GYMNÁZIUM JÍROVCOVA 8

MATURITNÍ PRÁCE

Programování grafické aplikace v C++

René Čakan

vedoucí práce: Dr. rer. nat. Michal Kočer

V Českých Budějovicích 6. února 2026 **2025/2026**

Prohlášení

Prohlašuji, že jsem tuto práci vypracoval samostatně s vyznačením všech použitých pramenů.

V Českých Budějovicích dne ..... podpis .....

René Čakan

Abstrakt

Tato práce se zaměřuje na základní vysvětlení počítačové grafiky. Zabývá se architekturou grafických karet a fungováním grafické zobrazovací pipeline. Dále popisuje fungování knihovny OpenGL a programovací jazyky C a C++, ve kterých se s ní často pracuje. V praktické části jsem naprogramoval hru pomocí OpenGL v C++.

Klíčová slova

počítačová grafika, grafická karta, grafická pipeline, OpenGL, C++

Poděkování

Rád bych poděkoval Dr. rer. nat. Michalu Kočerovi za rady a připomínky při vedení mé maturitní práce. Děkuji také své rodině a přátelům, kteří byli mojí emoční podporou.

Obsah

Úvod

Počítačová grafika je poměrně rozsáhlá oblast informatiky, která má širokou škálu uplatnění, jako je vývoj videoher nebo aplikací s grafickým uživatelským rozhraním. Většina programátorů využívá pro vývoj vysoce abstraktní knihovny nebo herní enginy jako *Pygame*, *Unity* či *Unreal Engine*. Vytváření výsledného produktu je díky tomu výrazně jednodušší, avšak samotný proces vykreslování je skryt v kódu dané knihovny či herního enginu. I když jsou tyto knihovny a herní enginy vysoce optimalizované, nesprávné využívání funkcí kvůli neúplnému porozumění vnitřních procesů může vést k zhoršení výkonu aplikace. Cílem této práce je vysvětlení základních principů počítačové grafiky.

První část práce vysvětluje fungování grafických karet a grafické zobrazovací pipeline. Dále popisuje fungování grafické knihovny OpenGL a představuje programovací jazyky C a C++, které jsou s touto knihovnou často používány. Druhá část popisuje fungování jednoduché hry napsané v C++, která využívá knihovnu OpenGL.

Část I

Teorie pro vývoj hry v OpenGL

1 Programovací jazyky

1.1 Programovací jazyk C

C je středněúrovňový programovací jazyk. Je svou podobou blízký strojovému kódu, ale už má prvky vyššího programovacího jazyka jako jsou funkce, datové struktury nebo to, že je strukturovaný. Je kompilovaný a statický, což znamená, že se program musí nejdříve přeložit do strojového kódu a až pak se může spustit. Datové typy jsou známy v čase kompilace, proto musí být všechny proměnné v kódu deklarovány, jelikož vkládání vstupních dat do programu probíhá až při běhu programu. Programuje se v něm strukturovaně a procedurálně, tedy kód se píše pomocí řídicích struktur (if, while, for atd.) a pomocí funkcí, které umožňují používat části kódu vícekrát. C nemá automatický správce paměti, takže je potřeba uvolňovat paměť manuálně. C má střídmou standardní knihovnu, která obsahuje základní matematické operace a funkce pro práci s pamětí a soubory, takže jakékoliv složitější datové struktury či funkce si člověk musí naprogramovat sám. Tato strohost a blízkost ke strojovému kódu z C dělá jeden z nejrychlejších programovacích jazyků. [18, 15, 11, 19, 20]

C bylo vytvořeno Dennisem Ritchiem na počátku 70. let 20. století v AT&T Bell Labs. Jeho předchůdci byly jazyky ALGOL, CPL, BCPL a B. Jeho prvotním účelem bylo přepsat operační systém UNIX do použitelnějšího jazyka než Assembly a B. Už koncem 70. let bylo C populární, ale nebylo standardizované a vznikalo mnoho různých variant. Na začátku 80. let tedy American National Standards Institute (ANSI) zahájil práci na formální standardizované verzi. Tu dokončil v roce 1989 a je známa pod jménem C89. V průběhu let vycházely další verze, které jazyk zlepšovaly a modernizovaly. Nejdůležitější verze byly C99, C11 a C17. Norma C23 byla nedávno schválena a teď se implementuje do kompilátorů. V současnosti mezi nejpoužívanější kompilátory patří *GCC*, *Clang* a *MSVC*. Jelikož bylo C velice populární, ovlivnilo řadu jiných programovacích jazyků, jako C++, C#, Java, Rust, Go atd. [3, 14, 18, 15]

C je univerzální programovací jazyk, má tedy širokou škálu využití. Jeho první využití bylo k napsání UNIXu, který později ovlivnil operační systémy jako Linux, macOS, iOS a Android. Používá se v programování softwaru s omezenou pamětí a výkonem, jako je firmware aut či v zařízeních chytrých domácností. Dále se využívá pro tvoření interpreterů a kompilátorů jako je *GCC* nebo interpreter Pythonu. Také jsou v něm napsané systémové databáze *MySQL* a *Oracle Database*. Kvůli jeho rychlosti jsou v něm napsané knihovny pro jiné programovací jazyky jako je *NumPy*, OpenGL či *GLFW*. [3, 18, 27]

Zde je jednoduchý program, který načte ze vstupu počet čísel, která chce uživatel setřídit. Následně daná čísla načte a vytiskne je seřazená:

```c
#include <stdio.h>
#include <stdlib.h>

int compare(const void *a, const void *b)
{
    return (*(int *)a - *(int *)b);
}

int main(void)
{
    int sizeOfArray;
    scanf("%d", &sizeOfArray);
    int *array = malloc(sizeOfArray *sizeof(int));

    for(int i = 0; i < sizeOfArray; ++i)
        scanf("%d", &array[i]);

    qsort(array, sizeOfArray, sizeof(int), compare);

    for(int i = 0; i < sizeOfArray; ++i)
        printf("%d\n", array[i]);

    free(array);
    return 0;
}
```

Zdrojový kód 1.1: sort_n_numbers.c

1.2 Programovací jazyk C++

Programovací jazyk C++ je v mnoha ohledech podobný jazyku C. Je stejně jako C středně-úrovňový, kompilovaný, statický, má datové typy známé v době kompilace a nemá automatický správce paměti. V C++ se také programuje strukturovaně a procedurálně, ale na rozdíl od C, také umožňuje programovat objektově. Objektové programování umožňuje používat objekty, které jsou instance tříd. Tyto třídy umožňují dědičnost, polymorfismus a zapouzdření, což dělá kód přehlednější a usnadňuje budoucí rozšiřování a debuggování. Dalším rozdílem je standardní knihovna, kterou má C++ rozsáhlejší. Obsahuje nové kontejnery jako `vector`, `map` a `priority_queue`, které jsou tvořeny pokročilejšími datovými strukturami jako binární vyhledávací strom nebo `heap`. Dále obsahuje nové algoritmy, například `sort`, `find` nebo `count`. Kvůli velké podobnosti C a C++ se často může C kód používat v C++, ale není tomu tak vždy. Například tento kód:

```
1 int class(int new, int bool);
```

