning false` a zároveň `VerticalVelocity` se pohybuje v rozmezí -0,01 až 0,01
  - Jump, pokud `VerticalVelocity` > 0,01
- Jump: animace výskoku. Přechází do:
  - Apex, jakmile `VerticalVelocity` klesne pod 3.
- Apex: stav mezi výskokem a pádem. Přechází do:
  - do Jump, pokud `VerticalVelocity` znovu překročí hodnotu 3
  - do Idle, pokud je `IsGrounded` true a zároveň `VerticalVelocity` je mezi -0,01 a 0,01
  - do Fall, pokud `VerticalVelocity` < -3.
- Fall: animace pádu. Přechází do:
  - Idle, pokud je `IsGrounded true` a zároveň `VerticalVelocity` je mezi -0,01 a 0,01
  - Apex, pokud `VerticalVelocity` překročí hodnotu -3

4.2 Změna ročního období

Součástí této práce je vizuální změna ročních období. Základ tvoří metoda `NextSeason`, která spouští celkový přechod do nového ročního období. Ve hře je tato metoda zavolána, když mikrofon detekuje hlasitý zvuk. Metoda `NextSeason` zároveň spustí událost `OnSeasonChangeStarted`, na kterou reagují prvky, které mění své chování na základě aktuálního ročního období (např. papírová letadla jsou v létě rychlejší). Přepínání období probíhá sekvenčně a cyklicky, což znamená, že není možné přeskočit např. z jara rovnou na zimu.

4.2.1 Spuštění události

Po aktivaci změny ročního období se spustí pohyb jednoho z dvojice *Edge Collider 2D* [12] (komponenta tvořená jednou nebo více propojenými úsečkami detekující kolize). Tyto collidery jsou nastaveny jako *triggery* [10] (tj. neovlivňují fyziku, ale spouští události při průchodu jiných objektů) a jeden z nich se vždy pohybuje po ose X nebo Y, střídavě podle předchozího směru. Rychlost pohybu závisí na zvolené ose. Cílem je, aby collider projel celou scénou – z výchozí pozice až do bodu určeného objektem `ColliderEndPoint`, který je umístěn v pravém dolním rohu herní úrovně. Díky tomu se změna dotkne všech relevantních objektů, které na tuto událost reagují.

4.2.2 ISeasonChange

Rozhraní `ISeasonChange` implementují všechny objekty ve scéně, které mají reagovat na změnu ročního období. Umožňuje vytvořit jednotné rozhraní pro přepínání vizuálních a funkčních stavů napříč různými komponentami bez nutnosti znát jejich konkrétní implementaci. Díky tomu může `SeasonsManager` předávat informaci o změně období libovolnému objektu, který toto rozhraní implementuje, a ten si sám interně určí, jakým způsobem bude na tuto změnu reagovat. Tento přístup zajišťuje rozšiřitelnost a modularitu celé mechaniky.

```
public interface ISeasonChange
{
    void AnimateToSeason(SeasonsManager.Season season);
}
```

4.2.3 Spuštění změny

Jakmile se pohybující `collider` dostane do kontaktu s objektem ve scéně, Unity automaticky volá metodu `OnTriggerEnter2D` [31]. Tato metoda prochází všechny komponenty objektu, který byl zasažen, a hledá ty, které implementují rozhraní `ISeasonChange`. Pokud takovou komponentu najde, zavolá její metodu `AnimateToSeason`, čímž dojde k vizuální změně podle aktuálního ročního období.

Jedním z příkladů objektu reagujícího na tuto událost je komponenta `SeasonMaterialController`. Ta obstarává vizuální změnu vzhledu plošin ve hře na základě aktuálního období. Každé období odpovídá jiné hodnotě parametru `Blend` v materiálu plošiny. Hodnoty 1,0; 0,25; 0,5 a 0,8 reprezentují postupně jaro, léto, podzim a zimu. Tato hodnota se mění plynulým přechodem, čímž je zajištěna vizuálně jemná změna.

Na konci přechodu ze zimy na jaro dochází ke speciální úpravě: pokud byl parametr `Blend` změněn zpět na hodnotu 1,0 (odpovídající jaru), je okamžitě nastaven na 0,0, aby bylo možné cyklus změn plynule opakovat. Tím je zajištěno, že shader zůstává vizuálně konzistentní a umožňuje hladký nekonečný přechod mezi všemi čtyřmi obdobími bez znatelných skoků.

