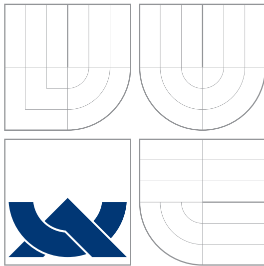
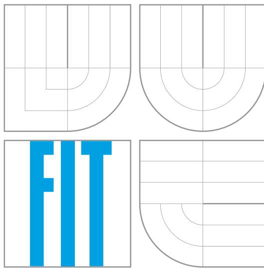


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

POROVNÁNÍ VÝKONU VYKRESLOVÁNÍ V IOS A METAL

RENDERING PERFORMANCE COMPARISON IN IOS AND METAL

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — RADEK PIŠTĚLÁK
AUTHOR

VEDOUCÍ PRÁCE — Ing. MICHAL TÓTH
SUPERVISOR

BRNO 2016

Zadání bakalářské práce/18921/2015/xpiste04

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2015/2016

Zadání bakalářské práce

Řešitel: **Pištělák Radek**
Obor: Informační technologie
Téma: **Porovnání výkonu vykreslování v iOS a Metal**
Rendering Performance Comparison in iOS and Metal
Kategorie: Počítačová grafika

Pokyny:

1. Seznamte se s knihovnou Metal na platformě iOS, zejména jejími rozšířeními a možnostmi implementace v nativním kódu. Popište rozdíly mezi Metal a OpenGL ES, případně Vulcan.
2. Prostudujte a popište vhodný algoritmus výpočtu osvětlení, jež by demonstroval výhody Metal.
3. Navrhněte jednoduchou aplikaci pro měření výkonu vykreslování na mobilních zařízeních.
4. Navrženou aplikaci implementujte, zhodnoťte dosažený výkon na různých zařízeních.
5. Zhodnoťte dosažené výsledky a diskutujte možnosti budoucího vývoje.
6. Vytvořte video prezentující výsledky vaší práce.

Literatura:

- Dle pokynů vedoucího

Pro udělení zápočtu za první semestr je požadováno:

- Bez požadavků.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

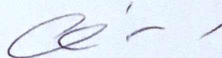
Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Tóth Michal, Ing.**, UPGM FIT VUT
Konzultant: Polok Lukáš, Ing., UPGM FIT VUT
Datum zadání: 1. listopadu 2015
Datum odevzdání: 18. května 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2

L.S.



doc. Dr. Ing. Jan Černocký
vedoucí ústavu

Abstrakt
Cílem této práce je porovnání grafických knihoven OpenGL ES a Metal na mobilním systému iOS. Pro porovnání výkonu obou knihoven slouží implementovaný částicový systém. Měřením bylo zjištěno, že při použití instanced renderingu lze v současné době dosáhnout vyššího výkonu pomocí knihovny OpenGL ES. Na základě zjištěných výsledků a teoretické části práce je možné provést informovanější rozhodnutí mezi oběma grafickými knihovnami.

Abstract
The goal of this thesis is the comparison of the OpenGL ES and Metal graphical libraries on the iOS platform. To compare the performance of the two libraries, a particle system has been implemented. By profiling it has been determined that using instanced rendering it is currently possible to achieve higher performance using OpenGL ES. Based on the results and the theoretical part of the thesis it is possible to make a more informed decision between the two graphical libraries.

Klíčová slova
TBDR, IMR, iOS, Metal, OpenGL ES, částicové systémy

Keywords
TBDR, IMR, iOS, Metal, OpenGL ES, particle systems

Citace
PIŠTĚLÁK, Radek. *Porovnání výkonu vykreslování v iOS a Metal.* Brno, 2016. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Tóth Michal.

Porovnání výkonu vykreslování v iOS a Metal

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana inženýra Michala Tótha. Další informace mi poskytl Ing. Lukáš Polok. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

....
Radek Pištělák
18. května 2016

Poděkování

Zde bych rád poděkoval Ing. Michalu Tóthovi za vedení práce a také Ing. Lukáši Polokovi za jeho odbornou pomoc.

© Radek Pištělák, 2016.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Architektura mobilních grafických čipů **3**
- 1.1 IMR . . . 3
- 1.2 TBDR . . . 4
- 1.3 Srovnání IMR a TBDR . . . 5

2 OpenGL ES **6**
- 2.1 Architektura . . . 6
- 2.2 Základní pojmy . . . 6
- 2.3 OpengGL a TBDR . . . 7
- 2.4 Problémy OpenGL . . . 8
- 2.5 Shadery . . . 9

3 Nízkoúrovňové knihovny **10**
- 3.1 Metal . . . 10
  - 3.1.1 Rozhraní knihovny Metal . . . 11
  - 3.1.2 Generování příkazů . . . 11
  - 3.1.3 Správa prostředků . . . 12
  - 3.1.4 Metal a TBDR . . . 13
  - 3.1.5 Shadery . . . 14
- 3.2 Vulkan . . . 14

4 Měření výkonu vykreslování **16**

5 Experimenty **17**
- 5.1 Ověření vykreslování po dlaždicích v OpenGL kontextu . . . 17
- 5.2 Částicové systémy . . . 18

6 Implementace **19**
- 6.1 Vývoj aplikací pro systém iOS . . . 19
- 6.2 GLKit . . . 21
- 6.3 Srovnání rozhraní OpenGL ES a Metal . . . 21

7 Vyhodnocení výsledků **25**

8 Závěr **29**

Literatura **30**

Úvod

100, 33, 23, 1.5 a 1.

Co tahle čísla znamenají? V červnu roku 2015 společnost Apple Inc.[1] oznámila, že v období 2008 – 2015 bylo z AppStore staženo 100 miliard aplikací, za které Apple vyplatil vývojářům 33 miliard dolarů. Také lze dohledat, že 23% z 1.5 milionu aktivních aplikací tvoří hry. Z nichž 2 nejúspěšnější[2] byly schopny každá generovat denní příjem přes 1 milion dolarů. Z těchto čísel lze vyvodit, že se jedná o perspektivní trh. Vždyť mobilní zařízení se systémem iOS má v rukou již více než půl miliardy lidí.[3] Je, ale potřeba přiznat, že konkurence je velká. Na AppStore je k dispozici více než 400 000 tisíc různých her. Aby byla hra úspěšná musí uživateli nabídnout jak zajímavou myšlenku, tak i kvalitní zpracování. Metal slibuje vývojářům dosáhnutí propracovanějšího herního světa. Naskýtá se, ale otázka: *"Vyplatí se investovat čas, tedy peníze, pro adopci této knihovny v době, kdy je de facto průmyslovým standardem knihovna OpenGL ES?"*. A zodpovědění této otázky je cílem této práce.

Do roku 2014 byla na systému iOS jedinou možností knihovna OpenGL ES. V průběhu téhož roku ovšem společnost Apple představila vlastní řešení s názvem Metal. Tato knihovna ihned začala v rámci systému nahrazovat OpenGL ES. Mezi hlavní výhody má patřit snížení režijních nákladů spojených s vykreslováním. Toho se Metal snaží docílit pomocí validace stavu vykreslovacího řetězce v čase inicializace. Dále také obeznámenost s architekturou současných mobilních grafických čipů.

Architekturou současných mobilních grafických čipů se zabývá první kapitola. V následující kapitole najde čtenář základní informace o knihovně OpenGL. Nicméně se nejedná o ucelený úvod do problematiky, jako spíše o výběr oblastí, které se v dnešní době jeví jako problémové. Na tyto "problémové" oblasti se pak zaměřuje i třetí kapitola z pohlednu moderních nízkoúrovňových API. Poznatky ohledně měření jsou obsaženy v kapitole čtyři. Pátá kapitola cílí na popis navržené testovací sady, která je implementována. Popis implementace je shrnut v kapitole šest. Sedmá a osmá kapitola se zabývá vyhodnocením výsledků experimentů, respektive celé práce.

[1] Celý název spolenočnosti Apple Inc. bude po zbytek textu zkracován pouze na Apple, případně společnost Apple.

[2] Data za prosinec 2015.

[3] Veškeré numerické údaje obsažené v tomto odsatvci pocházejí ze serveru http://www.statista.com.

Kapitola 1

Architektura mobilních grafických čipů

Architektura mobilních grafických čipů se od těch, které známe ze stolních počítačů v mnohém liší. V první řadě převážná většina mobilních grafických čipů tvoří společně s CPU jeden integravaný obvod. V tomto obvodu grafický čip také sdílí paměťové rozhraní společně s ostatními komponenty. Sdílené paměťové rozhraní (také známo jako sjednocená paměťová architektura z angl. *unified memory architecture (UMA)*) společně s GPU a CPU na jednom integrovaném obvodu umožnuje hardwarovým vývojářům dosáhnout menší velikosti, nižší spotřeby a dalších požadavků, které jsou pro mobilní zařízení napajené bateriemi nezbytné. Tato architektura se nazývá *System on Chip (SOC)*.

Literatura [15] rozděluje moderní grafické čipy do tří kategorií podle způsobu vykreslování na čipy s okamžitým vykreslováním (*immidiate mode renderer (IMR)*), čipy využívající dlaždic (*tile based renderer (TBR)*) a čipy využívající dlaždice s odloženým vykreslováním (*tile based deferred renderer (TBDR)*).

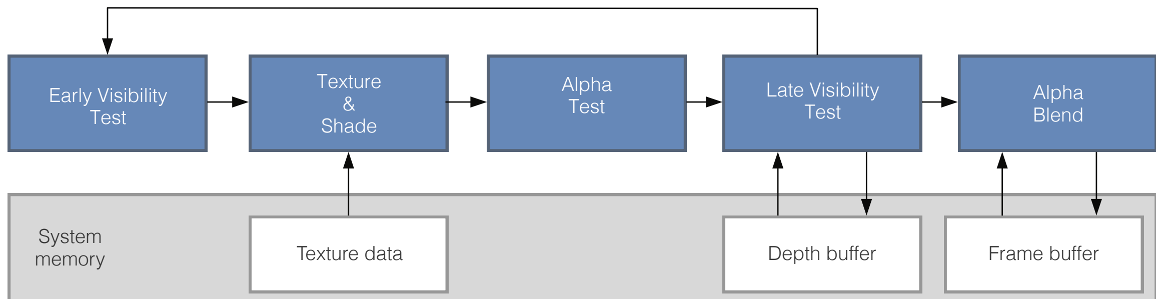
Další část této kapitoly se zaměřuje na dvě z těchto tří kategorií. A to na *IMR*, jako na tradiční architekturu, pro kterou bylo navrženo OpenGL, kterému se věnuje následující kapitola 2 a na *TBDR*. Čipy s architekturou *TBDR* pohánějí mobilní produktovou řadu společnosti Apple a primárně na ně je cílena knihovna Metal 3.1.