Zdrojový kód 1.2: incompatibility_example.c

V C tento kód vytvoří funkci `class`, která vrací int a má dva parametry `new` a `bool`. V C++ jsou ale class, new a bool klíčová slova, která nelze použít v názvu proměnných a funkcí. Pokud chce programátor napsat C kód, který lze používat v C++ programech, doporučuje se programovat v C tak, aby daný C kód byl podmnožinou C++. [26, 25]

C++ bylo vytvořeno Bjarnem Stroustrupem v roce 1979 v AT&T Bell Labs. Před vytvořením C++ pracoval Stroustrup s programovacím jazykem Simula 67, který byl objektově orientovaný a sloužil primárně k vytváření simulací. Stroustrupovi přišlo objektově orientované programování velmi užitečné, ale Simula 67 byl příliš pomalý pro větší projekty. Rozhodl se vytvořit nadmnožinu jazyka C, která by umožňovala objektově orientované programování a zároveň si zachovala rychlost C, s názvem C with Classes. V roce 1982 byl Stroustrup se stavem C with Classes zklamán. Nepřišlo mu, že oproti C přináší významné zlepšení a rozhodl se jazyk dále vylepšovat nad rámec objektově orientovaného programování. V roce 1983 byl jazyk přejmenován z C with Classes na C++. Dále bylo C++ v roce 1985 oficiálně vydáno a začalo se používat komerčně. V roce 1998 byla vydána první standardizovaná verze s jménem C++98. Další významné verze, které jazyk modernizovaly a přidávaly mu nové funkce, byly C++03, C++11, C++14, C++17, C++20 a nejnovější verze C++23. C++ se

kompiluje pomocí stejných kompilátorů jako C, tedy *GCC*, *Clang* a *MSVC*. [25, 24, 2, 29]

C++ je stejně jako C univerzální programovací jazyk, a využívá se v široké škále odvětví. První využití je ve videoherním průmyslu. V C++ jsou napsané populární herní enginy jako *Unity* nebo *Unreal Engine*. Dále se v něm vytváří aplikace jako *Photoshop* nebo *Blender*. Využívá se v částech operačních systémů jako Apple macOS nebo Microsoft Windows OS. Dále se využívá při vytváření internetových prohlížečů, například Firefox nebo Google Chrome. C++ se využívá i ve vědě, například v CERNu nebo v NASA. [26, 12, 23]

Program se stejnou funkcí jako z kapitoli o C, ale napsán v C++

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main()
{
    int sizeOfArray;
    std::cin >> sizeOfArray;
    std::vector<int> arr(sizeOfArray);

    for(int i = 0; i < sizeOfArray; ++i)
        std::cin >> arr[i];

    std::sort(arr.begin(), arr.end());

    for(int i : arr)
        std::cout << i << '\n';