4.3 Přepínání celoobrazových shaderů

Ve hře byly použity různé celoobrazové shadery, které slouží k vytvoření globálních vizuálních efektů aplikovaných přímo na výstup kamery [8] („oči“ hráče). Pro jejich správu byl vytvořen dedikovaný systém `FullScreenShaderManager`, který umožňuje efektivní přepínání mezi jednotlivými efekty v reálném čase, a to bez nutnosti zásahu do vykreslovací pipeline během běhu hry.

Každý shader je reprezentován výčtem `FullScreenShader` a přiřazen ke konkrétnímu materiálu v seznamu `ShaderList`. Při spuštění hry se tento seznam převede do slovníku, což zajišťuje rychlý přístup k požadovanému materiálu. Přepnutí shaderu je prováděno přiřazením příslušného materiálu do `passMaterial` komponenty *Full Screen Pass Renderer Feature*, která je součástí render pipeline URP. Ta je vyhledána přímo v seznamu aktivních renderer feature pomocí jejího názvu.

Pro přepínání mezi efekty slouží veřejná metoda `SwitchToShader`, která jako parametr přijímá konkrétní hodnotu z výčtu `FullScreenShader`. Na základě této hodnoty následně nastaví odpovídající materiál jako aktivní pro komponentu *Full Screen Pass Renderer Feature*. Díky tomu lze měnit celoobrazové efekty dynamicky během hry – například při změně ročního období, nebo při aktivaci magické schopnosti. Celý systém je navržen tak, aby byl snadno rozšiřitelný o další efekty bez potřeby zásahů do samotné render pipeline.

4.4 CRT Efekt

Filtr simulující vzhled starých CRT monitorů byl v projektu realizován pomocí komponenty *Global Volume* [60], která umožňuje globální aplikaci postprocessingových efektů na celý výstup kamery. Tento filtr obsahuje několik aktivních efektů, které dohromady vytvářejí typický retro vzhled:

- *Bloom* – zajišťuje záři jasných oblastí s intenzitou nastavenou na 1 a prahem 0,8, což zvýrazňuje světlé plochy.
- *Panini Projection* – přidává mírné zkreslení perspektivy pomocí parametru `Distance` (0,1), čímž simuluje lehké zakřivení obrazu.
- *Vignette* – ztmavuje okraje obrazovky s černou barvou a intenzitou 0,087.
- *Chromatic Aberration* – způsobuje mírné barevné posuny na okrajích objektů s intenzitou 0,1, což je běžný vizuální efekt u CRT zařízení.
- *Film Grain* – do obrazu přidává jemný šum pomocí vlastní textury s horizontálním rastrem, která imituje linkovou strukturu CRT monitorů. Intenzita šumu je nastavena na 0,05.

Tato kombinace efektů dohromady vytváří stylizovaný CRT filtr, který vizuálně sjednocuje scénu a podporuje estetiku připomínající starší zobrazovací technologie. Hodnoty parametrů byly zvoleny experimentálně tak, aby byl efekt patrný, ale nerušil ostatní grafické prvky hry.

4.4.1 Textura šumu

Textura použitá pro efekt *Film Grain* ve filtru CRT byla vytvořena ručně pomocí knihovny Turtle v jazyce Python. Cílem bylo vygenerovat pravidelný horizontální rastr, který věrně napodobuje vzhled obrazu na starých CRT monitorech.

Pomocí jednoduchého skriptu (viz Příloha D) se nejprve nastavil výstupní formát (rozlišení 1920×1080) a následně byly ve smyčce vykresleny černé vodorovné linky po celé ploše obrazu s rozestupem 8 pixelů. Každá čára začíná vlevo a končí vpravo, čímž vzniká hustý a pravidelný vzor. Výsledný výstup byl uložen ve formátu EPS a následně převeden do textury použitelné v Unity jako vstupní mapa pro efekt šumu.

Tento přístup umožnil naprostou kontrolu nad vzhledem textury a zároveň garantoval konzistentní a stylizovaný výstup, který nenarušuje ostatní vizuální efekty použitých shaderů.

4.5 Efekt padání listí a sněžení

Pro vizuální doplnění změny ročního období byly vytvořeny dva *Particle Systemy* [34] (systémy částic sloužící k vykreslování efektů jako je sníh nebo déšť), které se dynamicky aktivují na základě aktuálního ročního období. První z nich využívá vygenerovanou Kochovu vločku jako texturu pro částice sněhu (využití při sněžení v zimě). Druhý systém používá vlastnoručně nakreslené obrázky listů, které byly vytvořeny ručně. Tento systém se aktivuje na podzim a simuluje padání listí ve scéně. Ukázka efektu podzimního listí je zobrazena na Obrázku 4.2.

