

Střední průmyslová škola strojní   
a elektrotechnická a Vyšší odborná škola, Liberec 1, Masarykova 3

Paralelizace 3D renderování

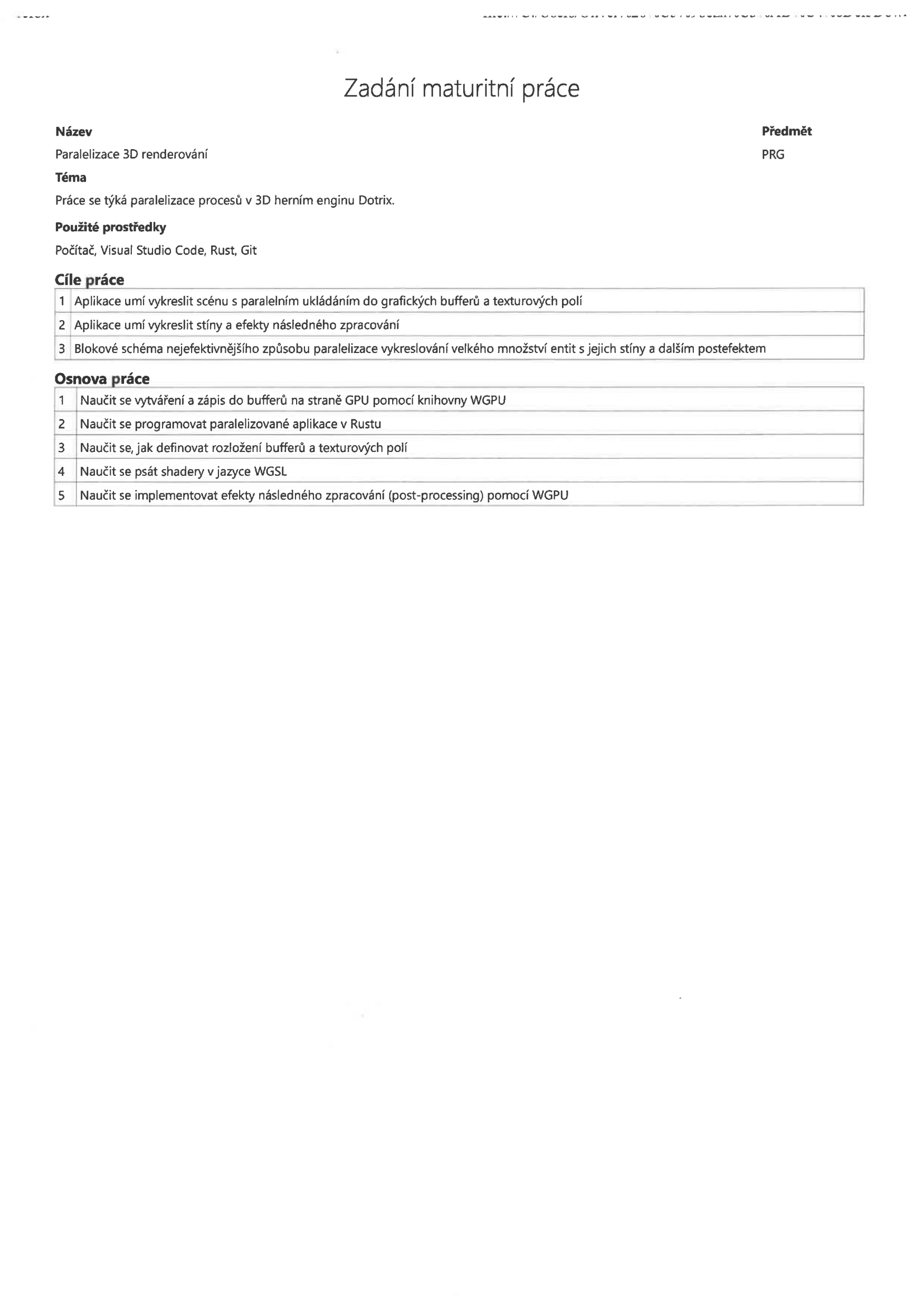
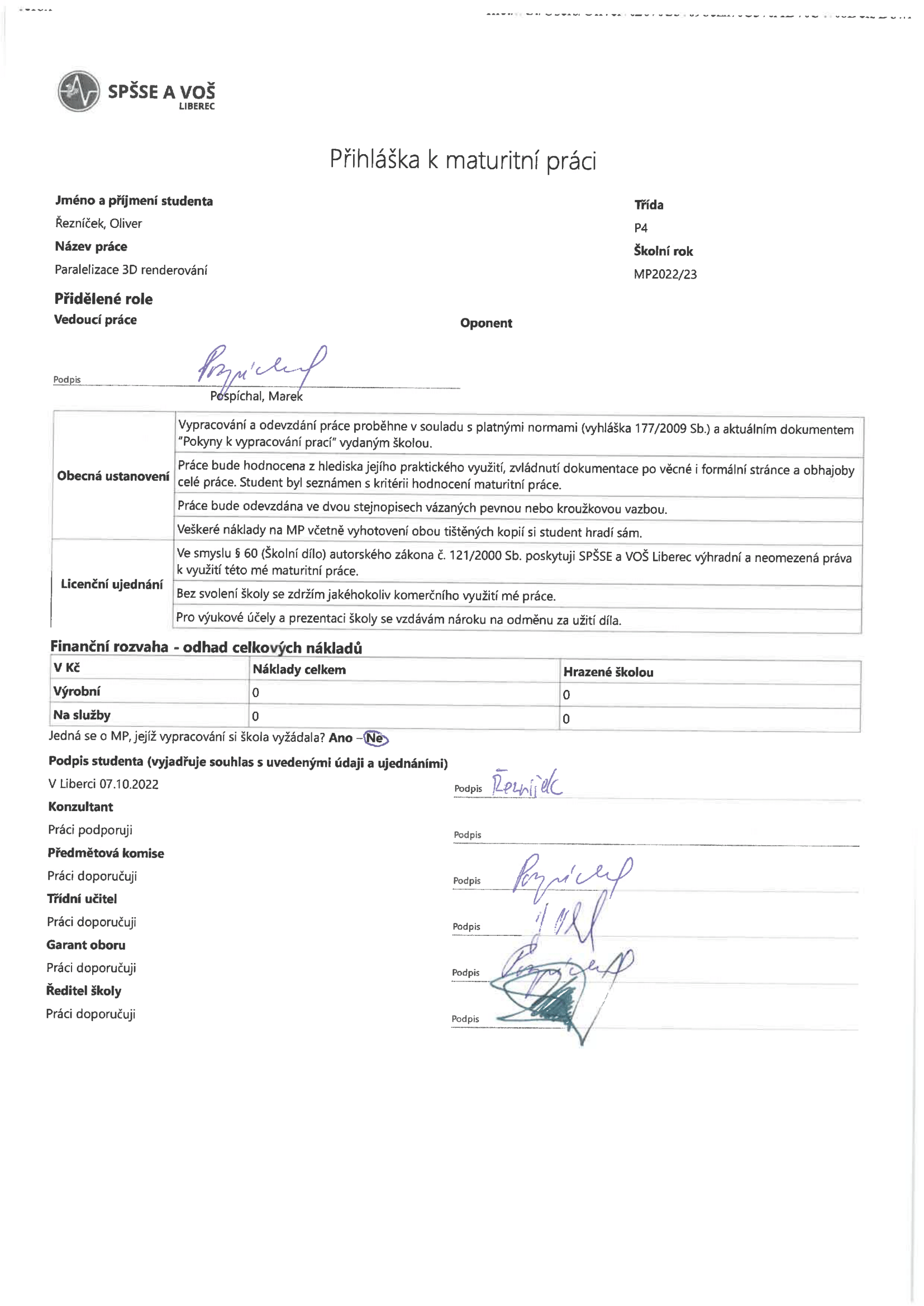
Maturitní práce

Autor **Oliver Řezníček**

Obor **Informační technologie**

Vedoucí práce **Ing. Marek Pospíchal**

Školní rok **2022/2023**



Anotace

Práce se zabývá paralelizací procesů, vykreslením stínů a efekty následného zpracování v open-source 3D herním enginu Dotrix. Bude primárně sloužit jako výzkum a reference před implementací do samotného enginu.

Summary

This work deals with process parallelization, shadow rendering and post-processing effects in the open-source 3D game engine Dotrix. It will primarily serve as research and reference prior to implementation into the engine itself.

Čestné prohlášení

Prohlašuji, že jsem předkládanou maturitní práci vypracoval sám a uvedl jsem veškerou použitou literaturu a bibliografické citace.