    return 0;
}
```

Zdrojový kód 1.3: sort_n_numbers.cpp

2 Počitačová grafika

2.1 Grafické karty

2.1.1 CPU vs. GPU

Tradičně běžela většina aplikací sekvenčně na procesorech s centrální výpočetní jednotkou (CPU), která provádí instrukce jednu po druhé. V průběhu 20. století se výkon CPU významně zlepšoval až na bilion operací za sekundu, čímž se mohly zlepšovat grafické aplikace a využívat náročnější funkce. Na přelomu tisíceletí se ale vývoj začal zpomalovat, kvůli problému se spotřebou energie a odvodem tepla. Proto výrobci začali vytvářet CPU s více jádry, ale ani to nebylo dostatečně výkonné na složité výpočty, a proto už v 70. a 80. letech začaly vznikat počítače, které nepracovaly sekvenčně, ale paralelně. V 90. letech se začaly vytvářet mikroprocesory, které se soustředily na paralelní výpočty (GPU). V průběhu let se počet jader v GPU dostal z jednotek na tisíce, což vytvořilo prostor pro inovaci v grafice a tvorbě softwaru. [17]

Prvně bylo GPU pouze v počítačích specializovaných pro 3D hry a vizualizace, ale postupně se stala součástí většiny počítačů. GPU a CPU se používají v různých případech kvůli rozdílné architektuře. CPU má jednotky až desítky jader a umí dělat jeden krok extrémně rychle kvůli sekvenčnímu zapojení. Velká část čipu je určena pro cache, tedy malou paměť, kam se ukládají data, která budou pravděpodobně v budoucnosti potřeba. Díky tomu není třeba tak často komunikovat s pamětí s náhodným přístupem (RAM). Část čipu je také určena pro řídící jednotku, která umožňuje provádět instrukce ve vláknech paralelně nebo v jiném sekvenčním pořadí, a přitom zachovat sekvenčnost celého procesu. CPU má také menší propustnost paměti než GPU, proto jsou programy s optimalizovaným přístupem do cache výrazně rychlejší. Na rozdíl od toho má GPU stovky až tisíce jader. Jednotka řídící logiku a cache jsou v jádru menší a jednodušší, takže reagují pomaleji na události. Po dokončení operací se většinou data vrací zpět do video RAM (VRAM), což je díky velké propustnosti

i pro velmi velký objem dat rychlé. [17]

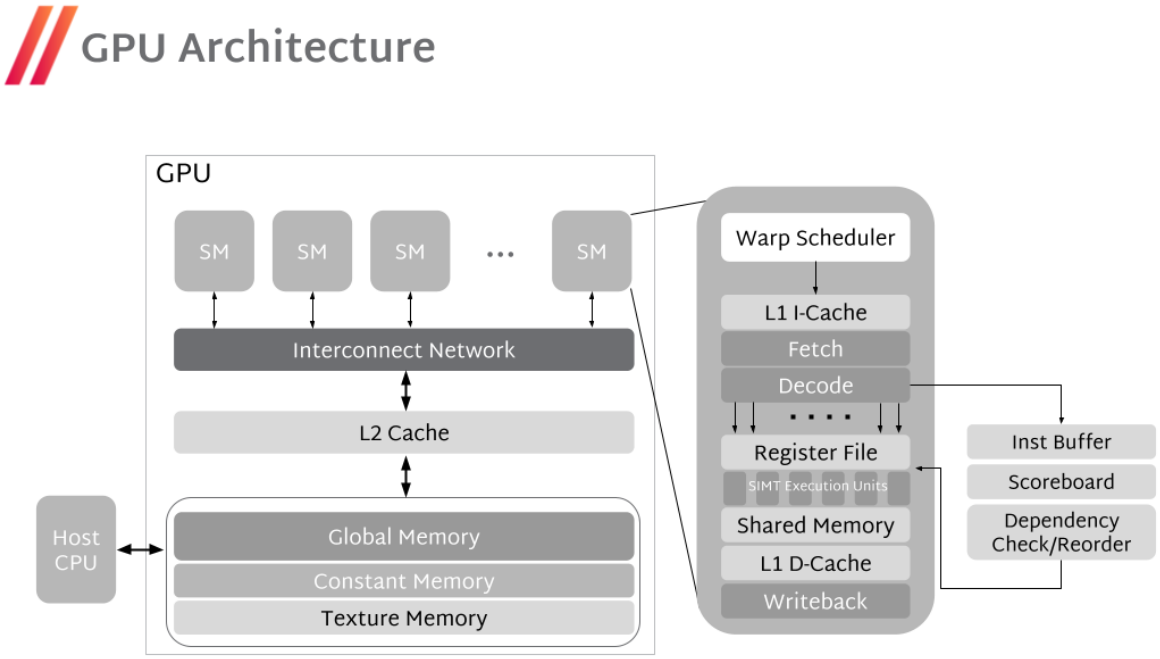
Do roku 2007 se GPU používalo převážně pro vykreslování, ale v roce 2007 přišla NVIDIA s programovacím modelem pro paralelní výpočty *CUDA*. Tento model umožňuje psát programy, které běží na GPU a využívají CPU na přenos dat a řízení těchto programů. Díky tomu mohly začít vznikat aplikace, které běží na CPU, ale při potřebě náročnějších početních operací mohou běžet na GPU, a tím zvýšit výkon. Pro účel vysvětlení programovacích modelů pro paralelní výpočty budu využívat model CUDA, přičemž existují i jiné podobné modely jako *OpenCL* nebo *SYCL*. [17]

2.1.2 Architektura a paměťová hierarchie GPU

GPU s podporou CUDA je organizováno do pole paralelních streamovacích multiprocesorů (SM). Každý SM se skládá z několika streamovacích procesorů (SP), které jsou základní výpočetní jednotkou, na nichž probíhají aritmetické operace. Další součástí jsou řídicí jednotky, které načítají instrukce pro konkrétní SM. Řídicí jednotky také obsahují malou instrukční cache, aby se pro instrukce nemuselo často přistupovat k VRAM, což by zpomalovalo výpočetní cykly. Dále je uvnitř SM warp scheduler, který řídí, jaké skupiny vláken budou v každém výpočetním cyklu použity. [17]

Většina grafických karet má vlastní dynamickou RAM (DRAM), ke které mají přístup všechny SM a nazývá se globální paměť. V grafických kartách se nachází specifický typ DRAM zvaný VRAM. Tato paměť má jednotky až desítky gigabytů a dochází v ní k výměně dat mezi systémovou pamětí a VRAM. Velká propustnost této paměti je zajištěna blokovým načítáním dat namísto po jednotlivých bytech. Samotná operace výměny dat mezi systémovou pamětí je ale poměrně náročná, a proto existuje hierarchie menších pamětí, jejichž cílem je minimalizovat počet přístupů do systémové paměti. Jedním z těchto typů je sdílená paměť, která má většinou desítky kilobajtů. Tuto paměť má každý SM a každý SP k ní může přistupovat. V každém SM se také vyskytuje L1 cache, která slouží k uchovávání dat, ze kterých se nedávno četlo nebo byla změněna. Dále existuje L2 cache, která je větší než L1 cache a je jedna pro všechny SM. Používá se pro uchovávání dat nedávno získaných nebo poslaných do VRAM. Část VRAM je také určena pro dva jiné typy paměti, a to texture a constant memory. Obě paměti mají k sobě přiřazenou svoji cache, která je pouze pro čtení. Díky tomu lze často používané hodnoty uchovávat v cache bez obavy z toho, že by se daná proměnná mohla v průběhu změnit. Nejmenší a nejrychlejší částí paměti jsou register files.

Tato paměť přímo komunikuje s SP a ukládají se do ní dočasné proměnné a mezivýpočty. [17, 5, 28]



Obrázek 2.1: Typická architektura GPU [6]

2.1.3 Paralelismus

Paralelismus je výpočetní technika, která umožňuje vykonávání více operací najednou. Paralelismus se dá rozdělit do dvou kategorií. První z nich je úlohový paralelismus. Tento typ následuje Multiple Instructions, Multiple Data (MIMD) model. Tento model rozdělí různé úlohy mezi více vlákny nebo i jádry. Tento model se využívá, pokud na sobě nejsou úlohy závislé. Pokud by se například chtěla vypočítat suma pole s čísly, může každé vlákno počítat jednu polovinu a oba výsledky sečíst. MIMD se využívá v simulacích nebo ve webových aplikacích. [17, 8]

