VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

HRA V GLSL

GAME IN GLSL

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **MICHAL ŠLESÁR**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. TOMÁŠ MILET**
SUPERVISOR

BRNO 2022

Abstrakt

Moderné hry v dnešnej dobe vykonávajú značný počet výpočtov. Niektoré z týchto výpočtov sa z dôvodu nedostatočnej výpočetnej rýchlosti CPU, vykonávajú na grafickej karte. Typicky sa jedná o vykresľovanie grafických elementov ako napríklad 2D a 3D objekty, ich textúry a uživateľské rozhranie. Na druhej strane, medzi činnosti ktoré sa vo väčšine prípadov vykonávajú na CPU, patrí napríklad logika hry, reakcia na uživateľský vstup alebo práca so zvukom. Cieľom tejto práce je pokúsiť sa väčšinu týchto činností implementovať na grafickej karte a minimalizovať použitie CPU. Táto práca popisuje nástroj pre tvorbu hier, ktoré vykonávajú celú svoju vnútornú logiku vrátane vykresľovania na grafickej karte. Výstupom tejto práce je hra vytvorená v tomto nástroji, ktorá demonštruje jeho možnosti a funkčnosť.

Abstract

Modern games these days perform a considerable amount of calculations. Some of these calculations are performed on the graphics card due to insufficient CPU speed. Typically, this involves rendering of graphical elements such as 2D and 3D objects, their textures and the user interface. On the other hand, activities that are usually performed on the CPU include, for example, game logic, response to user input, or working with sound. The goal of the thesis is to try to implement most of these activities on the graphics card and minimize CPU usage. This thesis describes a tool for creating games that execute all their internal logic, including rendering on the graphics card. The output of this thesis is a game created in the tool, which demonstrates its capabilities and functionality.

Kľúčové slová

GPU, CPU, grafická karta, herný engine

Keywords

GPU, CPU, graphics card, game engine

Citácia

ŠLESÁR, Michal. *Hra v GLSL*. Brno, 2022. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Tomáš Milet

Hra v GLSL

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána Ing. Tomáša Mileta. Uviedol som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpal informácie pre tvorbu tejto práce.

.....
Michal Šlesár
18. januára 2022

Poďakovanie

Rád by som sa poďakoval vedúcemu práce, Ing. Tomášovi Miletovi, za jeho odbornú pomoc, naprieč celou tvorbou tejto práce.

Obsah

1 Úvod **2**
1.1 Ciele práce 2

2 Teória **3**
2.1 GPU 3
2.1.1 Jadro 3
2.1.2 Program 7
2.1.3 API 8
2.2 OpenGL 9
2.3 Dostupné nástroje pre tvorbu hier 11

Literatúra **12**

Kapitola 1

Úvod

Na pozadí počítačovej hry sa odohráva veľký počet komplexných činností, ktoré častokrát uživateľ, respektíve hráč, nevidí. V závodnej hre by medzi tieto činnosti patrilo napríklad výpočet fyziky áut, prehrávanie zvuku motora vzhľadom od aktuálnej rýchlosti alebo zaradeného stupňa, prehrávanie zvuku okolitého prostredia, aktualizácia dát v uživateľskom rozhraní a interakcia s ním. Zariadenie na ktorom táto hra beží ma ale obmedzený výpočetný výkon, ktorý je potrebné medzi tieto činnosti rozložiť. Činnosť ktorá trvá príliš dlho, ovplyvňuje tie ostatné a prejaviť sa to môže napríklad formou oneskorenia a zasekávania. Za takúto činnosť je možné považovať napríklad vykresľovanie, ktoré vyžaduje značnú časť výpočetného výkonu. Navyše značne narastá pri vysokom počte a zložitosti prvkov, ktoré je potrebné vykresliť. Tento problém začal byť naprieč 90. rokmi natoľko limitujúci, že bolo potrebné presunúť tieto výpočty z CPU na inú komponentu zariadenia, ktorá by ich počítala rýchlejšie a efektívnejšie.