V Liberci dne

Oliver Řezníček

Obsah

[Úvod 1](#_Toc129721703)

[1 Práce ve firmě 2](#_Toc129721704)

[1.1 Začátky 2](#_Toc129721705)

[1.2 Zadání 2](#_Toc129721706)

[1.3 Výstup 3](#_Toc129721707)

[2 Použité technologie 4](#_Toc129721708)

[2.1 Rust 4](#_Toc129721709)

[2.1.1 Herní vývoj v Rustu 4](#_Toc129721710)

[2.2 WGPU 4](#_Toc129721711)

[2.2.1 WebGPU 5](#_Toc129721712)

[2.2.2 WGSL 5](#_Toc129721713)

[2.3 Visual Studio Code 6](#_Toc129721714)

[3 Teoretická část 7](#_Toc129721715)

[3.1 Paralelizace 7](#_Toc129721716)

[3.2 Herní engine 7](#_Toc129721717)

[3.2.1 Rendering engine 7](#_Toc129721718)

[3.3 Rendering 8](#_Toc129721719)

[3.4 GPU Buffers 8](#_Toc129721720)

[3.4.1 Vertex buffer 8](#_Toc129721721)

[3.4.2 Index buffer 8](#_Toc129721722)

[3.4.3 Uniform buffer 10](#_Toc129721723)

[3.4.4 Storage buffer 10](#_Toc129721724)

[3.4.5 Indirect buffer 11](#_Toc129721725)

[3.5 Rendering pipeline 11](#_Toc129721726)

[3.5.1 Vertex stage 12](#_Toc129721727)

[3.5.2 Geometry stage 12](#_Toc129721728)

[3.5.3 Rasterization stage 12](#_Toc129721729)

[3.5.4 Fragment stage 12](#_Toc129721730)

[3.6 Compute pipeline 13](#_Toc129721731)

[3.7 Shader 13](#_Toc129721732)

[3.7.1 Compute shader 13](#_Toc129721733)

[3.8 GPU Queue 13](#_Toc129721734)

[3.8.1 GPU commands 14](#_Toc129721735)

[3.9 Textures 14](#_Toc129721736)

[3.9.1 UV coordinates 15](#_Toc129721737)

[3.9.2 Sampling 15](#_Toc129721738)

[3.10 Culling 15](#_Toc129721739)

[3.10.1 Face culling 16](#_Toc129721740)

[4 Praktická část 17](#_Toc129721741)

[*4.1* *Framework.rs* 17](#_Toc129721742)

[4.2 Triangle 18](#_Toc129721743)

[4.3 Cube 19](#_Toc129721744)

[4.4 GPU Driven Rendering 20](#_Toc129721745)

[4.4.1 Zadání 20](#_Toc129721746)

[4.4.2 Princip 20](#_Toc129721747)

[4.4.3 Návrh 21](#_Toc129721748)

[4.4.4 Implementace 22](#_Toc129721749)

[Závěr 3](#_Toc129721750)

[Seznam zkratek a odborných výrazů 4](#_Toc129721751)

[Seznam obrázků 5](#_Toc129721752)

[Použité zdroje 6](#_Toc129721753)

[A. Seznam přiložených souborů I](#_Toc129721754)

Úvod

Téma renderování a matematiky, která stojí za vykreslením počítačové grafiky, mě vždycky zajímalo, ale nikdy jsem se mu nevěnoval tak do hloubky, jak bych chtěl.   
O letních prázdninách jsem poměrně náhodou narazil na firmu Löwenware, kde jsem s odbornou pomocí vyhotovil demo *GPU driven renderování* (vykreslování řízené grafickou kartou) a toto téma mě zaujalo natolik, že jsem se rozhodl ho zpracovat jako maturitní práci.

Tato práce tedy bude moci posloužit všem těm, kteří by chtěli více do hloubky pochopit, jak funguje renderování 2D nebo 3D grafiky. A mohu vám dopředu říct, že se znalostí WGPU budete mít velice dobré chápání toho, jak fungují dnešní moderní grafické API.

# Práce ve firmě

## Začátky

Jak jste se mohli dozvědět v úvodu, tak mě téma renderování už nějakou dobu zajímá, ale jelikož jsem se mu nikdy moc nevěnoval, byly pro mě začátky ve firmě docela složité.

Po asi tak dvou týdnech snahy pochopit Rust, WGPU (nebo celkově 3D grafické API), jak se reprezentují modely, jaké existují druhy bufferů, …, se mi podařilo naprogramovat první demo. Pořád jsem ale z celistvého kódu chápal jen mnou napsanou část a pro vyhotovení něčeho, jako jsou efekty následného zpracování, jsem se ještě musel doučit potřebné základy.

Začal jsem se proto vše učit od samého začátku a přišlo mi zajímavé to zahrnout do mé práce, aby se někdo jiný nemusel s podobnými problémy a otázkami tolik zabývat.

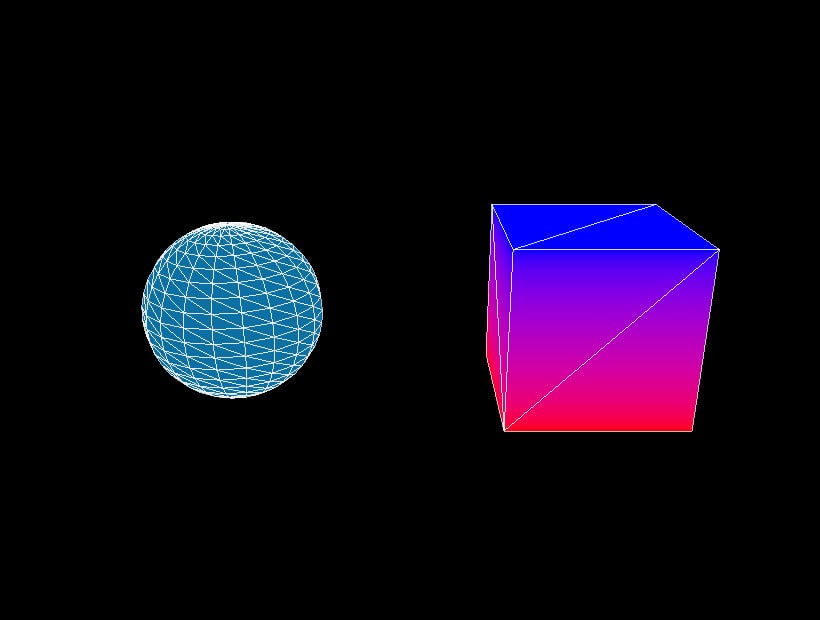
## Zadání

Moje práce ve firmě spočívá v takovém průzkumu. Obvykle dostanu pár odkazů na nějaký koncept (návrh, myšlenku, feature, …), který je potřeba implementovat do enginu. Plus se většinou ještě s nadřízeným osobně domluvím na detailech a v průběhu práce s ním veškeré svoje dotazy konzultuji.

Příkladem může být zadání na *GPU driven renderování*, kde bylo cílem, co nejefektivněji vykreslit velké množství objektů:

* <https://vkguide.dev/docs/gpudriven/gpu_driven_engines/>

## Výstup

Výstup práce může potom vypadat nějak takto:

Obrázek - GPU Driven Rendering

Výsledkem je spustitelná aplikace napsaná v jazyce Rust, která pomocí knihovny WGPU vykreslí nějakou scénu. Obvykle implementuje určitý koncept ve své jednoduché formě. To znamená, že se nezabývá problematikou samotného enginu, ale jen se snaží, co nejjednodušší cestou ukázat způsob implementace daného problému.

Jinými slovy, nepotřebujeme dělat věci jako přesun generování modelů z CPU na GPU, protože to zkrátka ani není cílem. Cílem je většinou implementovat jednu techniku renderování, a ne se jich tam snažit dostat co nejvíc.

# Použité technologie

## Rust

Rust je open-source, multiparadigmatický, víceúčelový programovací jazyk. Oproti jiným, jemu podobným programovacím jazykům jako je např. C nebo C++ nabízí bezpečnost paměti, která umožňuje vytvářet spolehlivé a bezpečné aplikace.