Druhou metodou je datový paralelismus, který je definován Single Instruction, Multiple Data (SIMD) modelem. Tento model aplikuje jednu úlohu pro velké množství dat a využívá se v GPU. Používá se při násobení matic, kde se daná matice dá rozdělit na menší části, nebo při vykreslování, kdy se jeden shader aplikuje pro každý pixel. V grafických kartách se jednotlivá vlákna sdružují do warpů, ty dále do bloků, které plánuje SM. Do každého warpu jsou načteny stejné instrukce a postupně se mu dodávají data, na kterých dané instrukce provádí. Díky tomu lze i velké množství dat zpracovat velmi rychle, protože se dodávají

s velkou propustností a instrukce jsou na nich prováděny současně na velkém množství warpů. [17, 8]

2.2 Grafická zobrazovací pipeline

Hlavní funkcí grafické zobrazovací pipeline je vykreslit dvourozměrný obraz z trojrozměrných objektů, pozice virtuální kamery, zdrojů světla atd., který je schopen být zobrazen například na monitoru. Pipeline se skládá z několika fází. Tyto fáze probíhají paralelně jak v rámci jednotlivých fází, tedy že jedna fáze probíhá pro větší množství dat najednou, tak i mezi jednotlivými fázemi. Po průchodu jednoho bloku dat jednou fází data přecházejí do další fáze, i když pro jiná data ještě předchozí fáze nemusí být ukončena. Hlavní čtyři fáze jsou aplikační fáze, geometrické zpracování, rasterizace a zpracování pixelů. [1]

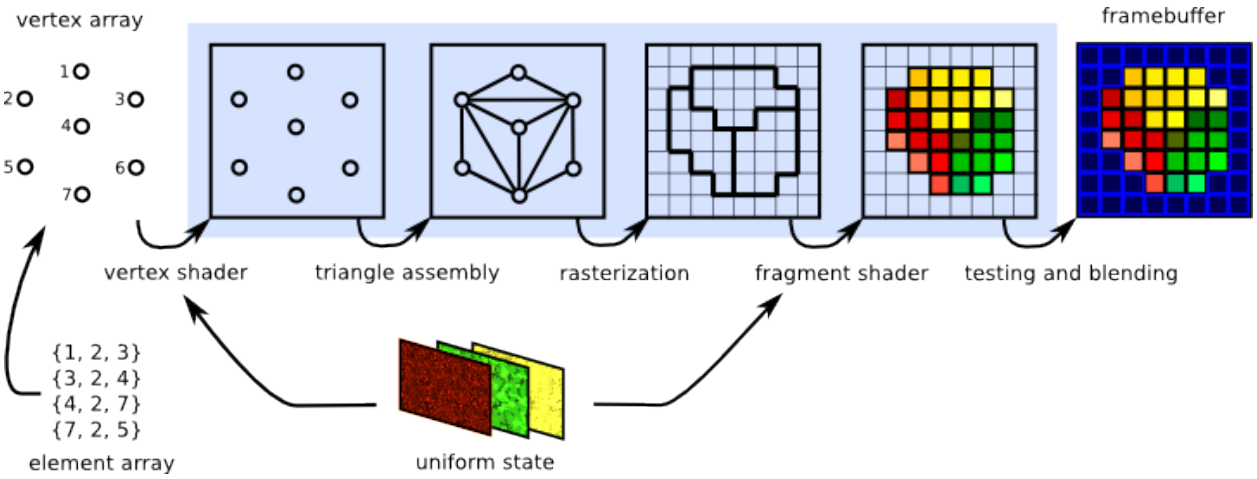
Aplikační fáze je řízena samotnou aplikací a je většinou implementována jako software běžící na CPU. Tato fáze nemá žádné podprvky, a proto může běžet paralelně na různých jádrech. V této části se zpracovávají vstupy od uživatele, které například pozměňují matici k posunutí, otočení nebo změně velikosti různých objektů. Také zde dochází k detekci kolizí, akceleračním algoritmům, fyzikálním simulacím a dalším výpočtům, které připravují data pro další scénu. [1]

Další částí pipeline je geometrické zpracování. Ta je zodpovědná za většinu operací pro každý trojúhelník a vrcholy, které tvoří objekty. Rozděluje se na čtyři podfáze: vertex shading, projekce, clipping a mapování na obrazovku. Vertex shading vypočítává pozici vrcholu a připravuje jeho data, jako barvy a textury. Objekt je transformován z modelového prostoru do světového prostoru, tedy souřadnic relevantních pro všechny objekty ve scéně. Dále se transformuje do kamerového prostoru, ve kterém je kamera v bodě nula. Následuje projekce, při které dochází k transformaci z kamerového prostoru do clip prostoru, aby bylo možné provést clipping. Clipping určuje, jaké objekty se budou vykreslovat. Pokud by například celý objekt byl mimo zorné pole scény, není vykreslován, aby zbytečně nezatěžoval GPU. Poslední částí je mapování na obrazovku, které převede souřadnice z clip prostoru do souřadnic obrazovky připravených pro rasterizaci. Součástí vertex shadingu jsou také volitelné fáze vertex zpracování. Tessellation generuje vhodné trojúhelníky pro zakřivené povrchy, aby vypadaly realističtěji. Geometry shader umožňuje podle potřeby vytvářet nové vrcholy z existujících trojúhelníků nebo bodů. To se využívá například u malých částic, které se rozšíří na malé čtverce, aby byly lépe vidět na obrazovce. Poslední volitelná fáze je stream output, která

umožňuje namísto poslání vrcholu do další fáze uložit vrcholy do bufferu, na kterých CPU může dělat další výpočty či simulace. [1]

Následující fáze je rasterizace, jejímž účelem je nalézt všechny pixely uvnitř vykreslovaného trojúhelníku. Rasterizace se dělí na přípravu trojúhelníku a průchod trojúhelníkem. Při přípravě se vypočítávají diferenciály, rovnice hran a jiná podobná data, která jsou využívána ke zjišťování, zda daný pixel leží uvnitř trojúhelníku. Při průchodu se zjišťuje, zda střed pixelu leží uvnitř trojúhelníku, a pokud ano, vytvoří se pro daný pixel fragment. Vlastnosti jako barva, hloubka atd. jednotlivých fragmentů se získávají interpolací mezi vrcholy daného trojúhelníku a následně se ukládají do specifických bufferů. Výsledné fragmenty jsou následně předány do fáze pixelového zpracování. [1]

Pixelové zpracování se dělí na pixelové stínování a slučování. Ve fázi pixelového stínování se provádějí výpočty stínování jednotlivých pixelů ze stínovacích dat. Výsledkem je jeden nebo více barevných výstupů. V této fázi se používá hodně specifických technik, jako například texturování. Pixelové stínování je na rozdíl od ostatních fází, které jsou dány architekturou GPU, dáno programovatelnými jádry GPU. Při slučování dochází ke kombinaci výstupu pixelového stínování a uložených barev v bufferech. V této fázi se také řeší viditelnost, aby se vykresloval pro daný pixel fragment s nejmenší hloubkou. [1]



Obrázek 2.2: Vizualizace grafické zobrazovací pipeline GPU [10]

2.3 Transformace

V průběhu vykreslování je často potřeba s objekty různě pohybovat. Mohli bychom pozměňovat souřadnice původních vrcholů a opětovně konfigurovat jejich buffery, ale to je výpočetně náročné. Místo toho se využívají transformace pomocí matic. Matice jsou obdélníková pole

matematických výrazů. Souřadnice objektů se ukládají do 4D vektorů a transformace do 4x4 matic, které se mezi sebou vynásobí. [30]

$$\begin{pmatrix} m_{00} \cdot x + m_{01} \cdot y + m_{02} \cdot z + m_{03} \cdot w \\ m_{10} \cdot x + m_{11} \cdot y + m_{12} \cdot z + m_{13} \cdot w \\ m_{20} \cdot x + m_{21} \cdot y + m_{22} \cdot z + m_{23} \cdot w \\ m_{30} \cdot x + m_{31} \cdot y + m_{32} \cdot z + m_{33} \cdot w \end{pmatrix} = \begin{pmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & m_{33} \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