1.1 IMR

Při okamžitém režimu vykreslování prochází veškerá geometrie celým vykreslovacím řetězcem (obrázek 1.1) v pořadí, ve kterém je odeslána aplikací ke zpracování. Nevýhodou této architektury jsou vysoké požadavky na paměťovou propustnost. Vysoké nároky na paměťovou prospustnost nemusí být problém u stolních počítačů, kde jde především o co nejvyšší výkon, ale jak bylo uvedeno v úvodu kapitoly u mobilních zařízení musí být brán ohled také na spotřebu. Literatura [8] uvádí, že jsou poměrně běžné situace, kdy pro zobrazení jednoho pixelu (~ 4 bajty informací) není problém vygenerovat přes 100 bajtů paměťového provozu. A to všechno za předpokladu, že nebude docházet např. k překreslování z angl. *overdraw*. K překreslování dochází v případě, že později odeslaný fragment překrývá pixel, který už je zapsaný ve *framebufferu*, a který tedy byl "obarvován" zbytečně, protože se nebude podílet na výsledném obraze.

Moderní IMR architektury se snaží redukovat překreslování pomocí techniky *Early-Z* testování, při kterém provádí *depth test* ještě před "obarvováním" pixelu. Nicméně výhody

této techniky lze plně využít pouze pokud jsou veškerá primitiva před odesláním seřazena od těch nejbližších po nejvzdálenější [15]. Nevýhodou ovšem stále zůstává fakt, že barva a hloubka se v průběhu průchodu vykreslovacím řetězcem ukládá do systémové paměti, což způsobuje nemalé nároky na paměťovou propustnost [17].



Obrázek 1.1: Část vykreslovacího řetězce IMR. Původní obrázek lze najít v [15].

Obrázek (1.1) ukazuje část vykreslovacího řetězce čipu s okamžitým režimem vykreslování. Lze vidět častou komunikaci se systémovou pamětí.

1.2 TBDR

V článku [14] autor popisuje, že za vznikem *TBDR* architektury stojí snaha o dosáhnutí větší efektivity, především v oblasti práce s pamětí (viz. překreslení a předešlá podkapitola o *IMR* 1.1).

Vykreslování pomocí *TBDR* by se dalo rozdělit do dvou fází. Během první fáze je veškerá geometrie, která se podílí na výsledném snímku rozdělena do dlaždic. Odtud pochází část názvu “*tile-based*”, tedy volně přeloženo jako vykreslování “založené na dlaždicích”. Druhá fáze znamená odložení vykreslování (z angl. *deferred rendering*).

Tile-based rendering

Za první částí názvu této architektury (*tile based*) se skrývá fakt, že veškerá geometrie, která se podílí na výsledné scéně je ve chvíli, kdy je známa její pozice na výsledné scéně rozdělena na dlaždice (*tilling* viz obrázek 5.1). Výstupem této části je struktura *parameter buffer (PB)*[1], která uchovává informace o dlaždici a seznam všech primitiv, které se k ní vážou.

Použití dlaždic přináší výhodu v podobě možnosti použít rychlou paměť na čipu a vyhnout se, tak operacím s pomalejší systémovou pamětí.

Deferred rendering

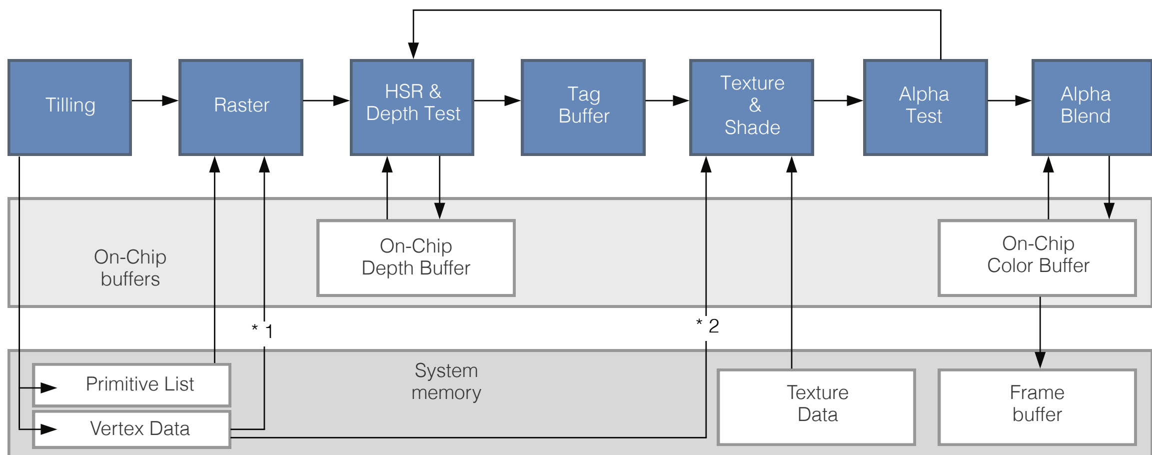
Druhá část názvu *deferred rendering* v tomto kontextu znamená odložení “obarvování” pixelu až na dobu, kdy jsou odstraněny všechny objekty (nebo jejich části), které nejsou viditelné z pohledu kamery.

[1]Název této struktury se liší v závislosti na literatuře/výrobci, lze se setkat i s názvy jako *frame data* nebo *polygon list*. Nicméně Imagination Technologies ve své dokumentaci používá termín *parametr buffer* a jelikož její čipy používá společnost Apple ve svých zařízeních, tak bude dále uváděn tento název.

Proces, který je za toto zodpovědný je na obrázku 5.1 označen jako *HSR* (nebo-li *Hidden Surface Removal*) a probíhá již kompletně v rychlé paměti čipu. V průběhu *HSR* se pro všechny fragmenty na dlaždici otestuje jejich hloubka oproti *depth buffer*-u uloženému v paměti na čipu, který je případně aktualizován. Když jsou všechny trojúhelníky takto zpracovány, tak jsou do dalšího stupně, kterým je vlastní “obarvování” pixelu, posílány nikoli po “řádcích”, ale ve skupinách, které patří k danému primitivu. Tím lze podle [2] dosáhnout efektivnějšího zpracování a přístupu do cache paměti.

Celý tento proces ve výsledku dokonce znamená, že není nutné řadit veškerou geometrii, jako je tomu v případě *IMR*, nebo *TBR* architektur. Jedinou vyjímkou jsou průhledné objekty.

Více se lze o této architektuře dočíst ze [15], [14], nebo [13].



Obrázek 1.2: Část vykreslovacího řetězce TBDR čipu. Původní obrázek lze najít v [15].

Na obrázku 5.1 lze oproti vykreslovacímu řetězci IMR čipu vidět, že od fáze rasterizace se již kromě případného načítání textur obejde čip bez systémové paměti a místo ní používá rychlou vyrovnávací paměť.

1.3 Srovnání IMR a TBDR

Autor [8] zmiňuje fakt, že u architektury *TBDR* je nutné snímek rozložit na dlaždnice a tato operace má určité nároky na systémové prostředky. Zatímco *IMR* může s veškerou geometrií okamžitě pracovat. Tento fakt nepopírají v publikaci [14]ani tvůrci *TBDR* . Nicméně uvádějí, že díky nižším nárokům na paměťovou propustnost můžou stále dosahovat lepších výsledků v typických scénach a hlavně v těch komplexních.

Porovnáním výkonu jednotlivých architektur se zabývá také práce [17]. Výsledným zjištěním je, že architektura *TBDR* vykazuje (na testovaných zařízeních) se zvyšujícím počtem *draw call* výrazně lepších výsledků oproti jiným architekturám.

[2]Přířučka pro vývojáře k PowerVR čipům páté série http://cdn.imgtec.com/sdk-documentation/PowerVR+Series5.Architecture+Guide+for+Developers.pdf.

Kapitola 2

OpenGL ES

Literatura [11] definuje OpenGL ES (*Open Graphics Library for Embedded Systems*) jako aplikační programové rozhraní pro 3D grafické vykreslování cílené na přenosné a vestavěné zařízení.

Kořeny OpenGL ES sahají až do roku 1982, kdy profesor Jim Clark založil na univerzitě ve Stanfordu jednu z prvních společností zabývající se počítačovou grafikou. Společnost se jmenovala Silicon Graphics Computer System, ale později byla známá spíše pod názvem SGI. Z počátku devadesátých let dvacátého století společnost SGI přepracovala svou knihovnu IRIS GL[1] a veřejně ji vydala pod názvem OpenGL jako průmyslový standard. Knihovna OpenGL tvoří základ knihovny OpenGL ES, jejíž vydání o více než 10 let později oznámilo konsorcium firem Khronos[2] na pravidelné konferenci SIGGRAPH[20].

Aktuální verze knihovny nese označení 3 a je zpětně kompatibilní s verzemi 1 a 2. Základní přehled jednotlivých verzích lze najít v [11], nebo v referenčních příručkách k jednotlivým verzím.

Dále v této kapitole lze nalézt informace o architektuře OpenGL ES a také několik sekcí zaměřených na problémy této knihovny z pohledu dnešní doby.

2.1 Architektura

OpenGL ES (i OpenGL) bylo navrženo s důrazem na to, aby byla tato knihovna použitelná na různých operačních systémech a grafických čipech. Knihovna je implementována v jazyce C, ale její rozhraní je vytvořeno tak, aby ji bylo možné používat v různých programovacích jazycích v závislosti na zvyklosti dané platformy, či preferenci vývojáře.