Základ počítačov, CPU, má poskytovať flexibilnú a všestrannú funkcionalitu pre implementáciu akéhokoľvek algoritmu. Výpočetný výkon CPU je ale na niektoré algoritmy a činnosti príliš pomalý. Grafické karty, ďalej len GPU, boli originálne vytvorené za účelom zrýchlenia vykresľovania 3D objektov a práce s textúrami. Ich paralelná štruktúra zaručuje vysokú efektivitu špecifických algoritmov, oproti CPU niekoľkonásobne, ktoré potrebujú pracovať paralelne s veľkým objemom dát. Naprieč rokmi sa GPU stali flexibilnejšími, využívajú sa v rôznych technologických sektoroch a ich použitie už nieje cielené iba na hry.

CPU sa dá predstaviť ako švajčiarsky nôž a GPU ako mačeta. Švajčiarsky nôž je nápomocný pri množstve rôznych úloh, od prestrihnutia lana až po otvorenie plechovky s fazuľou. Keď sa však chceme predierať hustou džungľou, využijeme silnú mačetu, nie malý švajčiarsky nôž.

1.1 Ciele práce

Hlavným cieľom tejto práce je vytvoriť nástroj pre tvorbu hier takých, že celá vrátane vykresľovania a vnútornej logiky pobeží na GPU. Práca má taktiež za úlohu nájsť limity a prednosti tohto spôsobu tvorby hier a porovnať ho z klasickým spôsobom, kde sa stále prevažne využíva CPU na väčšinu výpočtov logiky hry.

Kapitola 2

Teória

Táto časť popisuje všetký podstatné informácie pre tému tejto práce, teda popis GPU, porovnanie GPU s CPU, spôsob práce s GPU a dostupné nástroje pre tvorbu hier.

2.1 GPU

GPU má radu využití, obzvlášť vykresľovanie, ale aj napríklad strojové učenie alebo ťaženie kryptomien. Ak vynecháme použitia GPU, ktoré sa rozšírili hlavne poslednými rokmi a v tejto práci nie sú podstatné, môžeme GPU definovať ako "Výpočetnú jednotku zariadenia pre akceleráciu komplexných matematických a geometrických výpočtov, ktoré sú nutné pre grafické vykresľovanie.".

2.1.1 Jadro

GPU podobne ako aj CPU sa skladá z jadier.

- **Jadro** je samostatná výpočetná jednotka, ktorá prijíma inštrukcie pracujúce s nejakými dátami a vykonáva výpočty alebo operácie pre uspokojenie týchto inštrukcií.

Jadro CPU

CPU jadro je schopné širokého spektra operácií a vysporiadava sa s tokom inštrukcií. Viaceré programy súperia o výpočetný čas, ktorý im je prideľovaný operačným systémom. Inštrukcie pracujú s dátami uloženými v pamäti RAM. Tento prenos je relatívne pomalý nakoľko pamäť RAM nieje súčasťou CPU. Z toho dôvodu si CPU uchováva niektoré dáta vo vyrovnávacej pamäti, takzvanej CPU cache, za účelom minimalizácie prístupu k pamäti RAM.

Spracovanie veľkého počtu inštrukcií benefituje z vysokej rýchlosti jednotlivého jadra, pretože môže byť obtiažne rozdeliť jeden tok inštrukcií na niekoľko menších. Pre toto spracovanie je teda vhodnejší menší počet výkonných jadier.

Jadro GPU

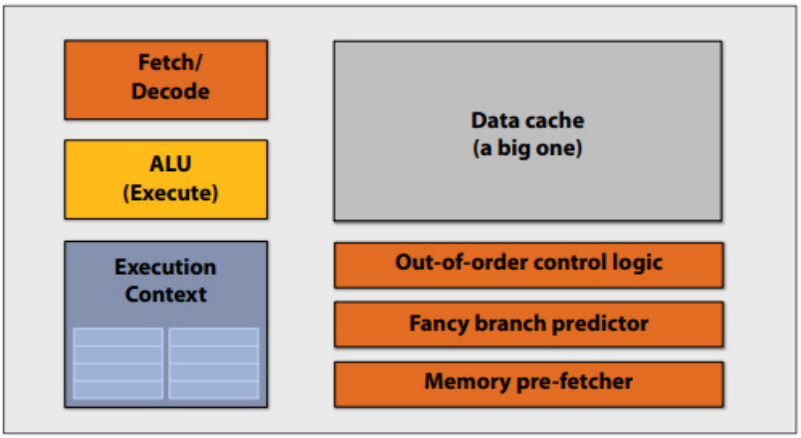
GPU jadro je na druhej strane navrhnuté pre prácu s datovými tokmi. Jednotlivé jadra pracujú s minimálnym počtom inštrukcií, ktoré pracujú s potencionálne veľkým počtom dát. Grafické karty majú vlastnú pamäť, zvanú video pamäť, v ktorej si dáta s ktorými pracujú ukladajú. Vzhľadom na to, že táto pamäť je súčasťou grafickej karty, je rýchlosť prenosu