Unikátní na Rustu je hlavně to, jakým způsobem řeší bezpečný přístup do paměti. Má totiž tzv. systém vlastnictví a půjčování (ownership and borrowing), díky kterému dokáže už během kompilace řešit chyby, na které byste narazili v jazyce C až při běhu programu. Takže se vyhnete častým chybám paměti, a to i bez použití garbage collectoru.[[1]](#footnote-1)

### Herní vývoj v Rustu

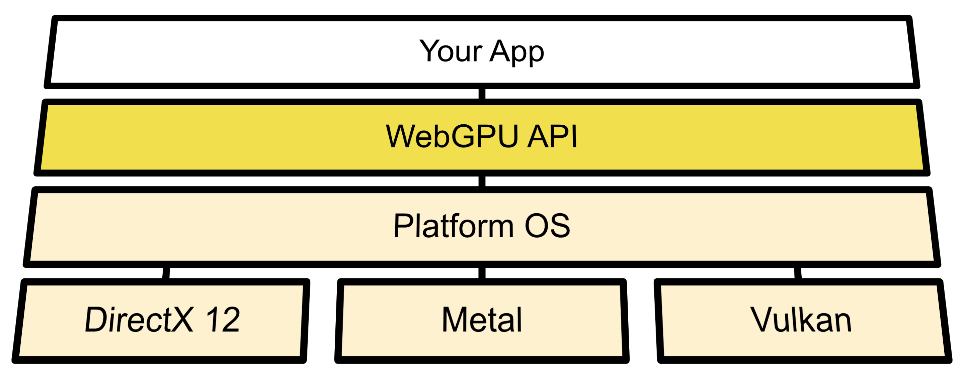
Rust je relativně nový jazyk, který byl poprvé vydán v roce 2010. Popularitou začal stoupat zejména v posledních letech. V důsledku toho je v Rustu méně zkušených vývojářů ve srovnání s jinými jazyky, jako je C++ nebo Java, které jsou častěji používány pro vývoj her.

Jedním z hlavních důvodů, proč si zvolit Rust pro vývoj her je bezpečnost   
a spolehlivost, kterou poskytuje. Dalším důvodem je vysoká úroveň optimalizace, která umožňuje vývojářům psát hry, které jsou rychlé a efektivní. To je zejména důležité pro hry, které potřebují vysoké frameraty a nízkou latenci (např. FPS).

Vývoj her v Rustu je stále v počátcích, ale stále se jedná o zajímavou možnost pro vývojáře, kteří hledají nové a moderní přístupy k tvorbě her. S rostoucím počtem vývojářů a herních nástrojů, které jsou v Rustu k dispozici, můžeme očekávat, že Rust bude hrát stále významnější roli v oblasti herního průmyslu.

## WGPU

WGPU je grafické API napsané v Rustu, které umožňuje kompilaci pro více platforem díky tomu, že běží na nativních backendech jako je Vulkan, Metal, DirectX a OpenGL.

To znamená, že můžeme naši grafickou aplikaci přepnout na jiný backend a tak sestavení směřovat na úplně jinou platformu. Jinými slovy řečeno, naše aplikace bude spustitelná na všech platformách. Někdo by mohl argumentovat, že aplikace napsaná ve Vulkanu nebo OpenGL bude taky multiplatformní a asi by měl pravdu. Výhodou WGPU ovšem je, že aplikace může běžet na takovém backendu, který je pro nás nebo danou platformu nejlepší.

Obrázek - diagram architektury WebGPU

Vývojáři by samozřejmě mohli integrovat víc grafických API do jejich herního enginu, ale to zabere spoustu času a vše se musí dopředu promyslet. Použití WGPU by jim v takovém případě práci dost usnadnilo, protože vytváří abstrakci nad všemi známými backendy.

### WebGPU

Pokud vás už podle nadpisu napadlo, že by mohla existovat určitá souvislost mezi WGPU a WebGPU, nejste na omylu. WGPU je totiž pouze Rust implementací WebGPU.

WebGPU je nadcházející moderní grafické API, které rozšíří možnosti počítačové grafiky na webu např. o compute shadery. Je vyvíjeno W3C (organizace pro mezinárodní standardy na webu) s profesionály z Applu, Mozilly, Microsoftu, Googlu a dalších. Údajně původní návrh vzniknul podle grafického API Metal od Applu.

V době, kdy tuto práci píšu, ještě WebGPU není plně integrované do webových prohlížečů. Zatím se objevilo pouze v testovacích verzích prohlížečů jako je Firefox Nightly nebo Chrome Canary.

### WGSL

WGSL (WebGPU Shading Language) je jazyk na psaní shaderů určený pro WebGPU. Svou syntaxí se velmi podobá Rustu, ale nenechte se zmást, jedná se o úplně jiný jazyk.

## Visual Studio Code

Visual Studio Code je v současnosti asi nejznámější textový editor pro vývojáře. Vyznačuje se elegantním designem, integrovaným terminálem a mimo jiné poskytuje také nástroje pro debugging a práci s githubem. Kdyby vám v editoru přece jen něco chybělo nebo byste chtěli jiné barevné schéma, můžete se podívat do záložky rozšíření (extensions).

Kromě uvedených vlastností se mi líbí, jak intuitivně s ním dokáže člověk pracovat a snadno se v něm zorientovat. Pro moje účely vývoje byl tak jasnou volbou.

# Teoretická část

## Paralelizace

Paralelizace je proces, při kterém je náš program nebo výpočet rozdělen na více úloh, které běží na více procesorech. Díky rozdělení zátěže mezi více procesorů můžeme ušetřit energii a znásobit výpočetní výkon (Výkon jednoho procesoru × počet procesorů[[2]](#footnote-2)).

V případě této práce se bude jednat o přesun grafických výpočtů ze strany procesoru na grafickou kartu. Grafické výpočty jsou opakované často za sebou a dají se snadno rozložit mezi více jader (např. násobení matic). Grafická karta je tedy ideální výpočetní jednotkou pro provádění těchto operací. Má v sobě spoustu jader, které ani zdaleka nedosahují výkonu hlavního procesoru, ale jejich síla spočívá právě v počtu.

Paralelizaci chceme tedy používat hlavně v případě výpočtů, které se dají rozložit na menší, jednodušší a využijeme u nich spíš velké množství procesorů než procesor jeden s velkou frekvencí.

U paralelních výpočtů je důležitá ještě jedna věc, kterou jsem nezdůraznil, a to je souběh. Někdy potřebujeme, aby jednotlivé dílčí úlohy byly vykonávány synchronně. Předávání informací mezi úlohami může být v takovém případě dost problematické. (1)

## Herní engine

Herní engine je vývojové prostředí speciálně navržené, aby pomáhalo vývojářům vytvářet hry efektivněji. Poskytuje sadu nástrojů, které umožňují tvorbu počítačových her bez nutnosti stavět vše od začátku. Typicky herní engine poskytuje nástroje pro vykreslování 2D a 3D grafiky, simulaci fyziky, skriptovací jazyk a různé další funkce.

Zjednodušeně se tak vývojáři mohou více soustředit na vytváření herních mechanismů a ostatní věci nechat na enginu.

### Rendering engine

Rendering (vykreslovací) engine je zodpovědný za vykreslení jednotlivých objektů, stínů a dalších věcí v naší scéně. Všechno, co v počítačové hře vidíte na obrazovce je tedy (mimo spousty práce, kterou museli hře věnovat herní vývojáři a designéři) zásluhou vykreslovacího enginu[[3]](#footnote-3).

## Rendering

Rendering neboli vykreslování je proces, kdy měníme naše modely objektů (jejich matematickou reprezentaci) na viditelný 2D obraz, který si můžeme zobrazit na monitoru nebo jiném zobrazovacím zařízení.

## GPU Buffers

GPU Buffer reprezentuje blok paměti, který může být použit v GPU operacích. Data jsou uložena v lineárním uspořádání, což znamená, že každý bajt může být adresován pomocí offsetu od začátku Bufferu.

### Vertex buffer

Vertex buffer je datová struktura v počítačové grafice, která uchovává vertex data jednoho nebo několika modelů v 3D prostoru. Používá se ve vertex shader stage.

Pojmem vertex data se myslí nějaké pole vertexů, kde každý vertex většinou uchovává data jako:

1. Pozice – 2D nebo 3D (X, Y, Z)
2. Barva– typicky RGB
3. Normálový vektor – definuje, jak je zakřivený povrch v místě vrcholu

…

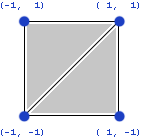
### Index buffer

Index buffer, také známý jako index array, je datová struktura, která se v počítačové grafice používá na uchování index dat nějakého modelu.

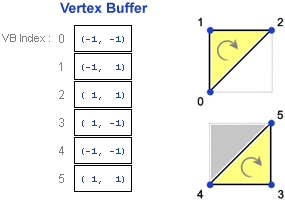
Index buffer obsahuje pole čísel, které odpovídají indexům jednotlivých vrcholu ve vertex bufferu. Tyto indexy se poté používají ke konstrukci trojúhelníků, které tvoří povrch 3D modelu.

S použitím index bufferu může GPU efektivně vykreslit stejný vertex několikrát bez duplikování dat ve vertex bufferu. Index buffer tak hraje důležitou roli ve snížení celkové paměti potřebné k uložení 3D modelu. U malých modelů není tento rozdíl v zabrané paměti příliš znatelný, ale čím víc vertexů náš model (mesh) má, tím výhodnější pro nás použití index bufferu bude.