Architektura OpenGL ES je založena na konceptu klient-server. Klient překládá data do formátu, kterému grafický hardware rozumí a posílá je na server[3], kde jsou s ohledem na aktuální stav vykreslovacího řetězce (z angl. *graphics pipeline state*) zpracovány v pořadí ve kterém přicházejí.

2.2 Základní pojmy

Základním stavebním prvkem OpenGL je tak zvaný kontext (*angl. context*). Kontext je datová struktura, která obsahuje veškeré stavové informace potřebné k vykreslení scény. Mezi

[1] IRIS GL - Silicon Graphics IRIS Graphics Library

[2] The Khronos Group `https://www.khronos.org`

[3] Server v OpenGL může, nebo nemusí běžet na stejném zařízení.

tyto informace patří například stavové proměnné pro funkce, které nejsou programovatelné (*angl. fixed-functions*)[4], vertex a fragment shadery, pole vrcholů a také v neposlední řadě aktuální *framebuffer*. OpenGL zná dva druhy framebufferů. První výchozí framebuffer (*angl. default framebuffer*), který je právě součástí OpenGL kontextu a obvykle je reprezentován oknem aplikace (případně obrazovkou, či jiným zobrazovacím zařízením). Druhým druhem jsou framebuffer objekty (*angl. framebuffer objects*), které jsou reprezentovány například texturou a nikdy nejsou přímo viditelné. Tyto objekty, lze použít například k vykreslování stínů technikou shadow mapping.

Objekty OpenGL ES

Dalším důležitým pojmem v souvislosti s OpenGL ES jsou OpenGL objekty (*angl. OpenGL objects*). Tyto objekty dovolují zapouzdřit více stavů a poté všechny tyto stavy nastavit voláním jediné funkce. Důležité je, že stav uchovaný v OpenGL objektech je validován pouze při prvním použití[5]. Následkem tohoto faktu je nižší režie a vyšší výkon při vykreslování. Je také vhodné si uvědomit, že o alokaci a dealokaci paměti pro tyto objekty se stará OpenGL a nikoliv vývojář.

OpenGL rozlišuje objekty do dvou kategorií. První mezi které patří třeba *buffer* objekty, nebo objekty pro práci s texturou. Objekty spadající do druhé kategorie se nazývají kontejnerové (*angl. container objects*) a patří mezi ně např. dříve zmiňovaný *framebuffer* objekt, nebo *vertex array object (VAO)*. Kompletní seznam objektů lze najít v dokumentaci (např. [23]). Mezi nejdůležitější patří *buffer objects* a *vertex array object (VAO)*.

Buffer objekty, jsou objekty pro správu dat pevné velikosti v (serverové části) paměti OpenGL ES. Mezi takové objekty patří například *vertex buffer object (VBO)* pro uložení pole vrcholů, nebo objekt pro data která jsou stejná po celou dobu vykreslování jednoho snímku (*uniform buffer object*). *VBO* umožňuje aplikaci alokovat paměťový prostor přímo v paměti GPU, čímž se zabrání opětovnému posílání dat pokaždé, když má být dané primitivum vykresleno. Podle [11] tento přístup umožňuje výrazně zvýšit výkon vykreslování a také omezit paměťový provoz, tedy snížit spotřebu energie. Nicméně *VBO* neobsahuje žádné informace o typu uložených dat. Pro uložení těchto parametrů je ideální použít strukturu *vertex array object*, která tato nastavení zapouzdřuje.

2.3 OpengGL a TBDR

OpenGL (ES) API není navrženo s ohledem na *TBDR* (více o TBDR, lze najít v kapitole 1), ale na tzv. okamžitý mód vykreslování při kterém se snímek vykresluje hned jakmile jsou dostupná požadovaná data. *TBDR* architektura se naopak o výsledné scéně snaží získat co nejvíc informací před tím než začne rasterizace a následné procesy. Tyto informace se shromažďují ve struktuře *parameter buffer*, která narůstá s počtem *draw calls* ovlivňujících výsledný snímek. Tato data by měla být správně smazána na začátku každého vykreslovaného snímku např. voláním funkce `glDiscardFramebufferEXT()`. Problémem může být, že OpenGL nedefinuje jeden průchod vykreslovacím řetězcem.

Více doporučení týkajících se práce s OpenGL knihovnou na *TBDR* architektuře, jako například fakt, že není nutné řadit neprůhlednout geometrii pro zabránění překreslování, lze najít v článku *Perfromance Tuning for Tile-Based Architectures* z knihy [8].

[4]Mezi funkce které nejsou programovatelné patří např. rasterizace

[5]Dle Apple WWDC (2010) vývojáři někdy používají techniku při které část scény pomocí daného objektu vykreslí do framebufferu, který není prezentován, aby zabránili první validaci v rámci vykreslovací smyčky.

2.4 Problémy OpenGL

Ačkoli se tato sekce nazývá problémy OpenGL je potřeba uvést věci na pravou míru. Dále se práce zabývá okolnostmi, které stály za vznikem nových nízkoúrovňových knihoven, jako je Metal a Vulkan (viz. následující kapitola 3), ale ne pro každou aplikaci musí být tyto "problémy" opravdu relevantní. Jde spíše o věci, které brání aplikacím dosáhnout ještě vyššího výkonu, či efektivnějšího běhu s ohledem na baterii.

Změna vykreslovacího řetězce

Každý *draw call* vyžaduje stavový vektor, který je dán kombinací všech stavových proměných. Faktem je, že změna stavu může být náročná a to jak na straně hardwaru, kde musí proběhnout aktualizace hodnot registrů, tak i na straně klienta, kde probíhá validace a překlad dat do formátu odpovídajícímu aktuálnímu stavu. Potenciálním problémem pak může být fakt, že veškeré operace související se změnou stavu jsou zahájeny až ve chvíli, kdy je volána funkce zajišťující vykreslení primitiv. Tedy v době, kdy má být uživateli prezentován výsledný snímek a v náročných scénách může být zapotřebí veškerého výkonu pro vykreslování.

Vícevláknové aplikace

V náročných scénách se může ukázat, že slabým článkem vykreslování je CPU. OpenGL (ES) pochází z doby, kdy CPU měly jedno jádro. To je velmi odlišný stav od současného, kdy se běžně používají vícejádrové procesory (s několika vlákny na jedno jádro). To znamená, že i když se CPU ukazuje jako slabý článek ve skutečnosti nemusí být plně využit. Bylo by tedy vhodné rovnoměrně zatížit všechna jádra (vlákna) čehož lze částečně docílit několika způsoby. Při jednom z nich se například na hlavním vlákně provádí vlastní vykreslování a na druhém časově náročné operace jako načítání textur, nebo kompilace shaderů[6][9].

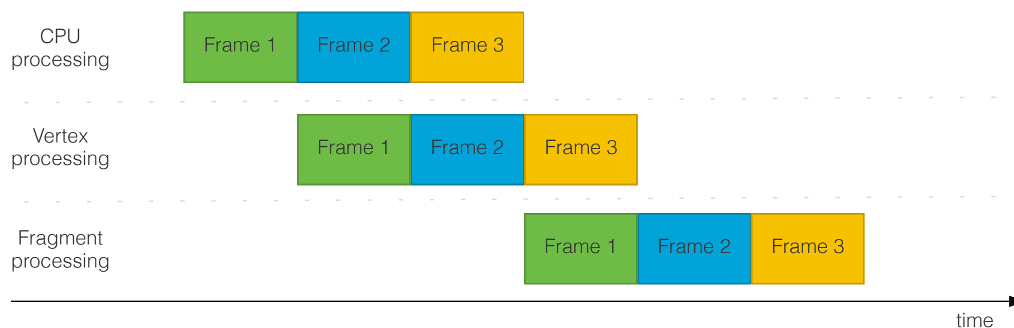
Za pomoci tohoto přístupu ovšem nelze efektivně vytížit všechna jádra. V článku [12] se lze dočíst o následujících problémech. Jako jeden z problémů je uvedena skutečnost, že většina funkcí v OpenGL ES je *thread safe* a to často znamená uzamykání přístupu do takových funkcí pro více než jedno vlákno a tedy čekání ostatních vláken. Dále fakt, že ačkoli aplikace může obsahovat více než jeden kontext, tak vlákno může pracovat pouze v jednom kontextu. To u většiny volání knihovních funkcí znamená vyhledání kontextu pro aktuální vlákno a tedy další procesorový čas. Autor také považuje za problém, že OpenGL ES nerozlišuje mezi generováním příkazů a jejich odesíláním na GPU.

Správa prostředků

V části o OpenGL objektech (2.2) se lze dočíst, že OpenGL se stará o alokaci a dealokaci objektů. Ve skutečnosti se obvykle stará o veškerou práci s pamětí a také synchronizaci.

Autoři [8] poukazují na situaci, kdy by měl vývojář na tento fakt pamatovat obzvláště pokud pracuje na mobilní zařízení, které je limitováno mj. velikostí paměti. V modelové situaci hraje roli zpoždění při vykreslování. Jak ukazuje obrázek 2.1 v určitém čase může být pomocí CPU zpracováván jeden snímek, jiný pak může být ve fázi zpracování vrcholů (*vertex processing*) a na třetím snímku se již můžou zpracovávat jednotlivé fragmenty (*fragment processing*). Pokud se pak vykresluje animovaný objekt, jehož pozice se každým snímkem mění, tak může dojít k situaci při které CPU modifikuje data se kterými ještě pracuje *vertex shader*. V takové situaci OpenGL vytvoří kopii dat, aby se nemuselo čekat na dokončení

předchozí operace. Článek pak uvádí, že situace může být horší pokud se objekt nachází na snímku vícekrát.



Obrázek 2.1: Zpracování snímků u *TBDR* architektury.

Další situace uvádí, také příručka pro vývojáře společnosti Qualcomm[6]. Mezi něž patří, například volání funkce `eglBufferSubData()`, které může v průběhu vykreslování vyústit v kopii dat. Nebo také skutečnost, že standardně dochází při volání funkce `eglSwapBuffers()` ke kopii depth bufferu z předchozího snímku.