medzi ňou a jadrami niekoľkonásobne vyššia ako v prípade CPU komunikujúcim s pamäťou RAM. Samotné jadrá v moderných GPU zdieľajú tok inštrukcií za účelom minimalizácie réžie jednotlivých jadier.

Spracovanie veľkého množstva dát častokrát znamená vykonávanie rovnakej činnosti nad rôznymi dátami, častokrát definovanej malým počtom inštrukcií. Pre takéto spracovanie je teda výhodný paralelizmus a vysoký počet jadier. Pre príklad môžeme uviesť vykresľovanie objektov, kde sa vykonáva ten istý algoritmus nad potencionálne obrovským počtom objektov.

Časti jadra

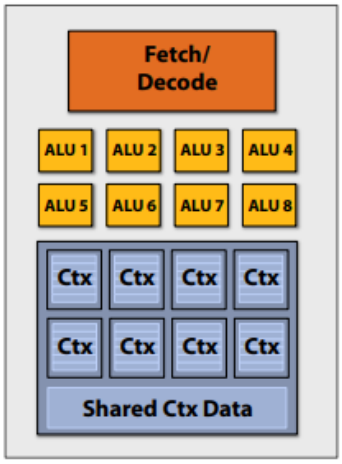
V čom spočíva efektivita GPU jadier oproti CPU jadrám? Na to aby bolo možné túto otázku zodpovedať, je potrebné začať rozložením častí CPU jadra.



Obr. 2.1: Rozloženie častí CPU jadra.

Jadro CPU funguje na princípe fetch/decode/execute. Teda načíta inštrukciu - fetch, dekóduje ju - decode a následne vykoná - execute. Stav procesoru, teda obsahy registrov a programový čítač je uložený v execution kontexte. Data cache obsahuje lineárne časti pamäte RAM, ktoré si CPU dočasne uchováva pomocou prefetcheru za účelom minimalizácie k jej prístupu, keďže ako už bolo spomenuté, je to relatívne pomalé a CPU potrebuje inštrukcie odbavovať čo najrýchlejšie. Pre zvýšenie efektivity CPU taktiež využíva predikciu vetvenia, kde sa snaží CPU uhádnuť smer vetvy, predtým ako je to definitívne známe. Posledná časť je vykonávanie inštrukcií v inom ako ich definovanom poradí. Pokiaľ procesor čaká na dokončenie nejakej inštrukcie, je možné medzitým vykonať následujúce inštrukcie, ktoré na nej nezávisia. To závisí od dát a jednotiek s ktorými inštrukcia pracuje.

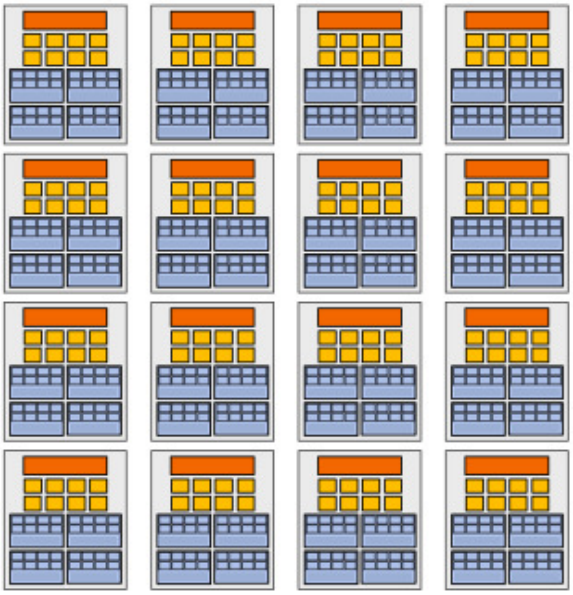
Funkcionalita ktorú obsahujú CPU jadrá prináša so sebou navýšenie réžie. Keďže GPU dnes dosahujú tisíce jadier, je táto réžia neprípustná. Ako už bolo povedané v popise GPU jadra, viz. 2.1.1, je účelom jadier paralelne vykonávať rovnakú činnosť. Nieje teda nutné aby každé GPU jadro obsahovalo podobné časti a funkcionalitu ako CPU jadro.