Mezi základní operace patří translace, rotace a škálování. Translace je operace, při které se k původnímu vektoru přičte jiný vektor, čímž se původní objekt posune. Potřebujeme translační hodnoty násobit složkou w vektoru, která je nastavena na 1, aby neupravila tyto hodnoty. Proto používáme 4D vektor, protože s 3D vektorem by translace nebyla možná.

$$\begin{pmatrix} x + T_x \\ y + T_y \\ z + T_z \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Další operací je škálování. Při této operaci se dané souřadnice násobí konstantou, čímž se může daný objekt zmenšit nebo zvětšit.

$$\begin{pmatrix} S_x \cdot x \\ S_y \cdot y \\ S_z \cdot z \\ 1 \end{pmatrix} = \begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Poslední důležitá operace je otáčení. Při této operaci se objekt otáčí podél jedné ze tří možných os.

Rotace podle osy X:

$$\begin{pmatrix} x \\ \cos\theta \cdot y - \sin\theta \cdot z \\ \sin\theta \cdot y + \cos\theta \cdot z \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Rotace podle osy Y:

$$\begin{pmatrix} \cos\theta \cdot x + \sin\theta \cdot z \\ y \\ -\sin\theta \cdot x + \cos\theta \cdot z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Rotace podle osy Z:

$$\begin{pmatrix} \cos\theta \cdot x - \sin\theta \cdot y \\ \sin\theta \cdot x + \cos\theta \cdot y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Při provádění transformací záleží na pořadí. Pokud je pro transformaci jednoho objektu více matic, pronásobí se matice mezi sebou a používá se výsledná matice. Při tvoření takové matice se ale postupuje v opačném pořadí násobení. Pokud bychom pro vektor $\mathbf{V}$ chtěli udělat operace a, b a c, tak to lze udělat následovně:

$$\mathbf{V}' = \mathbf{C} \cdot (\mathbf{B} \cdot (\mathbf{A} \cdot \mathbf{V}))$$

nebo

$$\mathbf{M} = \mathbf{C} \cdot \mathbf{B} \cdot \mathbf{A}$$

$$\mathbf{V}' = \mathbf{M} \cdot \mathbf{V}$$

3 OpenGL

3.1 Historie OpenGL

OpenGL je považováno za aplikační programovací rozhraní (API), které uživatelům poskytuje sadu funkcí, které mohou používat k manipulaci s grafickou kartou. Specifikaci pro OpenGL tvoří společnost Khronos Group, která určuje, jaké funkce existují a jak se budou chovat. Poté každá společnost implementuje OpenGL API pro svůj druh grafických karet. [30]

3.2 OpenGL pipeline

Grafická pipeline je velmi podobná obecné grafické pipeline popsané v předchozí kapitole. Uživatel je schopen kontrolovat programovatelné fáze pipeline jako jsou shadery. Ostatní fáze jsou definovány API a nelze je měnit. [30]

3.3 Shadery

3.3.1 Co to je shader

Shader je malý program, který běží na GPU. Tyto programy se spouštějí pro konkrétní část grafické pipeline. Shadery přeměňují vstupy a převádějí je na výstupy potřebné pro následující fáze. Starší verze OpenGL měly výchozí shadery, ale v novějších verzích je uživatel povinen vytvořit základní shadery, aby GPU mohlo něco vykreslit. Shadery jsou od sebe izolované a komunikují spolu pouze pomocí vstupů a výstupů. Shaderů je více druhů a dva základní jsou vertex a fragment shader. [30]

3.3.2 Vertex shader

Základní účel vertex shaderu je zpracovávání vrcholů objektu. Do vertex shaderu vstupují atributy vrcholů, jako barvy nebo 3D souřadnice bodů v modelovém prostoru. Vertex shader provádí transformace těchto souřadnic do světového, kamerového a následně clip prostoru pomocí transformačních matic. Vstupní data jsou uložena v GPU, kde k nim má vertex shader přímý přístup. [30]

3.3.3 Fragment shader

Účelem fragment shaderu je vypočítávání barevného výstupu pro pixely. Barvy se v OpenGL definují 4D vektorem s floaty od 0.0 do 1.0. Tato čísla udávají sílu barev červená, zelená a modrá (RGB). Poslední číslo udává alfa hodnotu, tedy jak moc bude kombinace daných tří barev průhledná. Ve fragment shaderu také dochází k texturování a různým efektům, jako jsou odlesky, stínování atd. [30]

3.3.4 Buffery a linkování

Vstupní data vertex shaderu se ukládají do paměti GPU pomocí Vertex Buffer Objektu VBO. Ten v sobě dokáže uložit velké množství dat, díky čemuž jich posíláme více najednou a využíváme vysokou propustnost GPU. Tato data jsou pak jednoduše přístupná pro vertex shader, který s nimi v GPU pracuje. Následně se propojují vstupní data, což udává, jak jsou data propojena k proměnným uvnitř vertex shaderu. Dalším důležitým objektem je Vertex Array Object VAO, do kterého se ukládá, jak jsou data v VBO uspořádána a jak se mají propojit s vertex shadery. [30]

Shadery se kompilují a linkují a následně ukládají do paměti GPU jako součást shader programu. To je objekt, který je výslednou verzí všech shaderů po linkování. Linkování shaderů propojuje výstupy jednoho shaderu se vstupy druhého. Uživatel může mít více shader programů a pro každý objekt si vybrat ten, který obsahuje požadované shadery. [30]

3.3.5 GLSL

Shadery se píší v programovacím jazyce OpenGL Shading Language GLSL, který je podobný jazyku C. GLSL je specificky navržen pro práci s grafikou a obsahuje užitečné funkce na manipulaci s vektory a maticemi. Pokud jsou na sebe propojené dva shadery, výstupy

z prvního shaderu se přenesou do druhého. Proměnné ovšem musí mít identický typ a velikost. Další důležitou funkcí jsou uniformy, což jsou globální proměnné pro všechny shadery uvnitř jednoho shader programu. V uniformách se často uchovávají matice a další konstanty. [30]

Ukázka vertex a fragment shaderu naprogramovaného v GLSL z mojí hry:

```
#version 330
layout(location = 0) in vec3 pos;
layout(location = 1) in vec2 tex;

out vec2 TexCoords;
uniform mat4 transform;

void main()
{
    gl_Position = transform * vec4(pos, 1.0);
    TexCoords = tex;
}
```

Zdrojový kód 3.1: vertex shader `fruit.vs`

```
#version 330 core
in vec2 TexCoords;
out vec4 FragColor;
uniform sampler2D myTexture;

void main()
{
    FragColor = texture(myTexture, TexCoords);
}
```

Zdrojový kód 3.2: fragment shader `fruit.fs`

3.4 Textury

Textura je obrázek používaný k přidávání detailů k objektu. Pro každý vrchol objektu přiřadíme souřadnici textury, která určuje, jaká část textury odpovídá danému vrcholu. Souřadnice textury se ukládají do vstupních dat spolu s pozicemi vrcholů. Textura se pak aplikuje ve fragment shaderu na každý fragment objektu, čímž se promítne. Je také možno mít původní barvu objektu s jednou nebo více texturami na jednom objektu. Následně se barvy a textury poměrově mixují podle nastavených alfa hodnot. [30]

Část II

Vývoj hry v OpenGL

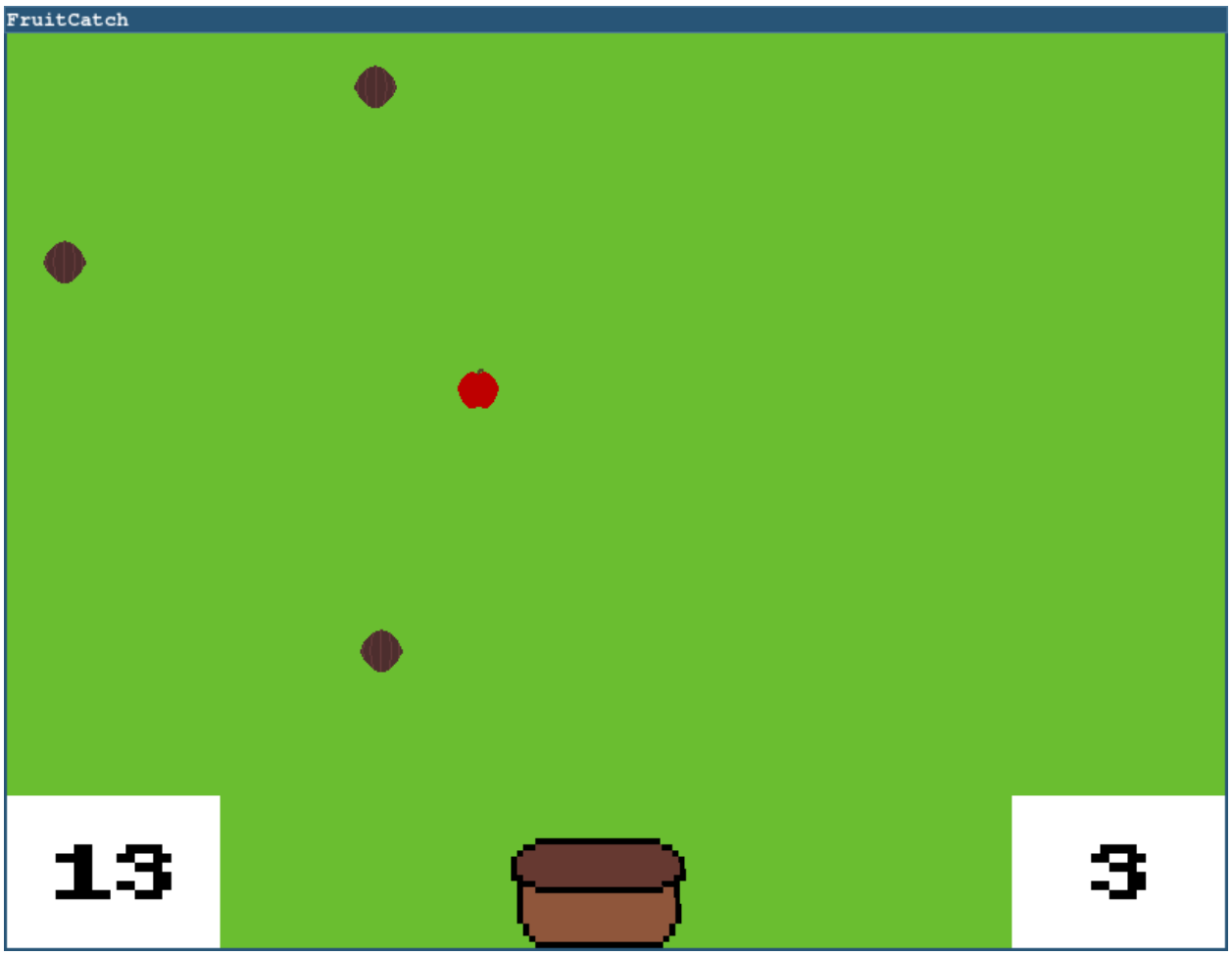
4 Architektura hry

Pro vytváření své hry jsem si vybral OpenGL, protože je to nízkoúrovňového API, jehož použití vyžaduje základní porozumění grafické pipeline. Jako programovací jazyk jsem si vybral C++, protože je s C knihovnami dobře kompatibilní, ale přitom mohu využívat prvky objektově orientovaného programování, které jsou ve vývoji hry užitečné.

Moje hra má dva základní stavy. Prvním z nich je menu, ve kterém se uživatel nachází při prvním zapnutí hry. Po kliknutí na tlačítko se uživatel přesouvá do druhého stavu běžící hry. Při běhu hry je cílem hráče pohybovat košíkem a sbírat jablka, což zvyšuje skóre. Zároveň je cílem hráče vyhýbat se padajícím kokosům, které hráči odebírají životy. Tento stav běží, dokud uživateli nedojdou všechny životy, a následně je přesunut zpět do herního menu. Hra je tvořena větším počtem souborů, které níže vysvětlím.

- `main.cpp` – Vytváří herní okno a spouští hlavní smyčku.
- `window.cpp` – Inicializuje GLFW a spravuje okno aplikace.
- `game.cpp` – Řídí přepínání herních stavů a resetuje proměnné při restartu hry.
- `game_menu.cpp` – Vykresluje stav menu a předává informaci, zda byla hra spuštěna.
- `game_running.cpp` – Vykresluje stav běžící hry a volá funkce pro objekty ve hře.
- `shader.cpp` – Spravuje shadery, tedy kompilaci, linkování, aktivaci a nastavování uniforem.
- `rendering_function.cpp` – Poskytuje funkce pro načítání shaderů a propojování vstupních dat s VBO a VAO.
- `texture.cpp` – Spravuje vytváření, aktivaci a nastavení parametrů textur.
- `vertices.cpp` – Uchovává vstupní data, jako jsou souřadnice objektů a textur.

- `fruit.cpp` – Spravuje vytváření, pohyb a mazání ovoce.
- `player.cpp` – Spravuje pohyb hráče.
- `text.cpp` – Spravuje vytváření a aktualizaci textu pro počítání životů a bodů.
- `collision_detection.cpp` – Zjišťuje, zda dochází ke kolizi hráče s ovocem, a případně volá aktualizaci životů a skóre.
- `random.cpp` – Vytváří náhodná čísla typu double a float.



Obrázek 4.1: Ukázka běžící hry

5 Použité programy a knihovny

5.1 CMake

CMake je program, který generuje nativní soubory sestavovacího systému pro různé platformy. Do textového souboru `CMakeLists.txt` se zapisují zdrojové soubory, cesty ke knihovnám, které program používá, a také konfigurace verze C++, kterou chce uživatel používat. CMake tyto informace předá kompilátoru, takže uživatel nemusí vypisovat pro každou kompilaci všechny závislosti svého programu. Další výhodou je, že sestavení vytvořené pomocí CMake může fungovat napříč platformami. [4]

5.2 GLAD

GLAD je knihovna, která generuje zavaděč na základě oficiální specifikace OpenGL. Od verze OpenGL 1.1 nejsou moderní funkce přímo v systému, takže nejdou importovat do programu jako knihovna, ale jsou implementovány v ovladačích grafických karet. Kvůli tomu by kompilátor nebyl schopen najít deklarace těchto OpenGL funkcí. Proto se využívají zavaděče jako GLAD, které načtou tyto funkce a skrze hlavičkový soubor je vloží do C++ programu. [16]

5.3 GLFW

GLFW je knihovna určená pro vývoj aplikací využívajících OpenGL nebo Vulkan. Hlavním účelem GLFW je otevírání, správa a nastavení vlastností oken. Pro každé okno je vytvořen kontext, který obsahuje stav grafiky spojený s daným oknem. GPU při vykreslování čerpá z kontextu uchovaný stav, který funguje jako prostředník mezi programem a GPU. Další důležitou funkcí je zpracovávání vstupů uživatele, jako je stisk klávesy či myši, čímž je program schopen reagovat na interakce uživatele s oknem. [13]

5.4 GLM

GLM je matematická knihovna určená pro grafický software. GLM poskytuje funkce a třídy implementované se stejnými názvy a funkcemi jako GLSL, což usnadňuje práci se shadery. GLM se převážně používá pro práci s maticemi a obsahuje funkce pro operace s nimi, což ulehčuje transformace objektů. GLM také obsahuje generátor náhodných čísel nebo kvaterniony, které slouží k provádění rotací ve 3D prostoru. [22]

5.5 stb_image

Knihovna *stb_image* je jednoduchá knihovna pro načítání obrázků. Dokáže zpracovat většinu běžných formátů obrázků, jako jsou PNG nebo JPG. Tyto obrázky ukládá jako surová data do paměti. Pro každý pixel jsou uložena 3 čísla při používání RGB nebo 4 čísla při RGBA. [9]

6 Herní mechaniky

6.1 Generátor čísel

Pro mou hru jsem potřeboval nějaký generátor náhodných čísel pro počáteční x souřadnici ovoce nebo pro rychlost, jakou bude padat. Moje třída využívá pseudonáhodný generátor mt19937. Tento generátor po dodání seedu vygeneruje náhodnou sekvenci čísel. Pro vytvoření seedu používám funkci random_device, která vygeneruje náhodné číslo z náhodných parametrů systému. Generování čísel podle parametrů systému je náhodnější, ale tento proces je poměrně pomalý, a proto se využívá jen na generování seedu. Pro vytvoření čísla z nějakého rozmezí se používá funkce uniform_real_distribution. Ta převádí náhodně vygenerované číslo z generátoru na reálné číslo z požadovaného intervalu. Minimální hodnota generátoru odpovídá dolní hranici intervalu a maximální hodnota odpovídá horní hranici. Vygenerované číslo se následně přemapuje z intervalu generátoru do cílového intervalu. [7]

6.2 Generování ovoce

Pro generování a práci s ovocem jsem si vytvořil dvě pomocné struktury.