2.5 Shadery

Pro programování programovatelných částí grafického řetězce slouží v OpenGL (ES) jazyk GLSL (z anglického OpenGL Shading Language). Jazyk GLSL je založen na ANSI C, který obohacuje o vektorové a maticové datové typy. Dále je také rozšířen o některé mechanismy jazyka C++, například o přetěžování funkcí.

Ve verzi OpenGL ES 3.0 jsou dostupné dva typy shaderů. První slouží pro zpracování vrcholů (*vertex shader*) a druhý pro zpracování fragmentů *fragment shader*. Novější verze OpenGL ES 3.2, která byla vydána v roce 2015 podporuje mimo vertex a fragment shadery, také shadery pro úpravu geometrie (*geometry shaders*) a pro výpočetní operace (*compute shaders*). Nicméně tato verze není dostupná na systémech iOS.

Překlad a linkování shaderů na platformě iOS v OpenGL ES probíhá vždy za běhu aplikace. Ačkoli, jak uvádí [20] jiné platformy můžou poskytovat více možností.

[6]Webové stránky společnosti Qualcomm https://www.qualcomm.com a její příručka pro OpenGL ES vývojáře https://developer.qualcomm.com/download/adrenosdk/adreno-opengl-es-developer-guide.pdf

Kapitola 3

Nízkoúrovňové knihovny

V současné době lze pozorovat snahu o nahrazení knihovny OpenGL, resp. OpenGL ES jinými nízkoúrovňovými knihovnami, které by přinesly lepší využití systémových prostředků. Tato kapitole se zaměřuje na dvě takové knihovny. První z nich je knihovna společnosti Apple Inc. s názvem Metal. Druhá je vyvíjena konsorciem firem The Khronos Group Inc. a nese název Vulkan.

V první části se kapitola věnuje základním informacím o knihovně Metal. Závěrečná část kapitoly se zaměřuje na knihovnu Vulkan. Jelikož se aktuálně jeví, jako velmi nepravděpodné, že by společnost Apple na svém mobilním systému povolila vedle vlastní knihovny i knihovnu Vulkan, tak se jedná spíše o shrnutí hlavních vlastností této knihovny.

3.1 Metal

Knihovnu Metal představil v roce 2014 na vývojářské konferenci WWDC zástupce společnosti Apple, Craig Federighi. Společnost Apple ve svých materiálech [4] definuje Metal, jako knihovnu pro 3D vykreslování a paralelní výpočty[1] na grafickém čipu, jejímž hlavním cílem je minimalizovat režijní náklady procesoru spojené s prováděním úkonů na GPU. Na rozdíl od OpenGL ES je Metal navrhnut výhradně pro systémy společnosti Apple. Z počátku byla tato knihovna dostupná pouze na operačním systému pro přenosná zařízení iOS od verze 8 a výše. O rok později se, ale rozšířila i na desktopový operační systém Mac OS X El Capitan a jeho novější verze.

Mezi body, které označil Jeremy Sandmel při prezentaci knihovny jako klíčové, figuruje i odrážka s textem *“Dělat náročné věci méně často”* [21]. K pochopení je třeba nastínit princip vykreslování pomocí OpenGL (ES). Ten, lze považovat za posloupnost událostí, jako například nastavení *shader*-ů, textur a *vertex bufferu* a volání kreslící funkce. V okamžiku volání kreslící funkce se začne provádět validace stavu (a třeba i kompilace *shader*-ů) a také samotná operace na GPU. Poté se opět tyto posloupnosti opakují pro jiný vykreslovaný objekt a tak dále. Knihovna Metal volí přístup při kterém se náročné operace jako validace stavu, případně kompilace *shader*-ů neprovádějí uvnitř vykreslovací smyčky. Kompilace *shader*ů probíhá již při samotné kompilaci aplikace a validace stavu při inicializaci. Uvnitř vykreslovací smyčky se provádí pouze operace na GPU.

Jednou z dalších odrážek je i *“Optimalizováno s ohledem na chování CPU”*. Za touto odrážkou, lze najít více věcí, ale třeba i skutečnost o které se práce zmiňuje v kapitole

[1] Knihovna poskytuje prostředky i pro paralelní výpočty na grafickém čipu. Toto téma, ale není předmětem této práce a tak se jím nebude více zabývat.

2.4. Tedy problémy OpenGL (ES) při práci s více vláknovými procesory. Respektive, že v rámci aktuálního kontextu může generovat příkazy pouze jedno vlákno (ostatní vlákna můžou sloužit například pro načítání zdrojů) a díky tomu nelze příliš dobře distribuovat práci mezi všechna jádra. Oproti tomu v knihovně Metal se nevyskytuje podobně "globální" kontext, který by působil podobné potíže pří práci s více vlákny.

Mezi další klíčové body společnost zařadila také *"Schopnost využívat moderní GPU"* (viz. 3.1.4) případně *"Explicitní generování/odesílání příkazů"* (viz. 3.1.2).

3.1.1 Rozhraní knihovny Metal

Knihovna Metal je stejně jako většina jiných knihoven společnosti Apple napsaná v jazyce Objective-C nad knihovnou Foundation[2]. Podle autora [16] lze považovat tohle protokolově založené API za mnohem modernější abstrakci vykreslovacího řetězce než je OpenGL ES.

Mezi klíčové objekty pro vykreslování v architektuře Metal-u patří objekt pomocí kterého se popisuje aktuální stav vykreslovacího řetězce (*pipeline state object*). Dále také objekt reprezentující jeden průchod vykreslovacím řetězcem (*render pass descriptor*) a v neposlední řadě i *render command encoder*, který slouží ke generování příkazů pro GPU v souvislosti s vykreslováním (kapitola 3.1.2). Kompletní grafický přehled architektury lze najít v prezentaci z vývojářské konference [21].

Pipeline state object

Autor článku [7] mezi výhody knihovny uvádí právě jeden z bodů, na které se tvůrci knihovny při jejím návrhu zaměřili, a to *"Dělat náročné věci méně často"*. Tedy schopnost před-vyhodnotit stav vykreslovacího řetězce, tak aby k tomu nedocházelo v uvnitř vykreslovací smyčky. K tomu slouží právě *pipeline state object* dále jen *PSO* (v dokumetanci [4] ho lze nalézt pod názvem `MTLRenderPipelineState`).

PSO je neměnný objekt, který zapouzdřuje informace, jako formát vstupních dat, nebo například používané vertex a fragment funkce. Kompletní přehled lze najít ve výše odkazované dokumentaci.

Render pass descriptor

Render pass descriptor slouží jako kontejner pro přílohy (z anglického *attachments*). Umožňuje pojmout až čtyři přílohy pro barvu, jednu pro paměť šablony (*stencil*) a jeden cíl pro hloubku.

Pro každou s těchto šablon lze specifikovat tzv. *load and store* akce (kapitola 3.1.4).

3.1.2 Generování příkazů

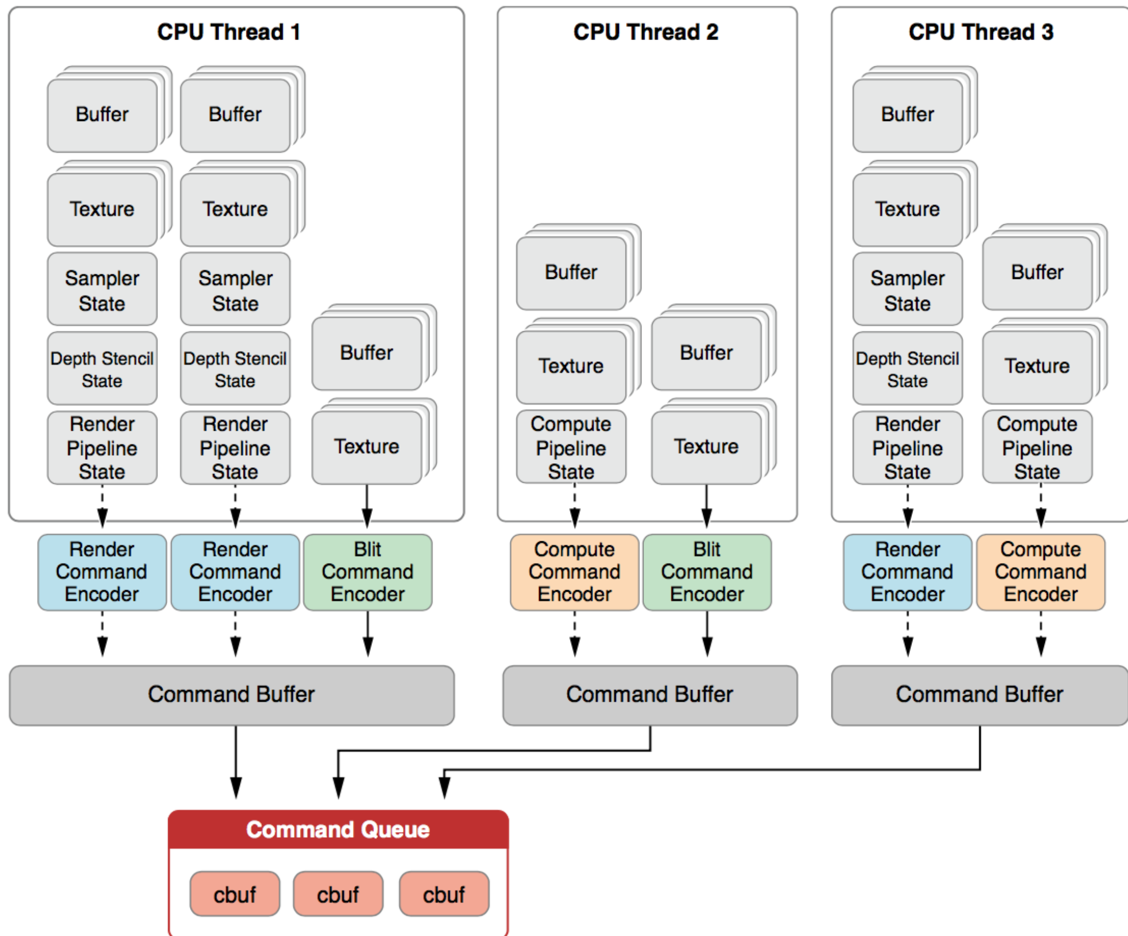
Pro generování se v knihovně Metal používájí objekty, zvané enkodéry, které převádí volání knihovních metod na HW příkazy. Metal rozlišuje čtyři druhy enkodérů. Dva pro vykreslování *render command encoder* a *parallel render command encoder*, další pro graficky akcelerované operace s daty a texturami (*blit command encoder).* A jeden pro obecné výpočty na GPU (*compute command encoder*).