Obrázek 4.2: Efekt padajícího listí

Oba *Particle Systemy* jsou propojeny se systémem ročních období ve hře, dí-

ky čemuž dochází k jejich automatickému zapínání a vypínání při přechodu mezi sezónami. Ukázka efektu sněžení je uvedena na Obrázku 4.3.

Obrázek 4.3: Efekt sněžení

5 Porovnání použitých nástrojů z hlediska náročnosti a efektivity

Cílem této kapitoly je zhodnotit různé nástroje a přístupy použité při tvorbě grafiky, shaderů a animací v rámci této práce. Jednotlivé metody jsou porovnány z hlediska časové a technické náročnosti, flexibility při úpravách a vhodnosti pro konkrétní typy grafického obsahu.

5.1 Metody tvorby grafiky - Výhody a nevýhody Adobe Illustratoru, Pythonu, Material Makeru a AI

Adobe Illustrator
Výhody:

- Vektorová grafika umožňuje vytváření ostrých a technicky přesných tvarů, které lze libovolně škálovat bez ztráty kvality. To je výhodné při tvorbě assetů, které se zobrazují v různých rozlišeních nebo velikostech.
- Ruční kreslení pomocí grafického tabletu v kombinaci s následným vektorovým zpracováním poskytuje úplnou kontrolu nad každým detailem. Tento způsob je ideální zejména pro návrh vizuálně charakteristických prvků, jako jsou postavy nebo interaktivní objekty.
- Illustrator umožňuje přesnou práci s barvami, geometrií a kompozicí. Díky tomu je snadné zajistit jednotný vzhled celé hry.

Nevýhody:

- Každý objekt musí být nejprve navržen, ručně překreslen a vizuálně doladěn. Při větším počtu assetů se může tento proces stát časově náročným.
- Adobe Illustrator není určen pro hromadné generování variací. V případě, že je potřeba vytvořit množství podobných prvků, je nutné je zpracovat manuálně.
- Software je součástí předplatného Adobe Creative Cloud, což může být finančně náročnější zejména pro studenty, nezávislé vývojáře nebo malé týmy.

Python

Výhody:

- Pomocí Pythonu je možné generovat komplexní vizuální vzory, jako například Voroného diagramy, fraktály nebo pravidelně se opakující geometrické vzory. Vstupní parametry lze snadno měnit, což umožňuje rychlou tvorbu různých variant bez nutnosti manuálního zásahu.
- Generované obrázky nebo struktury lze exportovat do bitmapových či vektorových formátů a následně použít jako vstupní textury pro shadery nebo jako assety přímo v herním enginu.

Nevýhody:

- Používání Pythonu k generování grafiky vyžaduje určité znalosti programování, orientaci v datových strukturách a schopnost pracovat se základními knihovnami.
- Mnohé algoritmy používané při generování vyžadují schopnost matematicky pochopit strukturu vizuálního výstupu. Bez určitého prostorového nebo analytického myšlení může být předvídání výsledků či jejich ladění problematické.
- Procedurální grafika často působí uměle nebo matematicky pravidelně, protože vychází z algoritmů a nikoli z ruční kresby. I když některé algoritmy (např. Perlinův šum) umožňují vytvářet přirozeně působící vzory, výsledný vizuál může být méně vhodný pro hry, které kladou důraz na stylizovanou kresbu nebo specifickou estetiku prostředí a postav.

Material Maker

Výhody:

- Material Maker umožňuje tvorbu textur pomocí uzlového systému, kde se jednotlivé funkce propojují graficky. To výrazně usnadňuje práci designérům a grafikům, kteří nemají programátorské dovednosti.
- Vestavěné uzly jako *Sine Wave*, *Voronoi* nebo různé generátory šumu umožňují dynamickou manipulaci s výsledným výstupem. Uživatel tak může měnit frekvence, míru kontrastu, orientaci vzorů a další vlastnosti, čímž získá vizuální variabilitu bez nutnosti zásahu do zdrojového kódu.
- Vygenerované textury je možné přímo exportovat do obrazových formátů jako PNG nebo je převést do GLSL kódu.

Nevýhody:

- Při tvorbě složitějších algoritmů nebo interaktivních prvků může být uzlový přístup omezující. Ve srovnání s programováním v jazycích jako Python nebo HLSL nabízí výrazně menší kontrolu nad výpočetním procesem, což ztěžuje implementaci komplexního chování materiálů nebo pokročilých vizuálních efektů.
- U větších nebo složitějších textur se může uzlová struktura rychle rozrůst do komplikované sítě, která je pak obtížnější na údržbu a opravy v případě chyb.