#### Použití index bufferu

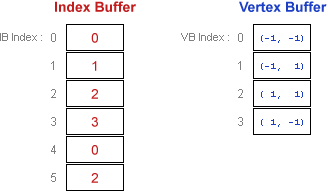
Pro lepší pochopení si ukážeme příklad vykreslení modelu bez a s index bufferem. Na začátku budeme mít nějaký čtverec se středem v bodě [0,0], ten vykreslíme jako dva trojúhelníky, náš primitive type tak bude *TriangleList*. (2)

Obrázek – čtverec

V našem příkladu bude jeden vertex ve vertex bufferu reprezentován jen jako pozice v 2D prostoru. Výsledný vertex buffer by mohl vypadat takto:

Obrázek - použití vertex bufferu

Pokud do našeho řešení vykreslení čtverce zakomponujeme index buffer, budou naše buffery vypadat následovně:

Když budeme předpokládat, že všechna čísla zabírají v paměti 4 byty, tak v prvním případě nám bude čtverec zabírat **6×8 = 48 bytů** a v druhém případě **6×4 + 4×8 = 56 bytů**. Jak můžete sami vidět, u vykreslení dvou trojúhelníků se použití index bufferu ještě zdaleka nevyplatí, ale s jejich přibývajícím počtem se tato situace v určitém bodě zlomí a index buffer nám už bude paměť pouze šetřit.

Obrázek - použití index bufferu

### Uniform buffer

Uniform je něco jako globální proměnná v našem shader programu. Uniform buffer se nazývá uniform, protože se nemění v rámci jednotlivých draw callů, ale jeho hodnota je stejná (jednotná) pro všechna volání.

Do uniform bufferu si můžeme uložit data potřebná k vykreslení, která se nebudou často měnit jako je např. projection nebo view matice.

### Storage buffer

Uniform buffery jsou vhodné pro malá data určená pouze ke čtení. Když ale budeme chtít mít v shaderu data o neznámé velikosti nebo data, která chceme měnit, na to se používají storage buffery. Vytvářejí se stejným způsobem jako uniform buffery. Vlastně i fungují hodně podobně, mají jen jiné vlastnosti jako větší maximální velikost a dá se do nich zapisovat ze shaderu.

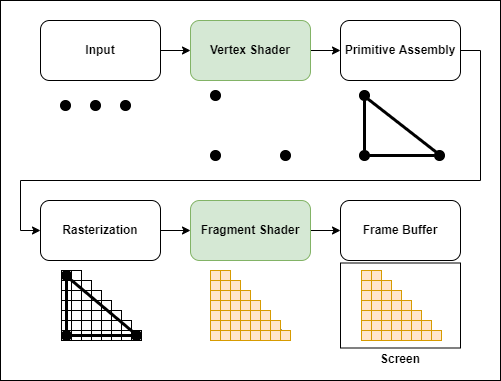
Storage buffery jsou obvykle o něco pomalejší než uniform buffery, ale můžou být mnohem větší. Pokud např. chceme nahrát celou scénu do jednoho bufferu, budeme je muset použít. (3)

### Indirect buffer

Indirect buffer se používá v indirect draw callu, kde nahrazuje funkci parametru. Má jasně definovanou strukturu, která se skládá z indirect draw commandů. Ty v sobě obvykle obsahují informace jako, odkud máme začít kreslit z index/vertex bufferu, počet vertexů k vykreslení nebo kolik instancí daného modelu chceme vykreslit.

Díky použití indirect bufferu můžeme snížit počet draw callů potřebných k vykreslení naší scény. Bude nám tak užitečný v situacích, kdy potřebujeme vykreslit velké množství objektů (příkladem můžou být particly v particle systému).

## Rendering pipeline

Rendering pipeline je sekvence kroků, kterými musí naše objekty ve scéně projít, než se vykreslí na obrazovku. Také si jí můžeme představit jako abstraktní třídu nebo model, který popisuje způsob vykreslení naší scény. Typicky se dělí na tyto kroky uvedené na obrázku:

Obrázek - WGPU render pipeline

Zeleně vybarvené části jsou programovatelné. Kód, kterým můžeme ovlivnit chování v těchto krocích, se píše do **shaderu**.

### Vertex stage

Vertex stage je zodpovědná za zpracování dat jednotlivých vertexů. To většinou zahrnuje transformaci souřadnic z world space do view space[[4]](#footnote-4). Typicky zde provádíme maticové operace jako např. násobení.

### Geometry stage

Po vertex stage následuje geometry stage nebo také tesselation stage. Ta je zodpovědná za sestavování primitiv (např. trojúhelníků) z vertexů.

#### Typy primitiv

V naší render pipeline můžeme většinou definovat tyto typy primitiv:

1. Point – každý vrchol představuje bod v prostoru
2. Line – každá dvojice vrcholů tvoří novou čáru
3. Triangle – každá trojice vrcholů tvoří nový trojúhelník

### Rasterization stage

Rasterization stage určuje pixely, které překrývají daná primitiva, abychom mohli vypočítat barvu všech překrytých pixelů. Neboli převádí naše primitiva z vektorové do rastrové grafiky.

### Fragment stage

Ve fragment stage spočítáme pro každý pixel překrytý primitivem očekávanou barvu. Barva se většinou předává rovnou s vertexy nebo ji můžeme namapovat z textury. Také zde můžeme provádět výpočet světla, abychom rozlišili, kolik světla dopadá   
na určité pixely.

Výstupem této stage je už konečný frame buffer.

## Compute pipeline

## Shader

Shader je počítačový program, který slouží k popsání programovatelných částí rendering pipeline. K tvorbě shaderů se používají speciální programovací jazyky tzv. shader jazyky. V našem případě budeme používat jazyk **WGSL**.

Nejčastěji se budete potkávat s těmito typy shaderů:

1. **Vertex shader**
2. **Fragment shader**
3. **Compute shader**

Pro co se používá vertex a fragment shader je popsáno výše u jednotlivých částí rendering pipeline, takže tyto dva vynechám a pouze popíšu, k čemu slouží compute shader.

### Compute shader

Když potřebujeme udělat nějaký dodatečný výpočet s použitím grafické karty, tak přichází na řadu compute shader. Pomocí něj můžeme urychlit obecné algoritmy spuštěním na grafické kartě. Jedním z příkladů použití compute shaderů je generování meshů nebo vytváření particle systémů.

Proč tedy není v render pipeline něco jako compute stage? Compute shader je taková pomocná ruka, která je při renderování občas budeme potřeba, ale se samotným procesem render pipeline nemá nic společného. Je totiž součástí compute pipeline[[5]](#footnote-5).

## GPU Queue

Queue neboli fronta je takový speciální typ bufferu, který slouží pro ukládání příkazů (commandů), než bude grafická karta připravena je zpracovat. Procesor může odesílat příkazy do více front současně, což umožňuje efektivní paralelní zpracování   
na GPU.

### GPU commands

Command je konkrétní objekt, který je zaznamenán do command bufferu   
a později proveden na grafické kartě. Příkazy odesílané do fronty obvykle obsahují instrukce pro vykreslování grafiky nebo instrukce pro univerzální výpočty na GPU.

Podle účelu výpočtu tak dělíme GPU commandy na dva typy:

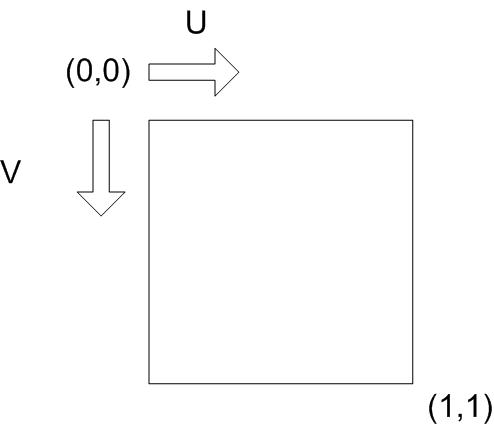
1. **Draw command** – spouští render pipeline
   1. **Draw** – kreslí primitiva
   2. **Draw indexed** – kreslí indexovaná primitiva
   3. **Draw indirect** – kreslí primitiva z dat, které oproti obyčejnému draw callu jsou předávány v GPU Bufferu
   4. **Draw indexed indirect** – kombinuje dva předchozí draw cally dohromady – je to tedy indexovaný indirect draw
2. **Dispatch command** – spouští compute pipeline

## Textures

Textury jsou nedílnou součástí počítačové grafiky, protože umožňují věrně realisticky zobrazovat povrch objektů ve 3D prostředí. Textury také přidávají scéně vizuální složitost, aniž by bylo nutné modelovat každý detail ručně, což by bylo časově náročné a celkem nepraktické.