Render command encoder-y jsou objekty sloužící pro generování HW příkazů pro jeden průchod vykreslovacím řetězcem. Nicméně před samotným generováním příkazů je potřeba

[2]Knihovna Foundation https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/ObjC_classic/.

specifikovat určité informace. Mezi které patří vstupní geometrie, používané textury a také objekty zapouzdřující stav. Nejdůležitějším stavovým objektem je již zmíněný *pipeline state object*.

HW příkazy generované pomocí jednoho z enkodérů jsou uchovávány ve struktuře zvané *command buffer*. V souvislosti s použitím enkodérů, lze generovat tyto struktury na několika vláknech současně (obr. 3.1) a co je důležitější v této fázi již neprobíhá žádná validace, nýbrž se jedná o přímou komunikaci s ovladačem GPU.



Obrázek 3.1: Schéma generování příkazů v knihovně Metal. Obrázek je převzat z [4]
.

Na obrázku 3.1 si lze povšimnout i toho, že do jednoho *command buffer*-u, lze generovat příkazy z několika enkodéru. V Metal-u platí, že jeden *command buffer* zpravidla obsahuje veškeré příkazy pro vykreslení jednoho snímku a to i v případě, že jeden snímek je složen z několika průchodů vykreslovacím řetězcem. Tento buffer příkazů reprezentuje také jedinou sledovatelnou “jednotku práce”. Začátek provádění příkazů explicitně určuje vývojář, který může být také informován o dokončení těchto operací.

3.1.3 Správa prostředků

Správa prostředků je navržena s ohledem na sdílený paměťový systém mobilních zařízení. Z toho důvodu zde nejsou žádné implicitní kopie. Samotná knihovna také neiplementuje

žádnou formu ochrany proti současnému přístupu CPU a GPU ke stejné části paměti. Synchronizace je tedy odpovědností programátora. Na druhou stranu tento přístup přináší vyšší výkon, protože nemusí být aplikovaná ochraná opatření na prostředky u kterých podobné problémy nehrozí.

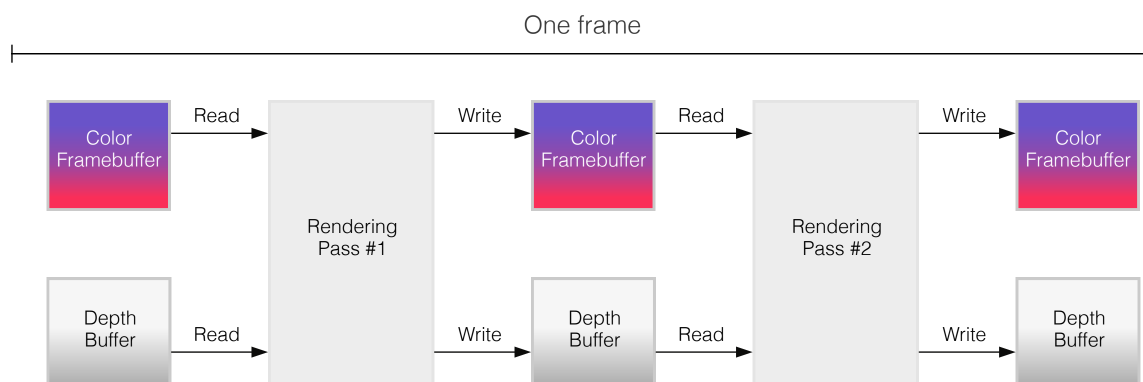
Metal rozlišuje pouze dva druhy vstupních dat. První pro ukládání formátovaných obrazových dat, jako jsou textury a druhý pro libolná data, který se používá pro uložení vstupní geometrie a podobně.

Důležitým faktem je, že jakmile je jednou specifikován formát vstupních dat, tak jej nelze změnit a musí být případně vytvořena např. nová textura. Tím se knihovna vyhýbá náročné validaci v průběhu vykreslování a na místo toho vede vývojáře k rychlejšímu “přepínání” mezi již vytvořenými texturami.

3.1.4 Metal a TBDR

Knihovna Apple byla navržena pro čip Apple A7, který v sobě ukrývá grafický čip architektury *TBDR* (1). S ohledem na tento čip a na fakt, že Metal vyžaduje explicitní popis průchodu vykreslovacím řetězcem (*render pass descriptor*), lze docílit značné úspory v přesunech paměti.

Následující obrázky pocházející z konference, na které byla knihovna Metal představena. Obrázky znázorňují vykreslování snímku pomocí techniky jež vyžaduje dva průchody. Mezi takové techniky patří například *deffered shading*.

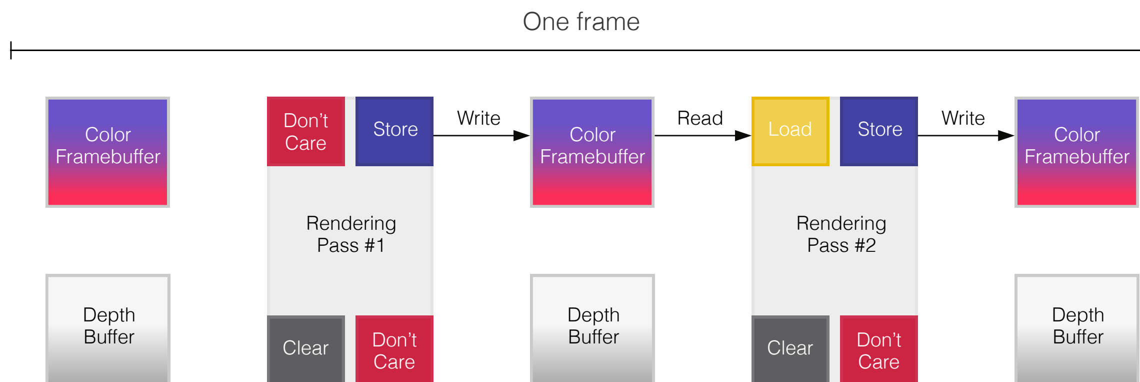


Obrázek 3.2: Bez specifikace akcí tzv. load and store akcí. [3]

Na obrázku 3.2 můžeme vidět stav, kdy se vývojář nezabývá optimalizací, nebo mu to není pomocí nástrojů, které používá umožněno. Lze vidět, že celkem je pro vykreslení snímku potřeba 8 paměťových operací (4 pro čtení a 4 pro zápis).

Druhý obrázek 3.3 ukazuje stav, kdy vývojář použije např. knihovnu Metal a specifikuje tzv. *load and store* akce. Tedy akce, které specifikují, zda je obsah předchozího *color bufferu*, nebo *frame bufferu* užitečný pro další průběh vykreslování a má být zkopírován ze systémové paměti do rychlé paměti na čipu a zda má být obsah toho aktuálního uložen (tedy zkopírován z rychlé paměti na čipu do systémové paměti). Zde je za pomocí *load and store* akcí redukován počet paměťových operací z osmi na tři.

[3] Původní obrázek je dostupný v [21].
[4] Původní obrázek je dostupný v [21].



Obrázek 3.3: S použitím tzv. load and store akcí [4]

3.1.5 Shadery

Pro psaní shaderů se v Metalu používá jazyk založený na jazyku `C++ (11)`, tedy jeho statické podmnožině s určitými rozšířeními a omezeními. Tato rozšíření se týkají zejména zlepšení možnosti interakce s GPU a také možnost přetěžování funkcí a šablony. Nelze ovšem používat rekurzi [5].

Na rozdíl od OpenGL (ES) obvkyle nejsou shadery kompilovány za běhu[5] aplikace, ale během překladu aplikace.

Jedním z nedostatků Metal-u oproti nejnovějším verzím knihovny OpenGL ES je absence shaderu pro geometrii.

3.2 Vulkan

Knihovna Metal je vyvíjena firmou Apple pouze pro použití na jejich zařízeních, Vulkan se snaží o přesně opačný přístup. Tedy o otevřené, multiplatformní řešení, které by se dalo použít na počítačích, přenosných zařízeních i konzolích.

Vulkan stejně jako OpenGL vyvíjí konsorcium firem The Khronos Group Inc. Specifikace první verze byla vydána 16. února roku 2016. Cílem knihovny Vulkan není (na rozdíl od knihovny Metal) nahradit knihovnu OpenGL (ES), ale doplnit ji.

Knihovna je založena na systému modulů, které vývojář může použít, nebo nemusí. Mezi takové moduly patří například modul pro validaci a debugging. Z programátorského hlediska je rozhraní knihovny Vulkan podobné OpenGL, tedy jedná se o API v jazyce C. Ovšem oproti OpenGL je alespoň striktně typované.

Oproti knihovně Metal je Vulkan "dospělejší" z hlediska podpory více GPU a například shaderů pro geometrii a teselaci. Nevýhodou může být větší implementační náročnost na kterou mimo jiné poukazuje i officiální prezentace [24] i když s vidinou dosažení vyššího výkonu.