Leonardo AI + generatívna AI

Výhody:

- Generativní AI umožňuje během několika minut vytvořit vizuálně komplexní obrazy, jako jsou pozadí, koncepty prostředí nebo celé scény. Tento přístup výrazně zkracuje čas potřebný na produkci, zejména v počátečních fázích návrhu nebo při přípravě grafických podkladů.
- AI generované výstupy jsou vhodné zejména pro části hry, které nejsou přímo interaktivní nebo neobsahují prvky ovlivňující hratelnost.
- V kombinaci s jazykovým modelem, který pomáhá formulovat kvalitní textové popisy (prompty), je možné efektivně ovlivnit výtvarný styl generovaných scén.

Nevýhody:

- Navzdory pokroku v generativní AI není výsledek vždy zcela předvídatelný. Výstup často přesně neodpovídá původnímu zadání a je nutné prompt opakovaně upravovat.
- Použití AI generovaného obsahu může být z pohledu autorských práv problematické. Není vždy jasné, kdo je vlastníkem výstupu a zda byl trénovací dataset sestaven v souladu s legislativou, což může komplikovat komerční využití.
- Výstupy z AI jsou nejčastěji bitmapové obrázky, které nejsou optimalizované pro animaci, shaderové zpracování nebo dynamickou manipulaci.

Nástroj	Výhody	Nevýhody
Adobe Illustrator	Ostrá, škálovatelná vektorová grafika; úplná umělecká kontrola; konzistentní vizuální styl	Časová náročnost; manuální tvorba variací; vyšší licenční náklady
Python	Automatizované generování vzorů; možnost exportu do různých formátů; vhodné pro shadery	Vyžaduje programování a schopnost porozumět algoritmům; méně vhodné pro výtvarný styl
Material Maker	Jednoduché vizuální programování; parametrizovatelné uzly; export do PNG/-GLSL	Omezená kontrola při komplexních efektech; uzlové sítě mohou být nepřehledné
Leonardo AI + generativní AI	Velmi rychlá produkce obrázků; vhodné pro neinteraktivní scény; možnost ovlivnit styl	Nepředvídatelnost výstupů; právní nejistoty; výstupy nevhodné pro technickou integraci

Tabulka 5.1: Srovnání nástrojů pro generování grafiky z hlediska výhod a nevýhod.

5.2 Metody tvorby shaderů

Shader Graph byl v této práci zvolen zejména kvůli své přímé integraci do Unity, která umožňuje pohodlné propojení s editorem a renderovacím systémem bez potřeby externích nástrojů. Díky tomu bylo možné shadery snadno přiřazovat materiálům, využívat je v rámci Universal Render Pipeline a efektivně je nasazovat jak na objekty, tak jako celoobrazové efekty přes *Full Screen Pass Renderer Feature*.

Významnou výhodou Shader Graphu je možnost snadno měnit hodnoty parametrů materiálů v kódu – například parametr `Blend` pro plynulý přechod mezi ročními obdobími nebo vstupní hodnoty pro crack efekt zničitelných plošin. Tento způsob práce umožnil vytvářet vizuálně výrazné, ale zároveň interaktivní efekty bez nutnosti psát vlastní animační logiku nebo upravovat shadery mezi jednotlivými fázemi hry.

Na druhou stranu má Shader Graph i svá omezení. V některých případech naráží na limity dané systémem uzlů, například při snaze o složitější matematické operace, precizní optimalizace nebo pokud by bylo třeba využít specifické techniky, které standardní uzly Shader Graphu nenabízejí. Klasický postup psaní shaderů v HLSL tak zůstává vhodnější volbou v případech, kde je třeba maximální kontrola nad výkonem a přesným chováním shaderu.

Pro potřeby této práce ale Shader Graph nabídl ideální kompromis mezi rychlým návrhem, integrací do enginu a možností manipulace s parametry z kódu. Byly v něm vytvořeny všechny použité shadery a výsledky naplnily jak vizuální, tak funkční cíle stanovené pro danou hru.

5.3 Metody tvorby animací

Procedurální animace

Výhody:

- Procedurální animace mají nízké nároky na paměť, protože nevyžadují vícesnímkové sprity ani složité animační klipy.
- Díky řízení animace skriptem lze v reálném čase upravovat parametry jako frekvence nebo amplituda, bez nutnosti měnit animace v editoru.
- Jsou velmi efektivní při použití pro jednoduché vizuální efekty, například pulsaci, levitaci či rotaci.