Obr. 2.2: Rozloženie častí GPU jadra.

Jadro GPU funguje taktiež na princípe fetch/decode/execute. Jedným z rozdielov medzi GPU a CPU je ten, že GPU dostane úlohu, respektíve program, a čaká sa na jej výsledok. Programov na CPU beží veľa naraz, niektoré nikdy nekončia, niektoré vykonávajú akcie počas ktorých môžu byť odložené a spustené neskôr. GPU jadro vďaka tomu nepotrebuje tak pokročilé mechanizmy na zefektívnenie toku inštrukcií. Samotné GPU jadro teda načíta inštrukciu ktorú má vykonať. Obsahuje viacero aritmetických logických jednotiek - ALU a bloky kontextov, ktoré tieto ALU využívajú. Tento princíp sa nazýva SIMD - Single Instruction Multiple Data. Kontexty majú vlastné dáta ale aj zdieľané medzi sebou.

- Príklad využita viacerých kontextov a výpočetných jednotiek môže byť napríklad výpočet 8 pixelov na obrazovke. Jadro si načíta inštrukciu pre výpočet farby pixelu. Každý z týchto 8 pixelov má vlastný context, ktorý obsahuje napríklad jeho pozíciu. Jednotlivé kontexty jadro spracuje paralelne pomocou dostupných ALU jednotiek. Počet jednotiek ALU a kontextov v jednotlivých jadrách závisí od samotnej GPU.



Obr. 2.3: Viacero jadier GPU.

Na obrázku 2.3 je možné vidiet predstavu viacero GPU jadier, kde môžeme vidieť 16 jadier. Každé jadro obsahuje 4 bloky kontextov, kde každý blok obsahuje 8 kontextov. V prípade aplikácie príkladu o pixeloch 2.1.1 na tento obrázok, by sme mohli paralelene vypočítať farbu 512 pixelov.

$$16 * 4 * 8 = 512 \tag{2.1}$$

Logická organizácia práce

Pri práci s GPU nepracujeme priamo s jadrami. Využíva sa pojem globálnych a lokálnych pracovných skupín, do ktorých je rozdelená požadovaná práca vo forme pracovných jednotiek. Pracovná jednotka je najmenšia jednotka zapúzdrujúca operácie, ktorú môže uživateľ vykonať. Pracovné skupiny sú definované v 3D priestore. Akákoľvek z osí priestoru môže byť 1, čo je vhodné pre spracovanie akýchkoľvek obrazových dát alebo lineárnych dát. Pracovné skupiny sú vykonávané paralelne, nezávisle od iných.

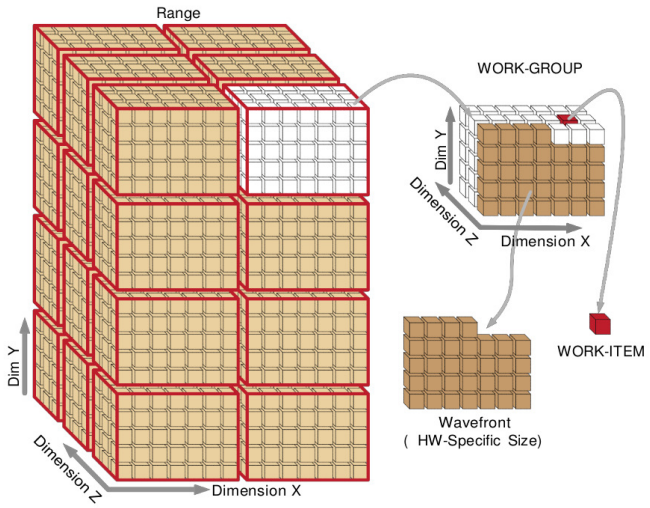


Figure 2.3 Work-Item Grouping Into Work-Groups and Wavefronts

Obr. 2.4: Organizácia pracovných skupín.