Textury lze použít k simulaci vzhledu nejrůznějších materiálů, jako je dřevo, kov, látka nebo kámen. Lze je také použít pro světelné efekty, například pro evokování vzhledu stínu nebo odrazu. Celkově hrají textury klíčovou roli při vytváření realistických a pohlcujících vizuálních zážitků v počítačové grafice.

### UV coordinates

Většina textur je pouhé dvourozměrné pole barevných hodnot. Jednotlivým barevným hodnotám se říká texel. Každý texel má v textuře unikátní adresu. Adresu si lze představit jako číslo sloupce a řádku, které se označují jako U a V. Obě tyto proměnné mohou nabývat hodnot pouze v rozsahu od 0 do 1. (4)

Obrázek - UV coordinates

### Sampling

Při použití textury na primitivum v 3D prostoru musí být adresy jejích texelů namapovány na souřadnice objektu. Neboli na adrese texelu se získá barevná hodnota, která poté připadne některému z pixelů uvnitř objektu. Tomuto procesu mapování se také někdy říká sampling.

#### Sampler

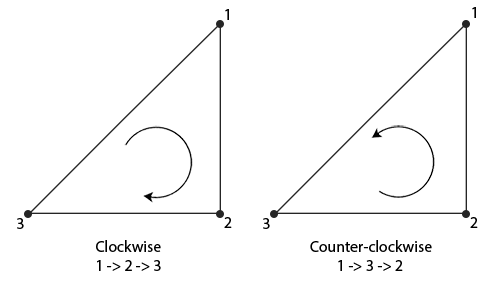
Sampler je objekt, který definuje průběh procesu mapování textur. Může definovat chování při zvětšování nebo zmenšování měřítka textur, co dělat při překročení rozsahu UV souřadnic (0–1) nebo určovat způsob míchání hodnot jednotlivých texelů   
u samplování.

## Culling

Culling obecně jsou v počítačové grafice techniky odstraňování objektů   
nebo jejich částí z procesu render pipeline. Díky tomu je snížen celkový počet objektů, které musí grafická karta vykreslit, a tak zvýšen výkon.

### Face culling

Zkuste si představit krychli, ta má 6 stěn. Pokud jí otočíte z jakéhokoliv úhlu, uvidíte maximálně 3 stěny. Při vykreslování by se nám hodilo brát v potaz pouze viditelné stěny a zbytek vůbec nevykreslovat, což platí i v případě jiných objektů. To je přesně případ použítí face cullingu.

Face culling nám dokáže ušetřit více než 50 % z primitiv[[6]](#footnote-6) potřebných pro vykreslení objektu. Obvykle se dá nastavit přímo v render pipeline.

Obrázek - Winding order

To, jestli vykreslujeme primitivum, které je k nám čelem nebo opačnou stranou se určuje podle pořadí jednotlivých vertexů. Jejich pozice mohou být uspořádány buď po, nebo proti směru hodinových ručiček. S použitím tohoto mechanismu, může grafická karta jednoduše rozlišit, která primitiva je třeba vykreslit a která nikoliv. (5)

# Praktická část

Celý projekt je rozdělen na několik příkladů, které popisují, jak se projekt postupně vyvíjel. Příklady nám pomáhají lépe pochopit jednotlivé problémy a jejich konkrétní implementaci.

## *Framework.rs*

Celý návrh tohoto projektu vychází z githubu WGPU a stěžejní pro nás bude soubor *framework.rs* nacházející se ve složce examples.

Na začátku souboru se nachází *trait[[7]](#footnote-7)* s názvem *Example*, který představuje konkrétní příklad. Ten se pro ulehčení používá vždy při vytváření nového examplu.

Následují tři podprogramy *setup*, *start* a *run[[8]](#footnote-8)*.

1. Podprogram *setup* vytváří nové okno v operačním systému a inicializuje WGPU.
2. Druhý podprogram v pořadí s názvem *start* spouští naší aplikaci a vytváří event loop našeho okna
3. Třetí podprogram už jen dává ty dva první dohromady. Takže spustí asynchronně *setup* a výsledek z něj předá podprogramu *start*.

Soubor *framework.rs* tedy tvoří vlastně gró celé aplikace. Při vytvoření nového příkladu je vždy nadefinována nová strukturu implementující trait *Example* , který se pouze předá jako parametr funkci *run*. Tímto předáním se poté spustí nově nadefinované chování v podprogramech *init*, *resize*, *update* a *render*.

## Triangle

Úplně prvním z příkladů v tomto projektu je jednoduchý trojúhelník. Ten vůbec nebyl součástí mojí práce ve firmě, ale jak jste se mohli dozvědět v jedné z prvních kapitol této práce, tak *GPU Driven Rendering* jsem naprogramoval bez jakéhokoliv hlubšího chápání procesu vykreslování. Proto jsem se rozhodl začít učit všechno od začátku.

Zvolil jsem si trojúhelník, protože je to vlastně nejjednodušší možný příklad vykreslování něčeho na obrazovku. Jedná se o takovou „Hello World!“ aplikaci pro grafiky.

Obrázek - Triangle example

## Cube

Cube je příklad celý přejatý z githubu knihovny WGPU, jen jsem ho osekal o pár nadbytečností. Vycházel jsem z něj při své první práci ve firmě.

Tento příklad, jak vyplývá z názvu, vykresluje krychli, která oproti trojúhelníku již leží v 3D prostoru. Obrazovka počítače je ale pouhá 2D plocha, takže pro vykreslení 3D objektů je potřeba najít způsob, jak je převést do 2D prostoru. Pro tyto účely se používá *projection* a *view* matice.

Projection (promítací) matice se používá pro transformaci 3D souřadnic do 2D prostoru. Existuje několik druhů promítání, ale nejčastěji používanými jsou*perspektivní* a *ortografické[[9]](#footnote-9)*.

Perspektivní promítání transformuje souřadnice tak, aby se bližší objekty zdály větší a ty vzdálenější menší, což vytváří pocit hloubky, který je nezbytný pro vytvoření realistické 3D scény.

Obsah obrázku papírnictví, obálka

Popis byl vytvořen automatickyView matice se používá k transformaci ze světových souřadnic objektu   
na souřadnice kamery, což umožňuje vykreslit objekt z pohledu kamery. Je to tedy normální trasformační matice, která se ale používá pro transformaci všech objektů ve scéně, aby byly vidět z pohledu kamery.

Obrázek - Cube example

## GPU Driven Rendering

### Zadání

Moje první zadání ve firmě bylo vyhotovit demo *GPU driven renderování* podle článku z webové stránky *Vulkan Guide*.

Cíl:

* Pomocí GPU driven renderingu co nejefektivněji vykreslit objekty ve scéně

Vedlejší cíle:

* Použít texture array
* Data objektů ve scéně jsou uloženy ve storage bufferu

### Princip

Pokud si lámete hlavu, proč je v názvu práce slovo paralelizace, je to právě kvůli tomuto příkladu. V podstatě už psaní shaderů je samo o sobě určitou paralelizací, protože se tyto programy spouští paralelně na grafické kartě. Paralelizace GPU driven renderování však spočívá trochu v něčem jiném.

Celý proces využívá výše uvedeného indirect draw callu, který si bere parametry z GPU bufferu místo ze samotného volání. Díky tomuto mechanismu si může grafická karta pozice a informace o následném vykreslení vzít přímo z bufferu, čímž se minimalizuje přesun dat mezi CPU a GPU. Jakým způsobem se zátěž rozloží mezi jednotlivými jádry, si už obstará grafická karta[[10]](#footnote-10).

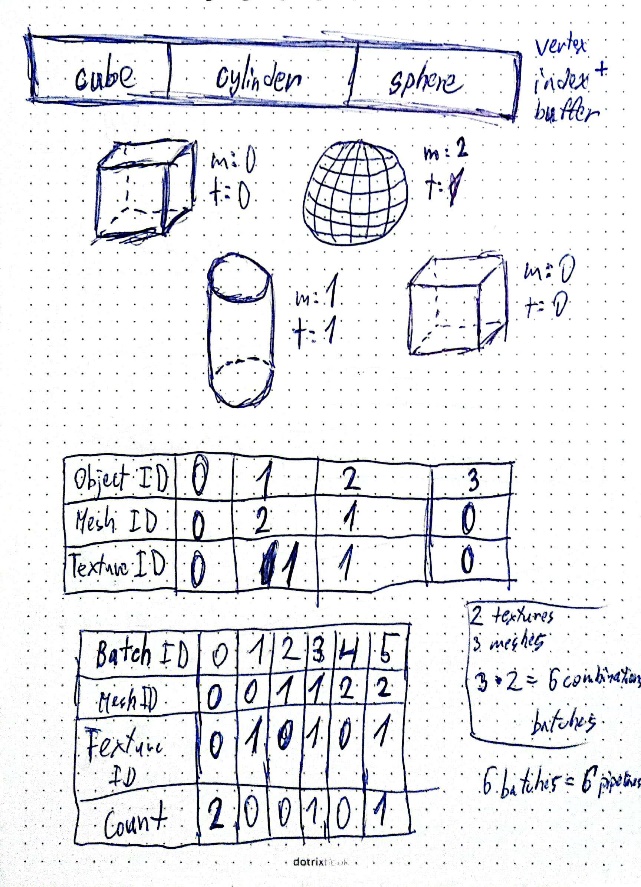
Klíčovým bodem při designu GPU driven renderování je, že veškerá scéna (např. naše objekty) by měla být na straně GPU. Vytvoříme si velké GPU buffery (vertex + index), které se poté pošlou na grafickou kartu. Tím se snažíme vyhnout konstantnímu bindování dat mezi draw cally. Bindless design nám pomůže zlepšit výkon jak na straně procesoru, tak na straně grafické karty, kterou dokážeme s velkými draw cally lépe využít.