Nevýhody:

- Tento typ animací je omezený v případě komplexních pohybů, jako je chůze či běh postav, které vyžadují detailní kontrolu nad fázemi pohybu.
- Vyžaduje schopnost programování, protože celkové chování objektů je řízené výhradně kódem a nelze ho intuitivně nastavovat v editoru.
- Vizuální výsledek je nutné často ladit experimentálně, protože ne vždy odpovídá očekávání.

Keyframe animace

Výhody:

- Poskytuje plnou vizuální kontrolu, protože každý snímek je ručně definovaný, což umožňuje dosáhnout vysoké úrovně stylizace a výrazu.
- Implementace do herního enginu je poměrně jednoduchá.

Nevýhody:

- Tvorba je časově náročná, protože každá fáze pohybu musí být samostatně nakreslená, což prodlužuje vývoj.
- Větší počet snímků zvyšuje velikost assetů a tím i paměťovou náročnost celého projektu.
- Jakákoliv změna animace vyžaduje úpravu zdrojových obrázků a opětovný export, což snižuje flexibilitu při úpravách.

Rigging a skeletal animace

Výhody:

- Skeletová animace je velmi efektivní pro postavy, protože umožňuje realistické pohyby bez potřeby velkého množství spriteů.
- Díky tomu, že animace pracují s deformací jednoho spritu pomocí kostí, je výsledná paměťová stopa výrazně nižší.
- V Unity lze rigging snadno kombinovat se stavovým automatem pomocí Animatoru, což umožňuje přirozené přechody mezi jednotlivými animačními stavy.

Nevýhody:

- Na začátku je potřeba vyšší časová investice – vytvoření kostry, definice vah a správné hierarchie vyžaduje pečlivou přípravu.
- Při špatném nastavení váhování vrcholů (uzlů) může docházet k vizuálním deformacím, které narušují plynulost animace.

Závěr

Cílem této bakalářské práce bylo prozkoumat možnosti využití shaderů a procedurálního generování 2D grafiky v herním enginu Unity. V teoretické části byla analyzována problematika tvorby 2D grafiky, její projekční metody, nástroje pro tvorbu vizuálních prvků a specifika implementace shaderů v prostředí Unity. Zvláštní pozornost byla věnována procedurálním technikám, které představují efektivní způsob tvorby opakovatelného a zároveň vizuálně rozmanitého obsahu.

V praktické části byly tyto poznatky aplikovány při vývoji konkrétních shaderů, textur a animací pro vybraný 2D herní projekt. Byly implementovány vizuální efekty jako simulace ročních období, efekty praskání plošin či celoplošné shadery ovlivňující výstup kamery. Pro tvorbu grafických prvků byl zvolen hybridní přístup: kombinace ruční kresby ve vektorovém formátu (Adobe Illustrator), algoritmicky generované grafiky (Python, Material Maker) a umělá inteligence (Leonardo AI). Tato rozmanitost nástrojů umožnila efektivně propojit estetické a technické aspekty vývoje.

Výsledky práce ukazují, že využití shaderů a procedurálního generování ve 2D hrách významně přispívá k vizuální rozmanitosti, optimalizaci výkonu a snížení nároků na paměťové zdroje. Zároveň však vyžaduje hlubší znalost renderovacích technologií a zpracování grafických dat v reálném čase. Pro vývojáře tato kombinace představuje silný nástroj, zejména při vývoji her s vysokými nároky na vizuální interaktivitu a estetiku při zachování technické efektivity.

Pro prezentaci výstupů práce byl navíc vytvořen samostatný projekt v Unity, který slouží jako interaktivní galerie, ve které si lze prohlédnout jednotlivé dosažené výsledky. Projekt je volně dostupný v repozitáři na GitHubu ve formě stáhnutelného ZIP archivu, který obsahuje spustitelný soubor: https://github.com/nikkyjetop/BP-Strecanska/releases.

Použitá literatura

[1] *2D game creation workflow* [online]. [cit. 2025-02-05]. Dostupné z: https://docs.unity3d.com/6000.0/Documentation/Manual/2d-game-creation-wokflow.html%5C#Sprites.

[2] *About Shader Graph* [online]. [cit. 2025-04-21]. Dostupné z: https://docs.unity3d.com/Packages/com.unity.shadergraph@17.2/manual/index.html.

[3] *Adobe Animate* [online]. 2025. [cit. 2025-02-19]. Dostupné z: https://www.adobe.com/products/animate.html.