Podobně jako Metal, nebo Metal podobně jako Vulkan, cílí knihovna na snížení režijních nákladů spojených s vykreslováním, explicitnější kontrolu a předvídatelnější chování ovladače grafického čipu. Vulkan nabízí podobný model při práci s více vlákny, kdy různá procesorová vlákna můžou generovat obálky s příkazy (známé jako *command buffer*), které

[5] Vertex a fragment funkce v knihovně Metal obvykle nejsou kompilovány za běhu, ale i taková možnost existuje.

jsou serializovány do jedné fronty. Oproti knihovně Metal je ovšem možné tyto obálky používat opakovaně. Podobný přístup ke generování příkazů je pouze jeden společný znak. Mezi další patří také obdobný přístup k popisu scény pomocí objektu stavu vykreslovacího řetězce (*pipeline state object*). Podobně jako v knihovně Metal i zde tento objekt zapouzdřuje většinu ze stavového vektoru GPU a výsledná aplikace přepíná mezi již vytvořenými stavy, které jsou validovány ideálně při inicializaci. Zejména pro mobilní platformu je důležitá také podpora architektur využívajících vykreslování po dlaždicích a informovanost o jednotlivých průchodech vykreslovacím řetězcem [25].

Odlišný přístup volí Vulkan při programování shaderů. Na místo volby jednoho jazyka používá jistou formu mezikódu `SPIR-V` do kterého jsou ostatní jazyky převáděny. V současné době existuje podpora například pro jazyk `GLSL`, který k tomuto účelu používá knihovna OpenGL (ES).

Kapitola 4

Měření výkonu vykreslování

Zatímco na grafických čipech s okamžitým režimem vykreslování (1.1), lze pro měření výkonu použít tzv. *timer queries*, které nám dovolí změřit časovou náročnost vykreslení scény, nebo její části. U *tile-based* čipů je situace složitější. Bruce Merry, autor článku *Performance Tuning for Tile-Based Architeruces* ze sbírky [8] uvádí, že ačkoliv by bylo možné podobnou funkcionalitu naprogramovat, tak by nebyla příliš užitečná. Důvodem je to, že u čipů, které provádí vykreslování po dlaždicích nejsou příkazy prováděny v pořadí ve kterém byly odeslány ke zpracování. Čipy se snaží provést veškeré zpracování vrcholů v jednom průchodu a zpracování fragmentů v průchodu následujícím. Autor pro detekci případných příčin nízkého výkonu doporučuje použití nástrojů přímo od výrobců konkrétních grafických čipů.

Pokud se ovšem najde situace, kdy by bylo vhodné použít klasické metody, nebo v případě, že nejsou dostupné nástroje od výrobců hardwaru, tak OpenGL dokumentace varuje na častou chybu při měření výkonu, kdy je použita nějaká forma následující konstrukce (4.1).

```
start_the_clock()
draw_a_bunch_of_polygons()
stop_the_clock()
swapbuffers()
```

Listing 4.1: Chybná forma měření času vykreslování.

Autoři varují, že implementace OpenGL (ES) téměř vždy obsahuje nějakou formu zřetězeného zpracování a pokračování běhu programu po návratu z vykreslovací funkce nemusí znamenat, že byla odeslaná geometrie skutečně vykreslena. Dále také uvádějí, že umístění funkce `glFinish()` před funkci pro ukončení měření nezaručí, že při odeslání příkazů na GPU již není GPU vytížena zpracováním jiných nedokončených úloh. Další informace lze najít přímo na wiki stránkách OpenGL[1].

Další skutečností na kterou je si potřeba také dávat pozor v souvislosti s měřením výkonu na mobilních zařízeních společnosti Apple je to, že obnovovací frekvence obrazovky je 60Hz. Vykreslovací funkce tedy nebude volána v kratším časovém intervalu než je 1/60 vteřiny [6].

[1]Wiki stránky OpenGL zaměřené na měření výkonu vykreslování `https://www.opengl.org/wiki/Performance`.

Kapitola 5

Experimenty

Tato kapitola obsahuje informace o implementovaných experimentech. Výsledky těchto experimentů, lze najít v kapitole 7. První expertiment ověří, zda se i v OpenGL kontextu použije vykreslování po dlaždicích. Před dalším experimentem je uvedeno nezbytné teoretické minimum pro pochopení částicových systému, na kterých je založen druhý experiment.

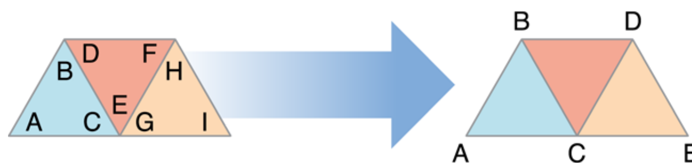
5.1 Ověření vykreslování po dlaždicích v OpenGL kontextu

Při vykreslování na IMR zařízeních v OpenGL dochází k rasterizaci po čtvercích o velikosti $2x2$ pixely. Vyplnění obrazovky nebo jiné plochy primitivy o velikosti $2x2$ pixely by tedy mělo být rychlejší než vyplnění prostoru stejné velikosti shodným množstvím geometrických primitiv o délce 4 pixely a výšce jednoho pixelu. Nejen z toho důvodu by mělo být také vyplnění plochy pomocí bodů o rozměrech jednoho pixelu nejpomalejší.

Na zařízeních využívající vykreslování po dlaždicích by rozdíl mezi vykreslením primitiv o velikostech $2x2$ a $4x1$ pixel měl být minimální. Vykreslování bodu za bodem by, ale i zde mělo být nejpomalejší.

Pro dosažení stejného počtu vrcholů a dodržení správných postupů práce s OpenGL ES, lze využít reprezentace dat, kdy jsou trojúhelníky seřazeny do pruhů (*triangle strip*) (v OpenGL `GL_TRIANGLE_STRIP`) a techniky zvané `GL_PRIMITIVE_RESTART`.

Použitím prvního z výše uvedených, tedy `GL_TRIANGLE_STRIP`, lze dosáhnout snížení počtu vrcholů, které je nutné zpracovat pomocé vertex shaderu.



Obrázek 5.1: Pruh trojúhelníků (*triangle strip*). Obrázek je převzat z [15].

Na obrázku lze vidět, že vrcholy C,E a G ve skutečnosti značí stejný vrchol. Pomocí této formy hraniční reprezentace ovšem nemusí být uváděn vícekrát [6].

Druhá technika `GL_PRIMITIVE_RESTART` umožňuje vykreslit v rámci jednoho volání kreslící funkce[1] více takových složených pruhů.

[1] Pro použití funkce `GL_PRIMITIVE_RESTART` musí být využita kreslící funkce využívající pole indexů, tedy například `glDrawElements`

5.2 Částicové systémy

Jedna z prvních (a stále platných) defincí částicového systému pochází z roku 1983 a byla publikována v článku [19]. Volný překlad by mohl znít i následovně: *"Částicový systém je kolekce mnoha objektů (částic), které dohromady reprezentují fuzzy objekt. Částice jsou v průběhu času přidávány do systému. V rámci systému se také pohybují či jinak mění a případně jsou ze systému (nebo spolu s ním) odstraněny."*

Jako první byl částicový systém použit ve filmu Star Trek II: Khánův hněv pro simulaci exploze torpéda. To je stále jedno z mnoha využití i v dnešní době, kdy jsou částicové systémy používány v mnoha animacích, videohrách a tak dále.

Částicový systém můžeme rozdělit na dvě části. První část se nazývá emitor, který slouží jako generátor částic. Druhou části jsou samotné částice. Při animaci exploze torpéda je místo výbuchu emitorem částic. Pseudonáhodně, případně jinak řízené částice, poté navozují dojem exploze.

Částicový systém, lze vytvořit různými způsoby jednou z možností je použití tzv. *point sprites*, kdy je částice tvořena jedním bodem a animovat pohyb těchto bodů kompletně ve vertex shaderu. Takový příklad je uveden mj. v knize [11]. V příkladu je cílem vytvořit animaci exploze. Systém je zde definován za pomoci několika vlastností, které se vztahují, jak k emitoru, tak i k částicím. Mezi vlastnosti emitoru patří pozice, barva a čas exploze. Každá částice je navíc definována pomocí počáteční a cílové pozice a také času během kterého se podílí na výsledné animaci. Celá animace pak funguje následovně, emitoru se definuje pozice v rámci scény a také jeho zbývající vlastnosti, tedy barva a čas exploze. Poté co jsou v rámci definovaného rozsahu náhodně přiřazeny vlastnosti i všem částicím se ve vertex shaderu vypočítá poloha jednotlivých částic. Výpočet je založen na základě uplynulého času animace, který určuje pozici mezi počátečním a cílovým bodem každé částice. V případě, že je překročen čas života částice je možné ji nastavit pozici mimo viditelnou část scény. Tento přístup má jisté výhody, ale také nevýhody. Mezi výhody patří jednoduchost a také rychlost. Mezi nevýhody kniha řadí, že všechny částice jsou vygenerovány ve stejnou chvíli a jejich pohyb je omezen výsledkem lineární interpolace výchozí a cílové pozice.

Jednou z dalších možností je počítat pozici částic pomocí CPU a vypočtenou polohu předávat při každém vykreslení nového snímku grafickému čipu. Výhodou tohoto přístupu je, že pohyb částic může být mnohem komplikovanější, resp. pro určité scény realističtější. Takový přístup je využit i ve druhém experimentu. V rámci kterého je součástí scény jeden emitor, který v konstantním časovém intervalu generuje předem stanovené množství částic.

Další užitečné informace o částicových systémech se dají najít například v knize [22], nebo již zmiňovaném článku [19].

Kapitola 6

Implementace

Tato kapitola obsahuje základní informace o vývoji mobilních aplikací pro systém iOS. Patří mezi ně například informace o vývojových nástrojích a programovacím jazyku, ale také o základních návrhových vzorech, které je potřeba před vlastní implementací znát.

V další částí kapitoly se práce soustředí na jeden z frameworků, který byl při vlastní implementaci použit a to *GLKit*. Mezi další použité frameworky patří *Foundation*[1] a *Model I/O* o kterém lze najít více informací v [18]. Jedním z důležitých nástrojů při práci s knihovnou Metal je také *Grand Central Dispatch (GCD)* [1].

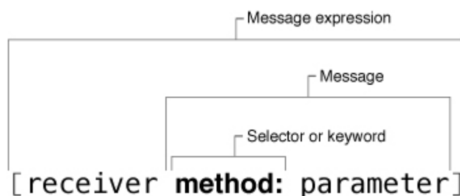
Závěrečná část kapitoly srovnává některé části implementace testovacích aplikací napříč knihovnami OpenGL ES a Metal.