Moderní grafické karty milují, když jim zadáte obrovské množství práce při každém draw callu, protože tak mohou dosáhnout 100% využití. Budeme se tedy snažit, vykreslit na jeden draw call co nejvíce objektů. (6)

### Návrh

Základem GPU driven renderování je mimo vytvoření velkých bufferů s naší scénou nebo bindless designu také použití indirect draw callu.

Abychom ho dokázali využít co nejlépe, rozdělíme objekty do tzv. *batchů.* Batch je množina objektů, které mají společný *mesh* a *texture*. Každý batch bude vykreslen pomocí jednoho indirect draw callu, který provede instancované kreslení[[11]](#footnote-11). Jakmile budou připraveny vertex a index buffery, vykreslíme scénu na tolik draw callů, kolik jsme sestavili batchů z našich objektů. V podstatě se snažíme objekty sjednotit do takových skupin, aby byl konečný počet draw callů co nejmenší.

Musím říct, že i mně se to po přečtení článku dost špatně představovalo, takže jsem si nakreslil obrázek:

Obrázek - Návrh GPU Driven Rendering

Na obrázku je vidět příklad rozdělení objektů do batchů. Jsou tam 3 meshe (krychle, válec, koule) a 2 textury označeny čísly. Když si v poslední tabulce odmyslíte sloupečky, kde je *count* nula, tak zbydou 3 batche, které představují indirect draw cally potřebné k vykreslení této scény.

Tento návrh řešení je pěkný, ale poněkud zastaralý. V době, kdy článek vzniknul, se totiž ještě nepočítalo s existencí draw callu s názvem *multi\_draw\_indirect*. Ten dokáže provést námi určený počet indirect volání. My si tak můžeme vytvořit indirect buffer obsahující hned několik struktur *DrawIndexedIndirect* popisujících indirect call.Díky jeho použití následně dokážeme vykreslit celou scénu na pouhé jedno volání[[12]](#footnote-12)!

Takže podle plánu nahrajeme celou scénu i s potřebnými bindingy na GPU, pak rozdělíme objekty do batchů, z nichž vytvoříme indirect buffer, vykreslíme naši scénu a máme hotovo.a

### Implementace

V praktické implementaci budeme mít 3 meshe a 3 textury, ze kterých vytvoříme 4 objekty.

Meshe:

* Krychle
* Válec
* Koule

Textury:

* Modrá



* Červená
* Žlutá

Textury jsem vytvořil v programu *Gimp* po domluvě s nadřízeným. Jsou to PNG obrázky o velikosti 8x8 a všechny pixely mají stejnou barevnou hodnotu. Důvod, proč nebyly zvoleny složitější textury je jednoduchý: *texture coordinates*. Vymýšlení, jak namapovat texturu na mesh, může být někdy komplikovaný úkol a pro ilustraci nám tyto jednoduché textury budou bohatě stačit*[[13]](#footnote-13)*.

Projekt jako takový je rozdělen do dvou částí. První z nich je jako u předchozích příkladů soubor *main.rs* spolu se shader kódem, kde je naprogramovaná hlavní část vykreslovacího mechanismu. Druhou částí je modul s názvem *shapes*, kde jsou všechny věci týkající se manipulace s objekty (obsahuje např. strukturu pro mesh, objekt a batch nebo pomocnou funkci, která spojuje vertex a index data několika meshů dohromady).

Abychom se dostali k výslednému snímku, musí se provést následující kroky:

1. Vytvoření meshů -> z nich poté jeden velký vertex a index buffer
2. Vytvoření objektů
   1. Modrá krychle
   2. Červený válec
   3. Modrá krychle
   4. Žlutá koule
3. Vytvoření batchů z existujících objektů
   1. Batch1 { mesh: krychle, texture: modrá, count: 2 }
   2. Batch2 { mesh: válec, texture: červená, count: 1 }
   3. Batch3 { mesh: koule, texture: žlutá, count: 1 }
4. Vytvoření bind groupy
   1. Transformační matice kamery (projection \* view) [UNIFORM]
   2. Pole transformačních matic objektů [STORAGE]
   3. Pole dalších informací o objektech (transform\_id, texture\_id) [STORAGE]
   4. Pole textur (modrá, červená, žlutá)
   5. Sampler
5. Vytvoření indirect bufferu pomocí batchů
6. Vytvoření render pipeline
7. Zavolání *multi\_draw\_indirect*

Důležité detaily:

U 1. kroku se nastaví všem vertexům u všech meshů *texture coordinates* na (0.5, 0.5) neboli si každý vertex bude brát barevnou hodnotu z prostředku textury. I když v případě našich jednobarevných textur je to vlastně irelevantní.

U 3. kroku se do batchů ukládají i trasformační matice jednotlivých objektů. Scéna se totiž vykresluje podle batchů a jejich vytvořením se úplně zpřehází původní pořadí objektů. Takže by objekty mohly být vykresleny s nežádoucí barvou nebo na nežádoucí pozici.

U kroku 4.b a 4.c by se dalo namítat proč nejsou spojeny dohromady v jeden binding. Měly být spojeny, ale nepovedlo se mi to provést. Oba bindingy představují informace o objektech. Snažil jsem se v shaderu vytvořit array struktur *Object*, které by nesly informace o transformační matici a indexu textury, která se má na objekt použít. Bohužel to nešlo tak, jak jsem si představoval[[14]](#footnote-14), takže jsem byl nucen jeden storage buffer rozdělit na dva:

1. Pole transformační matic
2. Pole (transform\_id, texture\_id)
   1. transform\_id – index transformační matice objektu
   2. texture\_id – index textury objektu

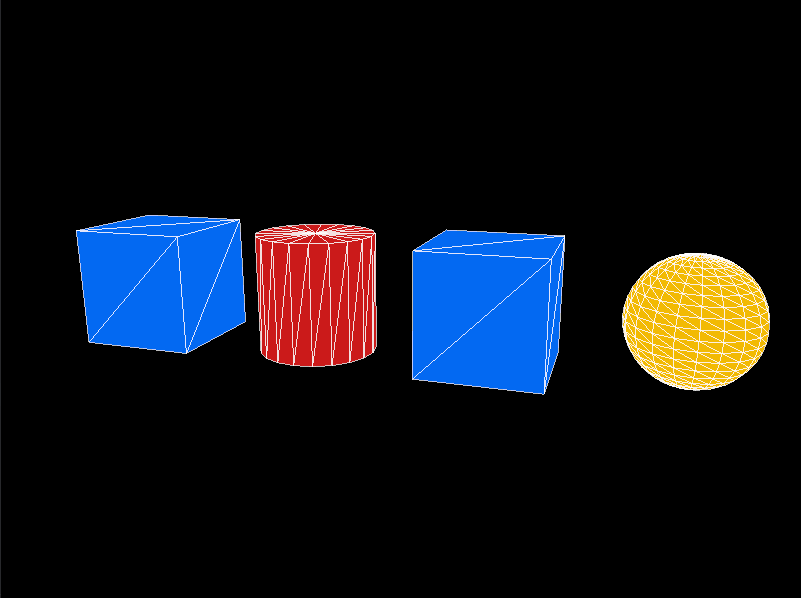
Páty krok je asi nejdůležitějším ze všech. V něm se vytváří indirect buffer.

Na začátku je vytvořeno pole bytů, do kterého se postupně přidávají struktury *DrawIndexedIndirect*.