- **Globálna skupina** je pracovná skupina, ktorej jednotky predstavujú lokálne skupiny. Globálne skupiny sú vykonávané nezávisle na sebe.
- **Lokálna skupina** je pracovná skupina, ktorej jednotky predstavujú spustenie programu, ktoré využívajú svoju pozíciu v priestore a častokrát pracujú s jemne odlišnými dátami. Lokálne skupiny sú vykonávané nezávisle na sebe, ale samotné jednotky lokálnej skupiny je možné medzi sebou synchronizovať.

V prípade vykonania pracovných skupín na základe obrázku 2.4, teda globálna skupina veľkosti (X=2, Y=3, Z=3) a lokálna skupina veľkosti (X=7, Y=5, Z=3), by sme dostali 1890 jednotlivých spustení programu, kde v každom spustení môžeme pracovať s inými dátami.

$$(2 * 3 * 3) * (7 * 5 * 3) = 1890 \tag{2.2}$$

Veľkosť pracovných skupín sa prispôsobuje požiadavkam programu, ktorý bude spúšťaný na GPU. Je možné spraviť aj 1 globálnu skupinu (X=1, Y=1, Z=1) s 1 lokálnou skupinou (X=1, Y=1, Z=1), ktorá by vykonala program iba raz.

Kedy použiť GPU a kedy CPU

Na teoretickej úrovni je častokrát možné dosiahnuť toho istého cieľa na CPU aj GPU, no spôsob jeho dosiahnutia by sa líšil vzhľadom na ich odlišný princíp. Na základe spôsobu akým fungujú, je GPU ideálne pre všeobecne zamerané toky inštrukcií, a GPU je ideálne pre prácu s veľkým objemom dát.

2.1.2 Program

- **Program** je sekvencia alebo zoznam inštrukcií v nejakom programovacom jazyku, ktoré môže počítač interpretovať alebo vykonať.

Programy na CPU sú definované inštrukciami, ktoré samotné CPU podporuje a ich forma sa naprieč rokmi podstatne nemení. Tieto inštrukcie sú zapísané v takzvanom assembly code, ktorý je určený pre priamu komunikáciu s hardwarom. Rôzne kompilované programovacie jazyky sú kompilované do assembly code, ktorý je následne skompilovaný do strojového kódu spustiteľného na CPU na danom operačnom systéme.

Programy na GPU sú taktiež definované inštrukciami, ale forma týchto inštrukcií a spôsob ich použitia sa častokrát mení v každej GPU pre zaručenie najvyššej efektivity. Z toho dôvodu nieje väčšinou inštrukčná sada GPU verejne dostupná a komunikácia prebieha inak. GPU potrebuje ovládač, ktorý implementuje rôzne verejne popísané API, pomocou ktorých je možné s GPU pracovať. Následne je možné v programoch bežiacich na CPU volať funkcie popísané týmito API. Jednou z funkcií týchto API je nahranie programu, respektíve jeho zdrojového kódu na GPU, kde je následne skompilovaný do optimalizovaného strojového kódu, ktorý je možné na GPU spustiť.

Programy GPU si typicky vopred definujú prostriedky s ktorými chcú na GPU pracovať a ako sa majú na GPU logicky rozložiť 2.1.1

2.1.3 API

- **Application Programming Interface - API** je zoznam funkcií alebo procedúr umožňujúcich vytváranie aplikácií, ktoré pristupujú k funkciám alebo dátam operačného systému, aplikácie alebo inej služby.

Komunikácia s GPU prebieha prostredníctvom programu spúšťaného na CPU, ktorý volá funkcie popísané API implementovanou ovládačom GPU. Takýchto API existuje v dnešnej dobe viac a každá má svoje vlastné využitie. Medzi najznámejšie a najviac používané patrí