6.1 Vývoj aplikací pro systém iOS

Základem pro vývoj iOS aplikací je *vývojové prostředí xCode*[2], které zahrnuje i iOS SDK. Určitým omezením je, že xCode lze používat pouze na počítači s operačním systémem Mac OS X. Xcode zahrnuje, také ucelenou sadu nástrojů *Instruments* pro testování nejrůznějších částí výsledné aplikace.

Primárním programovacím jazykem pro aplikace na systému iOS je *Objective-C* (ačkoli je postupně nahrazováno jazykem Swift). Objective-C je objektově orientovaný jakyk, který dědí syntaxi, primitivní typy a řídící struktury jazyka C. Navíc přidává syntaxi pro definici tříd a metod. Narozdíl od jazyka C se jedná o jazyk dynamicky typovaný [3].

Objekty v jazyce Objective-C spolu komunikují zasíláním zpráv. Terminologii zprávy, lze vidět na obrázku 6.1.



Obrázek 6.1: Terminologie Objective-C zprávy [3]

[1] Referenční příručka k frameworku Foundation https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/ObjC_classic/

[2] Vývojové prostředí xCode https://developer.apple.com/xcode/

Zpráva je tvořena klíčovým slovem, nebo selektorem a parametry. Zaslání zprávy objektu způsobí buď invokaci příslušné metody, nebo vyjímku v případě že objekt tuto metodu neimplementuje a zprávě tedy nerozumí. V souvislosti s výslednými aplikacemi je také potřeba zmínit i existenci *Objective-C++*, který přináší do dynamického Objective-C aspekty jazyka C++. Objective-C++ se v souvislosti s knihovnou Metal často používá zejména kvůli `SIMD` vektorým datovým typům.

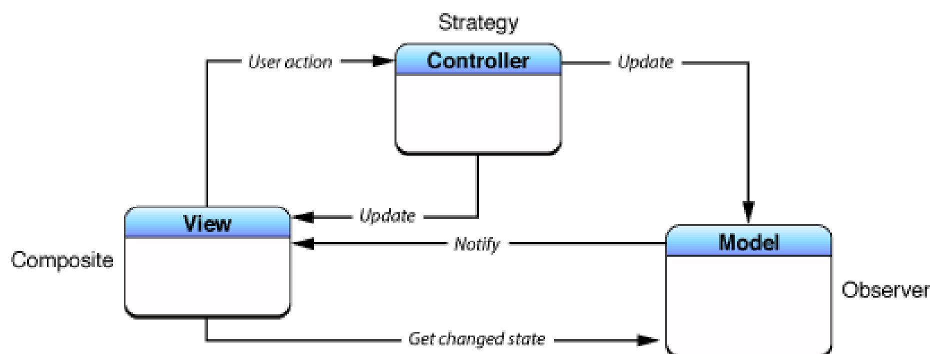
Návrhové vzory

Ačkoli by se správný popis návrhového vzoru měl podle [10] skládat ze čtyrech základních prvků, tedy názvu, popisu problému, řešení a v neposlední řadě důsledků, které aplikace daného vzoru způsobí, tak se zde zaměřím pouze na koncept daného vzoru v rozsahu jakém tato práce dovoluje.

MVC

MVC, nebo-li *Model, View, Controller*, kde *model* je označení pro datovou část aplikace, *view* slouží k prezentaci aplikace uživateli a *controller* má za úkol určitým způsobem propojovat právě uživatelské rozhraní a model.

MVC je zde uvedeno zejména z důvodu odlišnosti mezi tradiční verzí tohoto vzoru a verzí jak ji představuje společnost Apple ve svých materiálech k frameworku pro tvorbu uživatelských rozhraní *Cocoa*. Tradiční formu MVC zobrazuje obrázek 6.2 a "Cocoa verze" je zobrazena 6.3.

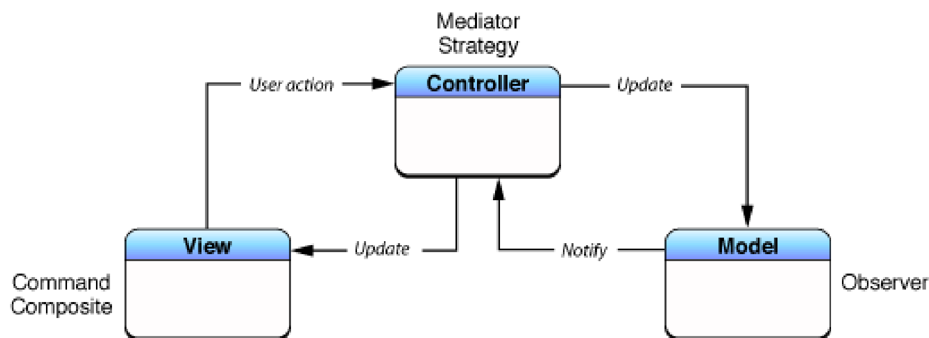


Obrázek 6.2: Tradiční verze MVC. Obrázek je převzat z [2].

Základním rozdíl mezi oběma verzemi je fakt, že v tradiční verzi je view obeznámeno s existencí modelu a opačně. Model, tak může informovat view, že proběhla nějaká změna v datové části, např. aktualizace dat pomocí vzáleného serveru a view pak provést příslušné akce. *Controller* se zde stará o správnou funkčnost s ohledem na uživatelské akce.

Oproti tomu "Cocoa verze" (obrázek 6.3) naprosto odstiňuje datovou část od uživatelského rozhraní za účelem dosažení větší znovupoužitelnosti a veškerá komunikace těchto dvou součástí probíhá přes prostředníka, tedy *controller* [2].

[3]Zdroj:`http://stpeterandpaul.ca/tiger/documentation/Cocoa/Conceptual/CocoaFundamentals/CocoaObjects/chapter_3_section_3.html`



Obrázek 6.3: Cocoa verze MVC. Obrázek je převzat z [2].

Delegace

Delegace je technika používaná v objektově orientovaném programování. Při delegaci jeden objekt deleguje akci na jiný informovanější objekt. Objekt na kterého je akce delegovaná pak rozhodne o dalších akcích, případně pro toto rozhodnutí využije i objekt, který akci deleguje.

Delegace je v Objective-C zpravidla spjata s protokoly. O protokolech a jejich využití nejen v souvislosti s delegací se lze dočíst zde [3].

6.2 GLKit

GLKit poskytuje funkce a třídy pro snadnější vytváření nových OpenGL aplikací využívajících shadery nebo k portování již existujích aplikací, které pro zpracování vertexů a fragmentů využívají funkce poskytované dřívějšími verzemi OpenGL[4].

GLKit mimo jiné poskytuje třídy `GLKViewController` a `GLKView`. `GLKView` je podtřídou `UIView`, tedy bázového *view*(6.1), která implementuje základní funkcionalitu pro práci s OpenGL ES. `GLKViewController` obsahuje mimo základní funkcionalitu bázového `UIViewController`-u také metodu `update` z vykreslovací smyčky.

Vykreslovací smyška je rozdělena na dvě části (obrázek 6.4)

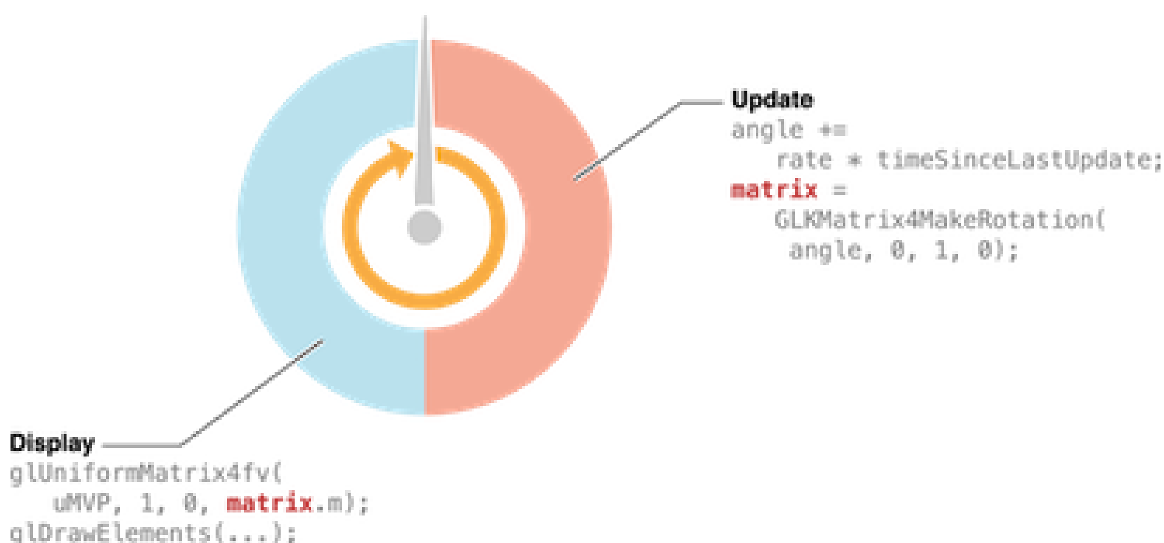
Konkrétně se jedná o část určenou pro vlastní vykreslování (na obrázku 6.4 označena jako `display`) a o část určenou k aktualizaci uniformních proměnných (na obrázku 6.4 označena jako `update`).

Jak je uvedeno výše, část vykreslovací smyčky pro aktualizaci uniformních proměnných je dostupná prostřednictvím objektu třídy `GLKViewController`, přesněji pomocí stejnojmené metody nebo za pomocí implementace protokolu `GLKViewControllerDelegate` na jiném objektu. Druhou část vykreslovací smyčky lze implementovat obdobným způsobem pomocí `GLKView` resp. protokolu `GLKViewDelegate`.