Struktura v sobě obsahuje hned několik polí. Prvním z nich je *vertex\_count*, který obsahuje velikost index dat meshe, který chceme vykreslit. Takže jsem si vytvořil pomocnou funkci *index\_data\_len*, která pomocí vstupního parametru typu *MeshType* vrátí tuto hodnotu. Trvalo mi opravu velmi dlouho pochopit, že vertex count zde znamená vlastně počet indexů v index bufferu, ale jelikož se jedná o indexované kreslení, tak to dává i docela smysl. Do pole ­*index\_count* se přiřadí počet instancí. *Base\_index* vyjadřuje počáteční index v index bufferu, ze kterého chceme začít kreslit. *Vertex\_offset* je hodnota, která se přičítá k indexům před indexováním do vertex bufferu. Nulu tam přiřazuji proto, že toto přičtení už bylo provedeno při sjednocení vertex a index bufferů dohromady[[15]](#footnote-15). Poslední pole je *base\_instance*, které určuje počáteční číslo instance indexu. Postupně se zvětšuje tak, abychom v shaderu mohli pomocí něj indexovat obsah ve storage bufferech.

Pak stačí z pole bytů vytvořit indirect buffer.

A pomocí něj vykreslit obsah našich vertex a index bufferů na obrazovku.

V posledním kroku se už jen do *multi\_draw\_indirect* callu předá indirect buffer jako parametr společně s počtem batchů, který odpovídá počtu obsažených *DrawIndexedIndirect* struktur.

Obrázek - GPU Driven Rendering

## Post processing

### Zadání

Cíl:

* Vytvořit nějaký hezký post processing efekt, který by se dal použít do enginu

Vedlejší cíle:

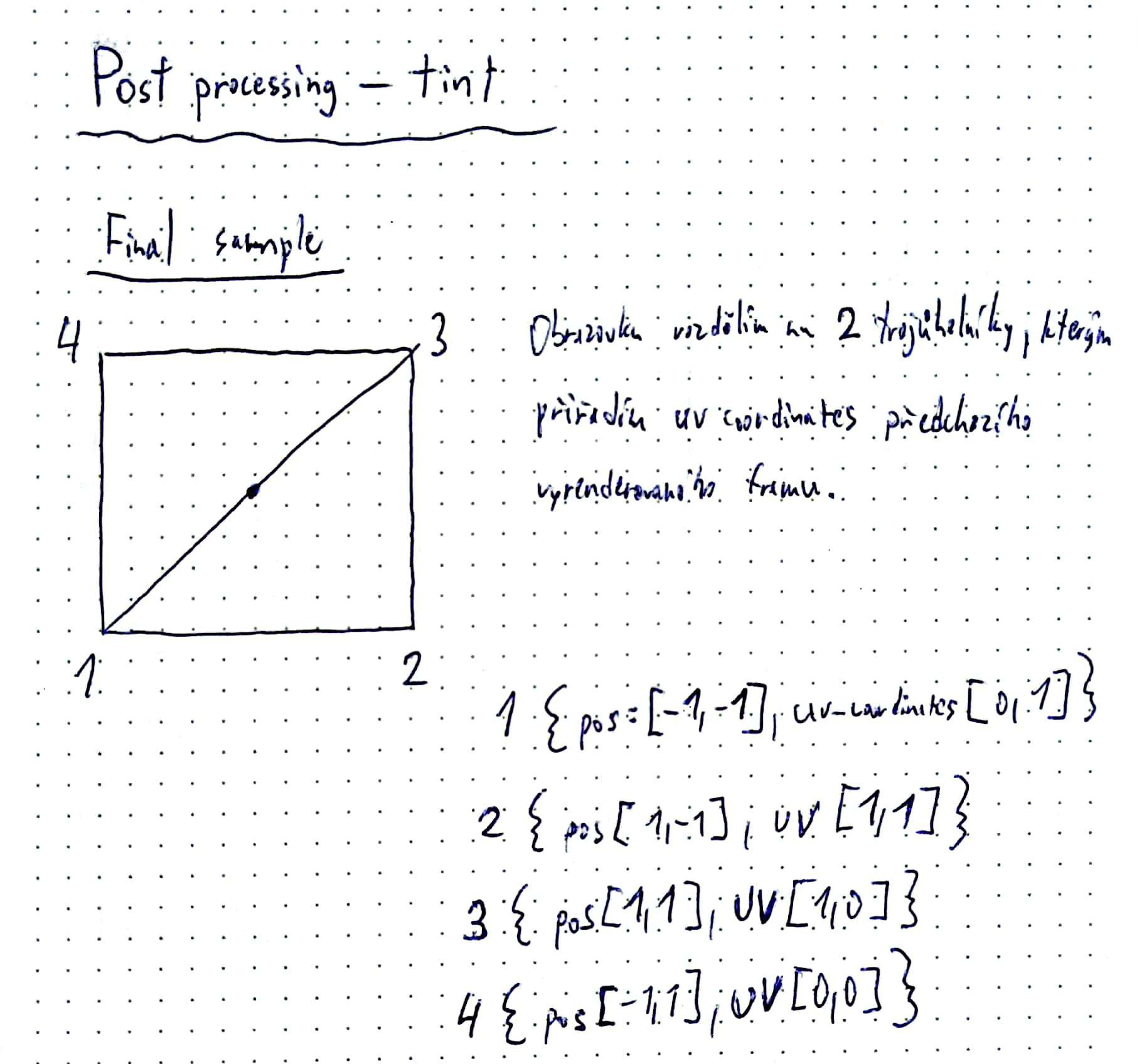
* Vytvořit post processing chain s více efekty, které se budou povolovat a zakazovat pomocí bitových flagů

### Princip

Post processing efekty nebo efekty následného zpracování jsou techniky vykreslování, které jsou použity až po dokončení hlavního vykreslovacího procesu. Čili jsou to operace prováděné na už vykresleném snímku.

Příkladem post processing technik může být color grading (úprava barev), bloom (rozjasnění), motion blur (rozostření pohybu) a mnoho dalších. Abychom mohli aplikovat více efektů najednou, používá se tzv. post processing řetězec. Řetězec může obsahovat několik vrstev efektů, z nichž každá je použita v určitém pořadí, aby byl vytvořen požadovaný efekt.

### Návrh a implementace

Řekl jsem si, že pro začátek vytvořím jednoduchý *Tint* effect[[16]](#footnote-16). Ten funguje tak, že se na začátku zvolí barva a tou se vynásobí všechny pixely předtím vykresleného snímku. Z čehož vznikne scéna, celá laděná do jedné barvy.

Obrázek - Návrh tint efektu

Pro lepší představu jsem si nakreslil obrázek:

Plán byl takový, že si pomocí 2 trojúhelníků namapuji barevné hodnoty jednotlivých pixelů z předchozího snímku, a ještě před jejich vrácením ve fragment shaderu je vynásobím vstupní barvou *Tint* efektu.

Obsah obrázku interiér, světlo

Popis byl vytvořen automatickyK vytvoření obou efektů, které jsou tu vidět jsem se inspiroval v článku ze stránky *Medium.* Abych si ověřil, jak dobře jsem efekty z článku napodobil, použil jsem pro porovnání stejnou výchozí scénu (Obr. 15).

Obrázek - Výchozí scéna pro post processing

Tato scéna se importuje jako texture a potom nalžeme post processing řetězci, že je to námi vykreslená scéna a předáme jí k dalším úpravám.

Závěr

Seznam zkratek a odborných výrazů

Engine

Softwarový engine je software, který poskytuje základní funkce pro ostatní software. Je to vlastně takový „motor“, který pohání nějakou aplikaci a zpravidla je jádrem celé její funkčnosti. Příkladem může být databázový engine, vykreslovací engine nebo herní engine.

Vertex

Vertex je bod v prostoru.

Mesh

Mesh je kolekce bodů, hran a stěn, které dohromady definují nějaký tvar nebo těleso. Příkladem takového tělesa může být koule.