```
1  enum class FruitType
2  {
3      apple = 0, coconut = 1
4  };
5
6  struct fruitAttributes
7  {
8      FruitType type;
9      glm::mat4 transform;
10     float speed;
11 };
```

Zdrojový kód 6.1: pomocne struktury

`FruitType` slouží k tomu, aby se přiřadila správná textura k danému ovoci a aby se vedělo, jaký efekt nastane při kolizi. Ve `fruitAttributes` jsou všechny potřebné parametry pro každé ovoce pohromadě, takže se s nimi lépe pracuje. Poté si vytvořím deque, do kterého si ukládám všechno ovoce, které je na obrazovce. Při vytváření ovoce vygeneruji náhodnou rychlost a matici, ve které je náhodná hodnota posunutí ve směru osy X. Pro mazání a kolize následně vždy jen iteruji skrz všechno ovoce a zjišťuji, zda se dostalo pod hranu obrazovky nebo zda kolidovalo s košíkem.

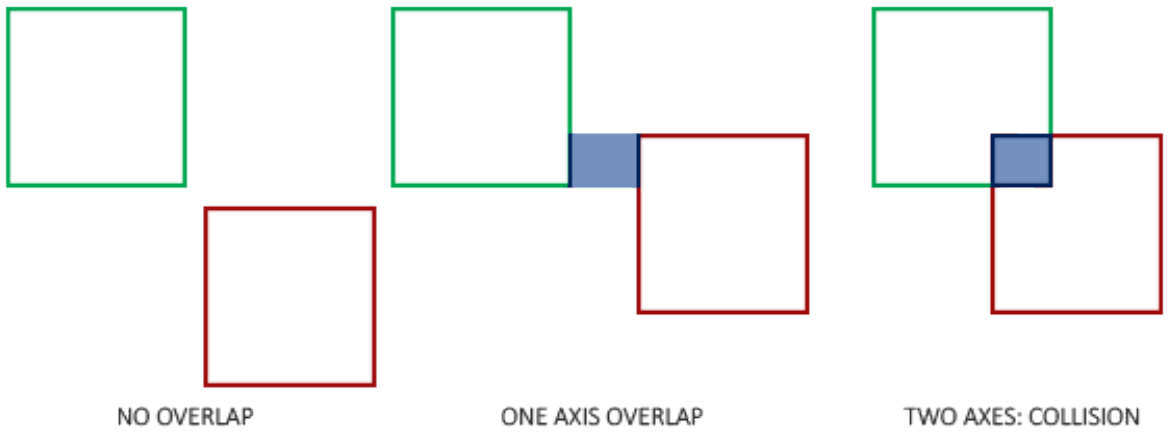
6.3 Vykreslování čísel

Pro počítání skóre a životů jsem si vytvořil třídu pro vykreslování čísel. Protože čísla, která potřebuji vykreslit, mohou mít libovolnou hodnotu, nevyplatí se pro každé číslo vytvářet samostatnou texturu. Místo toho jsem využil techniku zvanou texture atlas. Tato technika využívá jednu velkou texturu s číslicemi od 0 do 9. Každá cifra vykreslovaného čísla je samostatný objekt, kterému je přiřazena část textury s odpovídající číslicí. Aby byla číslice správně vycentrovaná, musí se každému objektu přiřadit transformační matice. Pozice jednotlivé číslice závisí na celkové délce vykreslovaného čísla a také na pozici dané cifry v čísle. V shaderu se pomocí posunu texturových souřadnic určuje, jaká část textury se má vykreslit na daný objekt.

Obrázek 6.1: Můj texture atlas pro vykreslování čísel

6.4 Detekce kolize

Pro detekci kolizí jsem využil AxisAligned Bounding Box (AABB) detekci. Každý objekt je reprezentován dvojicí bodů (horní levý roh a dolní pravý roh), které určují obdélník zarovnaný s osami. Kolize se zjišťuje porovnáním intervalů obou obdélníků na osách X a Y. Pokud dojde u obou os k překryvu, objekty mezi sebou kolidují.



Obrázek 6.2: AABB collision detection [1]

Závěr

Cílem první části práce bylo čtenáři přiblížit fungování grafických karet a grafické zobrazovací pipeline. Dále byly popsány principy grafické knihovny OpenGL a programovacích jazyků C a C++. Rozsah práce neumožňuje popsat fungování OpenGL tak dopodrobna, aby byl čtenář schopen naprogramovat vlastní aplikaci. Pro hlubší porozumění dané knihovny lze využít knihu Learn OpenGL.

V druhé části práce je popsána vytvořená hra v jazyce C++. Hra využívá teorii z předchozí části a základní OpenGL funkce pro práci s shadery a texturami, převádění souřadnic skrz různé souřadnicové systémy a práci s transformačními maticemi.

Vzhledem k složitosti jazyka C++ kód není plně optimalizovaný, avšak pro hru takto malého rozsahu je výkon dostačující.

Hra by se dala rozšířit například přidáním více typů ovoce nebo zobrazováním hráči nejvýše dosaženého skóre v menu, ale pro účely demonstrace popsané teorie je rozsáhlost dostačující.

Bibliografie

1. AKENINE-MÖLLER, Tomas et al. *Real-Time Rendering*. 4th. A K Peters/CRC Press, 2018. ISBN 9781138627000.
2. ALBATROSS. *History of C++* [Online]. [B.r.]. Dostupné také z: `https://cplusplus.com/info/history/`. [citováno 2025-10-17].
3. BANAHAN, Mike; BRADY, Declan; DORAN, Mark. *The C Book: Featuring the ANSI C Standart(Instruction Set)*. 2nd. Addison-Wesley, 1991. ISBN 9780201544336.
4. CMAKE.ORG. *Software Development with CMake* [Online]. [B.r.]. Dostupné také z: `https://cmake.org/about/`. [citováno 2025-01-08].
5. COMPUTE, Arc. *Memory Hierarchy of GPUs* [Online]. 2023. Dostupné také z: `https://www.arccompute.io/arc-blog/gpu-101-memory-hierarchy`. [citováno 2025-12-13].
6. COMPUTING, Scale. *Understanding GPU Architecture: Structure, Layers, and Performance Explained* [Online]. [B.r.]. Dostupné také z: `https://www.scalecomputing.com/resources/understanding-gpu-architecture`. [citováno 2025-02-05].
7. CPPREFERENCE.COM. *C++ reference* [Online]. [B.r.]. Dostupné také z: `https://cppreference.com/`. [citováno 2025-01-09].
8. CSBRANCH. *Types of Parallelism: Data vs. Taks Parallelism* [Online]. 2024. Dostupné také z: `https://csbranch.com/index.php/2024/10/26/types-of-parallelism-data-vs-task-parallelism/`. [citováno 2025-12-15].