[4] *Adobe Photoshop* [online]. 2025. [cit. 2025-02-17]. Dostupné z: https://www.adobe.com/products/photoshop.html.

[5] AKENINE-MÖLLER, Tomas et al. *Real-time rendering*. Fourth edition. Boca Raton: CRC Press/Taylor & Francis Group, [2018]. ISBN 978-1-138-62700-0.

[6] *Animator*. Dostupné také z: https://docs.unity3d.com/ScriptReference/Animator.html.

[7] BORKE, Paul. The Mandelbrot at a Glance. 2002. Dostupné také z: http://paulbourke.net/fractals/mandelbrot/.

[8] *Camera*. Dostupné také z: https://docs.unity3d.com/ScriptReference/Camera.html.

[9] *Camera Inspector window reference for the Built-In Render Pipeline* [online]. [cit. 2025-02-05]. Dostupné z: https://docs.unity3d.com/Manual/class-Camera.html.

[10] *Create and configure a trigger collider*. Dostupné také z: https://docs.unity3d.com/6000.0/Documentation/Manual/collider-interactions-create-trigger.html.

[11] EBERT, David S. *Texturing & modeling: a procedural approach*. 3rd ed. San Francisco: Morgan Kaufmann, c2003. ISBN 1-55860-848-6.

[12] *Edge Collider 2D component reference*. Dostupné také z: https://docs.unity3d.com/6000.0/Documentation/Manual/2d-physics/collider/edge-collider-2d-reference.html.

[13] *Full Screen Pass Renderer Feature reference for URP*. Dostupné také z: https://docs.unity3d.com/Packages/com.unity.render-pipelines.universal@14.0/manual/renderer-features/renderer-feature-full-screen-pass.html.

[14] *GameMaker* [online]. c2013-2024. [cit. 2025-02-19]. Dostupné z: https://gamemaker.io/en.

[15] *GIMP* [online]. 2025. [cit. 2025-02-17]. Dostupné z: https://www.gimp.org/.

[16] *Godot* [online]. 2007-2025. [cit. 2025-02-19]. Dostupné z: https://godotengine.org/.

[17] GREGORY, Jason. *Game engine architecture*. 2nd ed. Boca Raton: CRC Press, c2014. ISBN 978-1-4665-6001-7.

[18] GUMIN, Maxim. *WaveFunctionCollapse* [online]. 21.7.2022. [cit. 2025-03-24]. Dostupné z: https://github.com/mxgmn/WaveFunctionCollapse.

[19] *High-level shader language (HLSL)* [online]. [cit. 2025-02-19]. Dostupné z: https://learn.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl.

[20] *Houdini* [online]. 2025. [cit. 2025-02-21]. Dostupné z: https://www.sidefx.com/products/houdini/.

[21] *How a modular design approach crafted Sable*. Dostupné také z: https://unity.com/resources/shedworks-sable-modular-design-approach.

[22] *ChatGPT*. 2025. Dostupné také z: https://chatgpt.com/.

[23] *Introduction to the Universal Render Pipeline* [online]. [cit. 2025-04-21]. Dostupné z: https://docs.unity3d.com/6000.2/Documentation/Manual/urp/urp-introduction.html.

[24] *Koch Snowflake*. Dostupné také z: https://mathworld.wolfram.com/KochSnowflake.html.

[25] *Krita* [online]. 2025. [cit. 2025-02-17]. Dostupné z: https://krita.org/cs/.

[26] *Leonardo AI*. 2025. Dostupné také z: https://leonardo.ai/.

[27] LOSH, Steve. *Terrain Generation with Midpoint Displacement* [online]. [cit. 2025-03-24]. Dostupné z: https://stevelosh.com/blog/2016/02/midpoint-displacement/.

[28] MAJEWSKI, Jarek. *2D game art, animation, and lighting for artists*. 1. vyd. 2022. Dostupné také z: https://unity.com/resources/2d-game-art-animation-lighting-for-artists-ebook.

[29] *Mastering The Pixel Realm: A Comprehensive Guide To 2D Game Development* [online]. [cit. 2025-02-05]. Dostupné z: https://appsnado.com/blog/mastering-the-pixel-realm-a-comprehensive-guide-to-2d-game-development/.

[30] *Material Maker*. c2020-2021. Dostupné také z: https://materialmaker.org/.

[31] *MonoBehaviour.OnTriggerEnter2D(Collider2D)*. Dostupné také z: https://docs.unity3d.com/6000.1/Documentation/ScriptReference/MonoBehaviour.OnTriggerEnter2D.html.