- **OpenCL** - API pre tvorbu programov všeobecného účelu, spustiteľných na viacerých zariadeniach ako napríklad CPU, GPU alebo FPGA.
- **CUDA** - API podobne ako OpenCL využívaná pre tvorbu programov všeobecného účelu, spustiteľných na GPU. Je to proprietárna technológia pre GPU vyrobené firmou NVIDIA.
- **OpenGL** - API využívaná najmä pre vykresľovanie 3D grafiky. Je podporovaná takmer všetkými modernými GPU a niektoré jej implementácie majú verejne dostupný zdrojový kód. Jej prvotné verzie boli pomerne obmedzené a poskytovali obmedzenú funkcionalitu schovanú za pomerne vysokou abstrakciou. Naprieč rokmi bola táto funkcionalita rozdelená za účelom zníženia abstrakcie a teda poskytnutia vyššej flexibility vývojárom. Programy popisuje vlastným programovacím jazykom GLSL, ktorého syntax je založená na jazyku C.
- **Vulkan** - API novej generácie, ktorá by mala nahradiť OpenGL. Obsahuje minimum abstrakcií oproti OpenGL a teda jej réžia je minimálna. Programy popisuje formátom SPIR-V, ktorý môže pripomínať obdobu assembly kódu pri CPU. Je teda možné využiť viaceré programovacie jazyky, ktoré sú následne skompilované do SPIR-V, teda aj GLSL.
- **DirectX** - API využívaná podobne ako OpenGL a Vulkan pre vykresľovanie 3D grafiky. Táto API je proprietárna pre systém Windows a hernú konzolu Xbox a jej

implemntácia nieje verejne dostupná. Zariadenia so systémom Windows podporujú aj ostatné populárne API. Jej staršie verzie fungovali na podobnom princípe ako OpenGL, kdežto najnovšia verzia tejto API, DirectX 12, funguje na podobnom princípe ako Vulkan. Programy popisuje programovacím jazykom HLSL, ktorého syntax je, podobne ako ostatné API, založená na jazyku C.

- **Metal** - Proprietárna API pre zariadenia Apple pre prácu s GPU. Najnovšie zariadenia Apple nepodporujú Vulkan a novšie verzie OpenGL, a teda vývojárovi na tejto platforme nedáva inú možnosť ako pracovať s touto API. Existujú nástroje ktoré tento problém riešia, napríklad MoltenVK, ktorého úlohou je preklad volania funkcionality API vulkan na ekvivalentné volania API Metal.

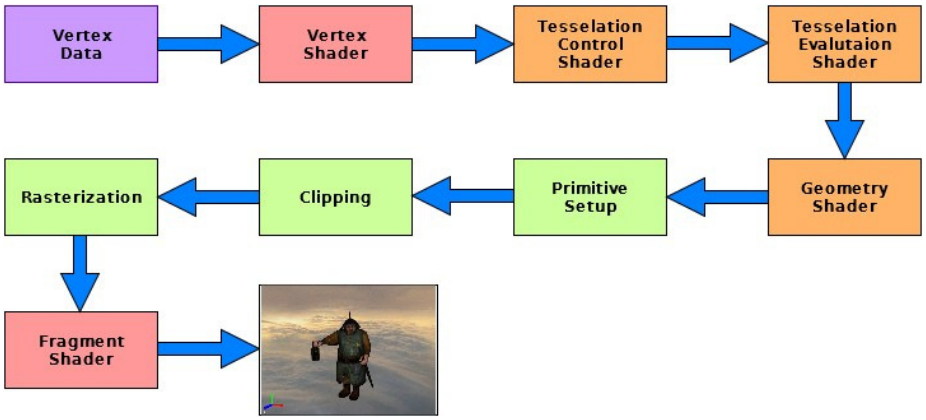
Vzhľadom na to že táto práca je zameraná primárne na grafické vykresľovanie, je pozornosť venovaná primárne API OpenGL, kvôli relatívne jednoduchému a jasnému použitiu.

2.2 OpenGL

OpenGL je voľne dostupná a multi platformová API pre prácu s 2D a 3D grafikou, vyvinutá organizáciou Khronos. Je definovaná ako zoznam funkcií a konštánt, ktoré môže klientsky program bežiaci na CPU využiť, respektíve zavolať. Ponúka viaceré abstrakcie pre uľahčenie práce s GPU a grafikou všeobecne.

Shader

V OpenGL sa spúšťaný program skladá z viacerých menších programov zvaných shader. Tento shader je na GPU nahraný v textovej podobe, v jazyku GLSL. Tá je následne skompilovaná do formy ktorú je GPU schopné spustiť. Existuje viacero typov shaderov, kde každý shader má vlastný účel a vopred definované vstupy a výstupy. Shadre v programe majú určené poradie a sú súčasťou grafickej pipeline, ktorá definuje poradie jednotlivých častí vykresľovania.