9. DEEPWIDI.COM. *Image Loading(stb_image)* [Online]. [B.r.]. Dostupné také z: `https://deepwiki.com/nothings/stb/2.1-image-loading-(stb_image)`. [citováno 2025-01-08].

10. DUNN, Ian; WOOD, Zoë. *Chapter 2: The Graphics Pipeline* [Online]. [B.r.]. Dostupné také z: https://graphicscompendium.com/intro/01-graphics-pipeline. [citováno 2025-01-15].

11. GEEKSFORGEEKS. *Features of C* [Online]. 2025. Dostupné také z: https://www.geeksforgeeks.org/c/features-of-c-programming-language/. [citováno 2025-10-12].

12. GEEKSFORGEEKS. *Top 25 C++ Applications inf Real World[2025]* [Online]. 2025. Dostupné také z: https://www.geeksforgeeks.org/blogs/top-applications-of-cpp-in-real-world/. [citováno 2025-10-22].

13. GLFW.ORG. *GLFW Introdruction* [Online]. [B.r.]. Dostupné také z: https://www.glfw.org/docs/latest/index.html. [citováno 2025-01-08].

14. HARBISON, Samuel P.; STEELE, Guy L. *C, a Reference Manual*. 5th. Prentice-Hall, 2002. ISBN 9780130895929.

15. KERNIGHAN, Brian W.; RITCHIE, Dennis M. *The C Programming Language*. 2nd. Pearson, 1988. ISBN 9780131103627.

16. KHRONOS.ORG. *OpenGL Loading Library* [Online]. [B.r.]. Dostupné také z: https://wikis.khronos.org/opengl/OpenGL_Loading_Library. [citováno 2025-01-08].

17. KIRK, David B.; HWU, Wen-mei W. *Programming Massively Parallel Processors: A Hands-on Approach*. 3rd. Morgan Kaufmann, 2016. ISBN 9780128119860.

18. KOCHAN, Stephen G. *Programming in C*. 4th. Pearson Education, 2015. ISBN 9780321776419

19. MORTENSEN, Peter. *What's the difference between a low-level, midlevel and hight-level language* [Online]. 2023. Dostupné také z: https://stackoverflow.com/questions/3468068/whats-the-difference-between-a-low-level-midlevel-and-high-level-language. [citováno 2025-10-13].

20. NOLLE, Tom. *What is structured programming(modular programming)* [Online]. 2023. Dostupné také z: https://www.techtarget.com/searchsoftwarequality/definition/structured-programming-modular-programming. [citováno 2025-10-14].

21. NVIDIA. *Accelerating Fibre Orientation Estimation from Diffusion Weighted Magnetic Resonance Imaging Using GPUs* [Online]. 2013. Dostupné také z: `https://www.researchgate.net/figure/Typical-NVIDIA-GPU-architecture-The-GPU-is-comprised-of-a-set-of-Streaming_fig1_236666656`. [citováno 2025-12-11].

22. OPENGL.ORG. *OpenGL Mathematics(GLM)* [Online]. [B.r.]. Dostupné také z: `https://www.opengl.org/sdk/libs/GLM/`. [citováno 2025-01-08].

23. SRUTHY. *What Is C++ Used For? Top 12 Real-World Applications And Uses Of C++* [Online]. 2025. Dostupné také z: `https://www.softwaretestinghelp.com/cpp-applications/`. [citováno 2025-10-22].

24. STROUSTRUP, Bjarne. *Design and Evolution of C++, The*. 1st. Addison-Wesley Professional, 1994. ISBN 9780201543308.

25. STROUSTRUP, Bjarne. *Programming Principles and Practice Using C++*. 2nd. Pearson Education, 2014. ISBN 9780321992789.

26. STROUSTRUP, Bjarne. *The C++ Programming Language*. 4th. Pearson Education, 2013. ISBN 9780321563842.

27. TECH, WsCube. *Top Applications of C programming* [Online]. 2024. Dostupné také z: `https://www.wscubetech.com/resources/c-programming/applications`. [citováno 2025-10-09].

28. UNIVERSITY, Cornell. *Memory Types* [Online]. [B.r.]. Dostupné také z: `https://cvw.cac.cornell.edu/gpu-architecture/gpu-memory/memory_types`. [citováno 2025-12-13].

29. VOLLE, Adam. *C++ computer language* [Online]. 2025. Dostupné také z: `https://www.britannica.com/technology/C-computer-language`. [citováno 2025-10-17].

30. VRIES, Joey de. *Learn OpenGL Learn modern OpenGL graphics programming in a stepbystep fashion.* 1st. Kendall & Welling, 2020. ISBN 9789090332567.

Zkratky

API Application progamming interface. 14, 19

CPU Central processing unit. 7, 8, 10, 11

DRAM Dynamic random access memory. 8

GLSL OpenGL shading language. 15, 16, 22

GPU Graphics processing unit. 7–12, 14, 15, 21, 28

RAM Random access memory. 7, 8

SM Streaming multiprocessor. 8, 10

SP Streaming processor. 8, 9

VAO Vertex array object. 15, 19

VBO Vertex buffer object. 15, 19

VRAM Video random access memory. 7, 8

Seznam obrázků

Zdrojový kód

Všechen zdrojový kód použitý při tvorbě této práce je v přiloženém zip souboru.