[32] NYSTROM, Bob. *Rooms and Mazes: A Procedural Dungeon Generator* [online]. [cit. 2025-03-24]. Dostupné z: https://journal.stuffwithstuff.com/2014/12/21/rooms-and-mazes/.

[33] *P5js* [online]. c2025. [cit. 2025-03-18]. Dostupné z: https://p5js.org/.

[34] *ParticleSystem*. Dostupné také z: https://docs.unity3d.com/ScriptReference/ParticleSystem.html.

[35] *Perlin Noise* [online]. [cit. 2025-03-18]. Dostupné z: https://www.arendpeter.com/Perlin_Noise.html.

[36] *Processing* [online]. c2025. [cit. 2025-03-18]. Dostupné z: https://processing.org/.

[37] PRUSINKIEWICZ, Przemyslaw a Aristid LINDENMAYER. *The Algorithmic Beauty of Plants*. 1. vyd. New York: Springer, 1996. ISBN 978-0-387-94676-4.

[38] *Scipy.spatial.Voronoi* [online]. [cit. 2025-04-28]. Dostupné z: https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.Voronoi.html.

[39] ŞEN, Zekai. *Spatial modeling principles in earth sciences*. Dordrecht [u.a.]: Springer, 2009. ISBN 978-1-4020-9671-6.

[40] *Shader block in ShaderLab reference* [online]. [cit. 2025-04-21]. Dostupné z: https://docs.unity3d.com/Manual/SL-Shader.html.

[41] *Shader Graph* [online]. [cit. 2025-02-17]. Dostupné z: https://docs.unity3d.com/Manual/com.unity.shadergraph.html.

[42] *Shader program fundamentals* [online]. [cit. 2025-04-21]. Dostupné z: https://docs.unity3d.com/Manual/writing-shader-programs-introduction.html.

[43] SHAKER, Noor, Julian TOGELIUS a Mark J. NELSON. *Procedural Content Generation in Games* [online]. Cham: Springer International Publishing, 2016 [cit. 2025-02-19]. ISBN 978-3-319-42714-0. Dostupné z DOI: 10.1007/978-3-319-42716-4.

[44] SCHELL, Jesse. *The art of game design: a book of lenses*. 1. vyd. Boca Raton : CRC Press, 2008. ISBN 978-0-12-369496-6.

[45] *Skeletal Animation: A Comprehensive Guide* [online]. [cit. 2025-02-19]. Dostupné z: https://garagefarm.net/blog/skeletal-animation-a-comprehensive-guide.

[46] SMUTNÝ, Jan. *Tvorba 2D platformer hry ve zvoleném herním enginu*. Liberec, 2025. Dostupné také z: https://dspace.tul.cz/. Bakalářská práce. Technická univerzita v Liberci.

[47] *Spine* [online]. c2013-2025. [cit. 2025-02-19]. Dostupné z: https://esotericsoftware.com/.

[48] *Sprite Editor*. Dostupné také z: https://docs.unity3d.com/Manual/sprite/sprite-editor/sprite-editor-landing.html.

[49] *SpriteRenderer* [online]. [cit. 2025-05-01]. Dostupné z: https://docs.unity3d.com/ScriptReference/SpriteRenderer.html.

[50] *SpriteRenderer.flipX*. Dostupné také z: https://docs.unity3d.com/6000.0/Documentation/ScriptReference/SpriteRenderer-flipX.html.

[51] *Style in Profile: Beat-'Em-Up*. Dostupné také z: https://retronaissance.wordpress.com/2022/06/06/style-in-profile-beat-em-up/.

[52] *Substance 3D Designer* [online]. c2025. [cit. 2025-02-21]. Dostupné z: https://www.adobe.com/sk/products/substance3d/apps/designer.html.

[53] *Terraria* [online]. c2021. [cit. 2025-02-21]. Dostupné z: https://terraria.org/.

[54] *Tilemaps* [online]. [cit. 2025-03-24]. Dostupné z: https://docs.unity3d.com/Manual/tilemaps/tilemaps.html.

[55] *Time.time*. Dostupné také z: https://docs.unity3d.com/ScriptReference/Time-time.html.

[56] *Turtle — Turtle graphics* [online]. [cit. 2025-04-28]. Dostupné z: https://docs.python.org/3/library/turtle.html.

[57] *Unity* [online]. 2024. [cit. 2025-02-19]. Dostupné z: https://unity.com/.