Obr. 2.5: Grafická pipeline.

- **Vertex data** predstavuje vstup grafickej pipeline. Je to zoznam vrcholov primitívnych objektov, typicky trojuholníkov.

- **Vertex Shader** je prvý shader grafickej pipeline, ktorý má za úlohu spracovať vrcholy primitívneho objektu. V tomto shaderi je napríklad možné upraviť pozície vrcholov tak, aby odpovedali pozícii kamery scény.
- **Tesselation Shader** je shader, pomocou ktorého je možné rozdeliť primitívny objekt na niekoľko menších, za účelom manipulácie s detailom objektu. Vďaka tomuto shaderu je možné napríklad z kocky, ktorá je definovaná 8 vrcholmi, spraviť hladkú guľu, ktorá by musela mať oveľa viac vrcholov.
- **Geometry Shader** je shader, ktorý má na vstupe zoznam vrcholov ktoré formujú primitívny objekt, napríklad bod alebo trojuholník. Tie môže následne ľubovoľne transformovať a poslať ďalej. Shader teda môže prijať napríklad jeden bod, a vytvoriť z neho trojuholník so stredom v tomto bode.
- **Clipping** je interná záležitosť grafickej karty, ktorá na základe nejakých podmienok zahodí primitívne objekty. Typicky sa jedná o objekty ktoré sa nachádzajú mimo obrazovky a teda ich nieje potrebné vykresľovať.
- **Rasterizácia** je predposledná časť pipeline. Prevádza primitívne objekty definované vrcholmi na rastrový obrázok, teda sériu pixelov, zvaných fragmenty.
- **Fragment Shader** je shader, ktorý má na vstupe fragmenty vytvorené v rasterizácii. Je možné s nimi manipulovať, napríklad zmeniť ich farbu. Výstupom fragment shaderu je výsledný obraz.

Program v OpenGL ktorý má za úlohu vykresliť obraz môže byť teda definovaný 4 shadermi. OpenGL má ale ešte jeden dôležitý typ shaderu, ktorý je podstatný pre prácu s niečim iným ako vykresľovaním. Jedná sa o Compute Shader, ktorý sa používa na výpočet ľubovoľných informácií. Computer shader môže aj vykresľovať, ale vo väčšine prípadov sa používa pre úlohy, ktoré s vykresľovaním priamo nesúvisia. Compute shader nieje možné kombinovať s inými typmi shadrov vrámci jedného programu.

Textúra

Textúra je typicky 2D obraz, pomocou ktorého nanášame na časti objektu detaily. Pomocou OpenGL je možné textúru manuálne vytvoriť a nahrať do nej dáta. V jednotlivých shaderoch je následne možné s textúrami pracovať.

Buffer

Buffer je vyhradená časť video pamäte GPU s prideleným identifikátorom, pomocou ktorého môžeme s týmto blokom pamäte pracovať. Dáta uložené v bufferoch je možné modifikovať, čítať a meniť ich veľkosť. Buffery je možné naviazať na konkrétny typ. Jednotlivé shadre a interné funkcie OpenGL očakávajú dáta v konkrétnych typoch bufferov.

Framebuffer

Fragment shader sám o sebe nevykresľuje svoj výstup priamo na obrazovku. Výstup je uložený do bloku pamäte, zvaného framebuffer. V OpenGL existuje predvolený framebuffer, do ktorého sa ukladá výstup určený pre obrazovku. Je možné vytvoriť dodatočné framebuffery, ktoré sa mapujú na textúry, s ktorými ďalej možeme pracovať.

Uniformná premenná

Uniformná premenná je globálna premenná shaderu. Predstavuje parameter, ktorý môže uživateľ shaderu odoslať. Shader môže tento parameter využiť ľubovoľne podľa potreby.

SSBO

SSBO, alebo Shader Storage Buffer Object, je taktiež premenná shaderu, ktorá sa chová podobne ako uniformná premenná. Rozdiel spočíva v tom, že veľkosť SSBO môže byť oveľa vyššia a je možné do nich zo shaderu zapisovať, nie len z nich čítať. Je možné teda napríklad využiť program s Compute Shaderom, ktorý svoj výpočet uloží do SSBO a výsledky sa následne použijú v inom programe.

2.3 Dostupné nástroje pre tvorbu hier

Literatúra