API

Application Programming Interface

Buffer

Vyrovnávací paměť …

Seznam obrázků

[Obrázek 1 - GPU Driven Rendering 3](file:///C:\Users\Oliver%20Řezníček\Documents\Olas\Škola\Maturita\Maturitní%20práce\paralelizace_3d_renderovani\docs\Paralelizace_3D_renderovani.docx#_Toc129798500)

[Obrázek 2 - diagram architektury WebGPU 5](file:///C:\Users\Oliver%20Řezníček\Documents\Olas\Škola\Maturita\Maturitní%20práce\paralelizace_3d_renderovani\docs\Paralelizace_3D_renderovani.docx#_Toc129798501)

[Obrázek 3 – čtverec 9](file:///C:\Users\Oliver%20Řezníček\Documents\Olas\Škola\Maturita\Maturitní%20práce\paralelizace_3d_renderovani\docs\Paralelizace_3D_renderovani.docx#_Toc129798502)

[Obrázek 4 - použití vertex bufferu 9](file:///C:\Users\Oliver%20Řezníček\Documents\Olas\Škola\Maturita\Maturitní%20práce\paralelizace_3d_renderovani\docs\Paralelizace_3D_renderovani.docx#_Toc129798503)

[Obrázek 5 - použití index bufferu 10](file:///C:\Users\Oliver%20Řezníček\Documents\Olas\Škola\Maturita\Maturitní%20práce\paralelizace_3d_renderovani\docs\Paralelizace_3D_renderovani.docx#_Toc129798504)

[Obrázek 6 - WGPU render pipeline 11](file:///C:\Users\Oliver%20Řezníček\Documents\Olas\Škola\Maturita\Maturitní%20práce\paralelizace_3d_renderovani\docs\Paralelizace_3D_renderovani.docx#_Toc129798505)

[Obrázek 7 - UV coordinates 15](file:///C:\Users\Oliver%20Řezníček\Documents\Olas\Škola\Maturita\Maturitní%20práce\paralelizace_3d_renderovani\docs\Paralelizace_3D_renderovani.docx#_Toc129798506)

[Obrázek 8 - Winding order 16](file:///C:\Users\Oliver%20Řezníček\Documents\Olas\Škola\Maturita\Maturitní%20práce\paralelizace_3d_renderovani\docs\Paralelizace_3D_renderovani.docx#_Toc129798507)

[Obrázek 9 - Triangle example 18](file:///C:\Users\Oliver%20Řezníček\Documents\Olas\Škola\Maturita\Maturitní%20práce\paralelizace_3d_renderovani\docs\Paralelizace_3D_renderovani.docx#_Toc129798508)

[Obrázek 10 - Cube example 19](file:///C:\Users\Oliver%20Řezníček\Documents\Olas\Škola\Maturita\Maturitní%20práce\paralelizace_3d_renderovani\docs\Paralelizace_3D_renderovani.docx#_Toc129798509)

[Obrázek 11 - Návrh GPU Driven Rendering 21](file:///C:\Users\Oliver%20Řezníček\Documents\Olas\Škola\Maturita\Maturitní%20práce\paralelizace_3d_renderovani\docs\Paralelizace_3D_renderovani.docx#_Toc129798510)

[Obrázek 12 - GPU Driven Rendering 5](file:///C:\Users\Oliver%20Řezníček\Documents\Olas\Škola\Maturita\Maturitní%20práce\paralelizace_3d_renderovani\docs\Paralelizace_3D_renderovani.docx#_Toc129798511)

[Obrázek 13 - Tint efekt 6](file:///C:\Users\Oliver%20Řezníček\Documents\Olas\Škola\Maturita\Maturitní%20práce\paralelizace_3d_renderovani\docs\Paralelizace_3D_renderovani.docx#_Toc129798512)

[Obrázek 14 - Tint effect 6](file:///C:\Users\Oliver%20Řezníček\Documents\Olas\Škola\Maturita\Maturitní%20práce\paralelizace_3d_renderovani\docs\Paralelizace_3D_renderovani.docx#_Toc129798513)

Použité zdroje

1. **Paralelní výpočty. *Wikipedie.* [Online] https://cs.wikipedia.org/wiki/Paraleln%C3%AD\_v%C3%BDpo%C4%8Dty.**

**2. Microsoft. Rendering from Vertex and Index Buffers (Direct3D 9). *MSDN.* [Online] 1. 6 2021. https://learn.microsoft.com/en-us/windows/win32/direct3d9/rendering-from-vertex-and-index-buffers.**

**3. Blanco, Victor. Storage buffers. *Vulkan Guide.* [Online] https://vkguide.dev/docs/chapter-4/storage\_buffers/.**

**4. Microsoft. Texture Coordinates (Direct3D 9). *MSDN.* [Online] 1. 6 2021. https://learn.microsoft.com/en-us/windows/win32/direct3d9/texture-coordinates.**

**5. Face culling. *Learn OpenGL.* [Online] https://learnopengl.com/Advanced-OpenGL/Face-culling.**

**6. Blanco, Victor. GPU Driven Rendering. *Vulkan Guide.* [Online] https://vkguide.dev/docs/gpudriven/gpu\_driven\_engines/.**

**7. sotrh. Textures and bind groups. *Learn Wgpu.* [Online] https://sotrh.github.io/learn-wgpu/beginner/tutorial5-textures/.**

**8. WebGPU Shading Language. *W3C.* [Online] https://www.w3.org/TR/WGSL/.**

**9. WGPU documentation. *Docs.rs.* [Online] https://docs.rs/wgpu/latest/wgpu/.**

**10. *The Rust Programming Language - book.* [Online] https://doc.rust-lang.org/book/.**

**11. Rust (programovací jazyk). *Wikipedie.* [Online] https://cs.wikipedia.org/wiki/Rust\_(programovac%C3%AD\_jazyk).**

**12. WebGPU. *Wikipedie.* [Online] https://en.wikipedia.org/wiki/WebGPU.**

**13. Tunnel, Raimond. Textures and Sampling. *Computer Graphics.* [Online] https://cglearn.codelight.eu/pub/computer-graphics/textures-and-sampling.**

**14. Chapter 2: The Graphics Pipeline. *GPC.* [Online] https://graphicscompendium.com/intro/01-graphics-pipeline.**

**15. Cincotti, Carmen. Drawing a Triangle with WebGPU. *Carmen's Graphics Blog.* [Online] 18. 4 2022. https://carmencincotti.com/2022-04-18/drawing-a-webgpu-triangle/.**

**16. Pecheux, Mina. Shader Journey #3: Basic Post-Processing Effects. *Medium.* [Online] 22. 9 2021. https://medium.com/geekculture/shader-journey-3-basic-post-processing-effects-e9feb900ceff.**

1. Seznam přiložených souborů

Na přiloženém datovém nosiči se nacházejí následující soubory a složky:

* **Aplikace** – zdrojové kódy

1. Jednotlivými koncepty a syntaxem jazyka Rust se ve své práci nezabývám, takže pokud byste se chtěli o Rustu dozvědět víc a třeba si v něm i zkusit napsat vlastní program, určitě bych pro začátek doporučil e-knihu Rust book, kterou mám uvedenou ve zdrojích. (10) [↑](#footnote-ref-1)
2. Procesor možná není v tomto případě úplně vhodný termín, protože se může jednat pouze   
   o vlákno nebo jádro procesoru. [↑](#footnote-ref-2)
3. Důležitý je pro nás tento termín hlavně proto, že vykreslovací engine bude jedinou částí herního enginu, kterou se tato práce bude zabývat. [↑](#footnote-ref-3)
4. World space je náš 3D svět do kterého jsme si umístili nějaké objekty. Oproti tomu view space je prostor naší 2D obrazovky, kam chceme naše objekty přetransformovat. [↑](#footnote-ref-4)
5. Tady je důležité podotknout, že všechny tyto typy shaderů můžeme psát do jednoho **WGSL** souboru. [↑](#footnote-ref-5)
6. Když zůstaneme u krychle, tak tam je z některých úhlů vidět pouze jedna nebo dvě stěny, proto můžeme ušetřit více jak 50% primitiv. [↑](#footnote-ref-6)
7. Trait v Rustu je něco jako abstraktní třída nebo interface. Umožňuje přidat chování do třídy bez nutnosti použití dědičnosti. V našem případě nám definuje nějaké metody a mohl by i vytvářet jejich defaultní implementaci. [↑](#footnote-ref-7)
8. Z důvodu délky jednotlivých kódů vám ukážu pouze podprogram *run*. [↑](#footnote-ref-8)
9. Zde je použito perspektivní. [↑](#footnote-ref-9)
10. Paralelizací se zde tedy myslí co nejefektivnější využití grafické karty a minimalizace přesunu dat mezi CPU a GPU. [↑](#footnote-ref-10)
11. Instancované kreslení [↑](#footnote-ref-11)
12. Uvnitř se pravděpodobně bude *multi\_draw\_indirect* chovat stejně, jako kdybychom spustili kýžený počet indirect draw callů, ale zase nám o něco víc pomůže snížit potřebnou komunikaci mezi procesorem a grafickou kartou. [↑](#footnote-ref-12)
13. Jakoukoli souřadnici v těchto texturách zvolíme, měla by se nám vrátit stejná barevná hodnota. Takže výsledkem budou 4 jednobarevné objekty. [↑](#footnote-ref-13)
14. Myslím si, že celý problém tkví v této kapitole z WGSL dokumentace:

    <https://www.w3.org/TR/WGSL/#memory-layouts> [↑](#footnote-ref-14)
15. Konkrétně ve funkci *shapes::merge\_index\_vertex\_data*. [↑](#footnote-ref-15)
16. Jako inspiraci pro oba efekty jsem použil článek uvedený ve zdrojích. (16) [↑](#footnote-ref-16)