[58] *Unlit Shader*. Dostupné také z: https://docs.unity3d.com/Packages/com.unity.render-pipelines.lightweight@5.10/manual/unlit-shader.html.

[59] *Vector2.MoveTowards*. Dostupné také z: https://docs.unity3d.com/6000.0/Documentation/ScriptReference/Vector2.MoveTowards.html.

[60] *Volumes*. Dostupné také z: https://docs.unity3d.com/Packages/com.unity.render-pipelines.universal@7.1/manual/Volumes.html.

[61] WILLIAMS, Richard. *The animator's survival kit*. Expanded edition. London: Faber a Faber, 2009. ISBN 978-0-571-23834-7.

A Generování Voroného diagramu v Pythonu

Následující skript slouží k vygenerování Voroného diagramu na základě náhodně rozmístěných bodů. Výsledný obrázek byl dále použit jako textura ve vizuální části práce.

```
import numpy as np
from scipy.spatial import Voronoi
from PIL import Image

multiplier = 9
width, height = 189 * multiplier, 75 * multiplier
num_points = 100

points = np.random.rand(num_points, 2) * [width, height]
vor = Voronoi(points)

image = np.zeros((height, width), dtype=np.uint8)

for y in range(height):
    for x in range(width):
        distances = np.linalg.norm(points - [x, y], axis=1)
        distance = np.min(distances)
        intensity = np.clip(distance * 2, 0, 255)
        image[y, x] = 255 - int(intensity)

img = Image.fromarray(image)
img.save("voronoi.png")
```

B Generování sněhové vločky pomocí želví grafiky

Tento skript implementuje rekurzivní vykreslování Kochovy vločky pomocí knihovny `turtle` v jazyce Python. Výsledný obrázek je exportován ve formátu `.eps` pro další vektorové zpracování.

```
import turtle
def draw_koch_segment(length, depth):
    if depth == 0:
        turtle.forward(length)
    else:
        length /= 3.0
        draw_koch_segment(length, depth - 1)
        turtle.left(60)
        draw_koch_segment(length, depth - 1)
        turtle.right(120)
        draw_koch_segment(length, depth - 1)
        turtle.left(60)
        draw_koch_segment(length, depth - 1)
def draw_snowflake(length, depth):
    for _ in range(3):
        draw_koch_segment(length, depth)
        turtle.right(120)
turtle.speed(0)
turtle.bgcolor("lightblue")
turtle.color("white")
turtle.penup()
turtle.goto(-150, 90)
turtle.pendown()

draw_snowflake(300, 0)
turtle.hideturtle()

canvas = turtle.getcanvas()
canvas.postscript(file="snowflake.eps")

print("Saved as snowflake.eps")
turtle.done()
```

C Generování Mandelbrotovy množiny v Pythonu

Tento skript slouží k vizualizaci Mandelbrotovy množiny pomocí iterativního výpočtu v komplexní rovině. Výsledkem je černobílý rastrový obrázek, ve kterém intenzita pixelů závisí na rychlosti divergence posloupnosti pro daný bod $c \in \mathbb{C}$.

```
import numpy as np
from PIL import Image

width, height = 800, 600
max_iter = 250

re_start, re_end = -2.0, 1.0
im_start, im_end = -1.0, 1.0

image = np.zeros((height, width), dtype=np.uint8)

for y in range(height):
    for x in range(width):
        c = complex(re_start + (x / width) * (re_end - re_start)
            ,
                    im_start + (y / height) * (im_end - im_start
                       ))
        z = 0
        n = 0
        while abs(z) <= 2 and n < max_iter:
            z = z * z + c
            n += 1
        color = 255 - int(255 * n / max_iter)
        image[y, x] = color

img = Image.fromarray(image)
img.save("mandelbrot.png")
```

D Generování horizontálních čar pomocí knihovny Turtle

Tento skript využívá knihovnu `turtle` k vytvoření pravidelně rozmístěných horizontálních čar přes celou šířku plátna. Výsledkem je soubor `horizontal_lines.eps`, který lze použít jako texturu nebo podklad pro další grafické zpracování.

```
import turtle

screen = turtle.Screen()
screen.setup(width=1920, height=1080)
screen.bgcolor("white")

t = turtle.Turtle()
t.hideturtle()
t.speed(0)
t.color("black")
t.pensize(4)

y = -540
while y <= 540:
    t.penup()
    t.goto(-960, y)
    t.pendown()
    t.forward(1920)
    y += 8

canvas = turtle.getcanvas()
canvas.postscript(file="horizontal_lines.eps", colormode='color'
    )
turtle.done()
```