6.3 Srovnání rozhraní OpenGL ES a Metal

Z následujících několika ukázek si lze udělat představu o rozhraní obou testovaných knihoven. Doprovodný text, ale není referenční přiručkou k jednotlivým funkcím. Popis jednot-

[4]Volně přeloženo z `https://developer.apple.com/library/ios/documentation/GLkit/Reference/GLKit_Collection/`



Obrázek 6.4: GLKit – vykreslovací smyčka. Obrázek je převzat z [6].

livých funkcí se nachází v dokumentaci ke knihovně Metal[5] a OpenGL ES[6].

Práce s daty

První ukázka se zaměřuje na rozdíl ve správě dat, například vrcholů. Na ukázce 6.1 je vytvoření prostoru pro data v knihovně Metal. Jak je uvedeno v kapitole 3.1 Metal má veřejné rozhraní v jazyce Objective-C. Samotný *buffer* je zde reprezentován jako objekt implementující protokol `MTLBuffer`. K vytvoření takového objektu stačí poslat příslušnou zprávu specifikující velikost a další možnosti objektu reprezenující zařízení.

```
id<MTLBuffer> modelBuffer = [device newBufferWithLength:size
                                            options:0];
```

Listing 6.1: Chybná forma měření času vykreslování.

Oproti tomu OpenGL využívá API ve stylu jazyka C. Pro vytvoření prostoru pro data vrcholů je potřeba vytvořit identifikátor nového bufferu. Následujícím voláním funkce `glBindBuffer` specifikovat v jakém prostoru paměti se bude pracovat a také přidat identifikátor bufferu. Poté lze vytvořit (případně rovnou nakopírovat) data do dané oblasti.

```
glGenBuffers(1, &bufferID);
glBindBuffer(GL_ARRAY_BUFFER, bufferID);
glBufferData(GL_ARRAY_BUFFER,
        size), // size
        NULL,  // data
        GL_STATIC_DRAW); // type
```

Listing 6.2: Vytvoření místo pro data v OpenGL ES.

[5]Dokumentace ke knihovně online je dostupná na https://developer.apple.com/library/ios/documentation/Metal/Reference/MetalFrameworkReference/index.html

[6]Dokumentace ke knihovně OpenGL ES je dostupná online na https://www.khronos.org/opengles/sdk/docs/man3/.

Pokud budeme chtít upravit pouze část obsahu bufferu v OpenGL ES je potřeba pomocí funkce `glBindBuffer` a identifikátoru bufferu specifikovat část paměti. Následně voláním `glMapBufferRange` určit část paměti kterou chceme namapovat do adresového prostoru CPU a teprve nyní lze modifikovat danou část paměti. Po dokončení úprav je potřeba oznámit úpravy paměti funkcí `glFlushMappedBufferRange` a odmapovat buffer.

```
// Bind and map buffer.
glBindBuffer(GL_ARRAY_BUFFER, bufferID);

void *dst = glMapBufferRange(GL_ARRAY_BUFFER,
                            kOffset, bufferLength,
            GL_MAP_WRITE_BIT | GL_MAP_FLUSH_EXPLICIT_BIT |
            GL_MAP_UNSYNCHRONIZED_BIT);

// Modify buffer, flush, and unmap.
memcpy(dst, newData, newDataSize);

glFlushMappedBufferRange(GL_ARRAY_BUFFER,
                        kOffset,
                        bufferLength);

glUnmapBuffer(GL_ARRAY_BUFFER);
```

Listing 6.3: Modifikace části dat v OpenGL ES.

Oproti tomu v knihovně Metal stačí pouze vybranému objektu poslat zprávu `contents`. Výsledkem provedení této metody je příslušná adresa v paměti, ke které stačí pouze přičíst případný offset. Další operace nejsou nutné. Důležité je ovšem si uvědomit, že knihovna Metal ponechává veškerou synchronizaci na programátorovi a je tedy nutné věnovat pozornost tomu, aby se nemodifikovala část paměti, která může být používána grafickým čipem.

```
void *ptrData = (void *) ([_buffer contents] + kOffset);

// Copy data to CPU/GPU buffer
memcpy(ptrData, newData, newDataSize);
```

Listing 6.4: Modifikace části dat v Metal-u.

Specifikace formátu dat

Následující ukázky se zabývají stylem specifikace formátu dat vrcholů a také ukazují na jednu vlastnost typickou pro Objective-C a tou je “upovídanost”, která dle mého názoru přispívá k čitelnosti.

V knihovně Metal probíhá specifikace formátu vrcholů (ukázka 6.5) pomocí objektu třídy `MTL-VertexDescriptor`, který se podílí na celkovém popisu scény za pomocí objektů implementujících protokol `MTLRenderPipelineState`.

```
MTLVertexDescriptor *vertexDescriptor= [[MTLVertexDescriptor
      alloc] init];
vertexDescriptor.attributes[0].format = MTLVertexFormatFloat3;
vertexDescriptor.attributes[0].offset = 0;
```

```
vertexDescriptor.attributes[0].bufferIndex = 0;

vertexDescriptor.layouts[0].stride = 12;
vertexDescriptor.layouts[0].stepFunction =
      MTLVertexStepFunctionPerVertex;
```

Listing 6.5: Popis formátu vrcholů v knihovně Metal.

Oproti tomu na ukázce 6.6 z knihovny OpenGL ES je viditelná kratší syntaxe, ale bez komentářů může na nezasvěceného čtenáře působit nic nevypovídajícím způsobem.

```
glVertexAttribPointer(0,
                3, // size
                GL_FLOAT, // type
                NO, // normalized
                12, // stride
                0); // offset
```

Listing 6.6: Popis formátu vrcholů v knihovně OpenGL ES.

Kapitola 7

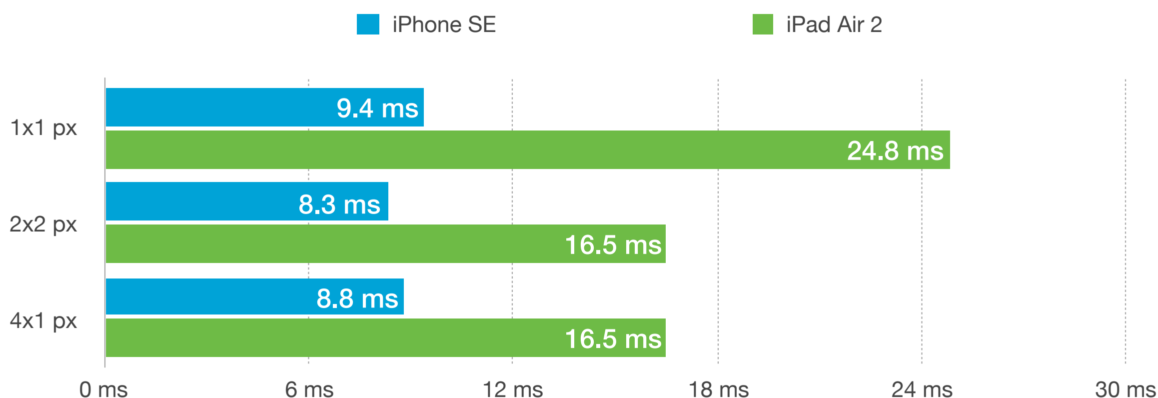
Vyhodnocení výsledků

Vyhodnocení výsledků bude probíhat na dvou zařízeních. První zařízení je iPad Air 2 (model A1566) s rozlišením 2048x1536 bodů a druhým zařízením je iPhone SE (model A1723) s rozlišením 1136x640 pixelů.

Ověření vykreslování po dlaždicích v OpenGL kontextu

Pro měření rychlosti vykreslování výsledné aplikace je použito vývojové prostředí *xCode* z důvodů uvedených v kapitole o měření výkonu vykreslování (4).

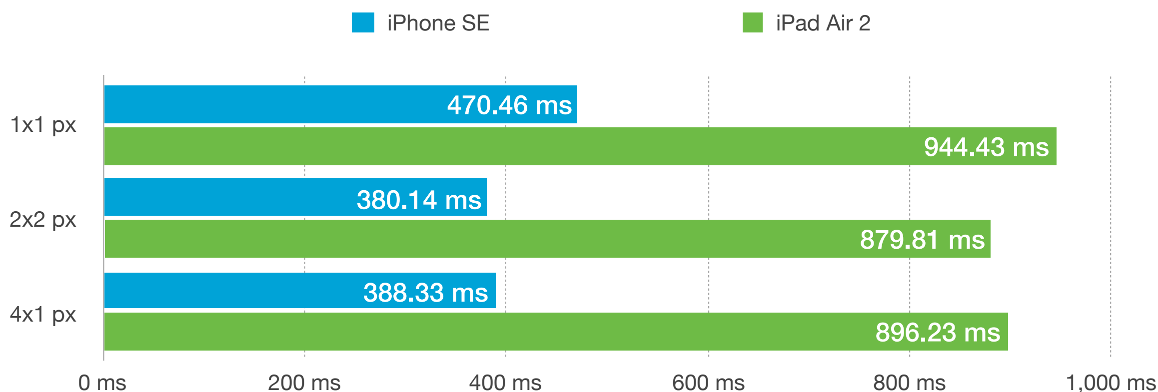
xCode prezentuje vývojáři mimo jiné údaje o využití jednotlivých částí zařízení a časech, které jsou potřeba pro provedení daných úloh na CPU i GPU.



Obrázek 7.1: Závislost času zpracování snímku na typu úlohy.

Graf na obrázku 7.1 ukazuje závislost času potřebného k vykreslení snímku na typu vykreslovaných primitiv. Z výsledných hodnot se dá vyčíst vyšší výkon telefonu iPhone SE. Výsledky také potvrzují použití vykreslování po dlaždicích i v OpenGL ES kontextu. Rozdíl mezi variantou při které jsou použity polygony o délce čtyř a výšce jednoho pixelu a variantou s polygony o délce strany dva pixely je v případě použití telefonu pouze 0.5ms ($\sim 6\%$). V situaci při které je testovaným zařízením iPad Air 2 je rozdíl dokonce nulový. V případě nepoužití *TBDR* by měl být rozdíl času potřebného k vykreslení snímku s primitivy 4x1px oproti 2x2px téměř dvojnásobný.

Graf 7.2 ukazuje výsledky situace při které bylo vykreslení primitiv zopakováno stokrát v rámci jednoho snímku.



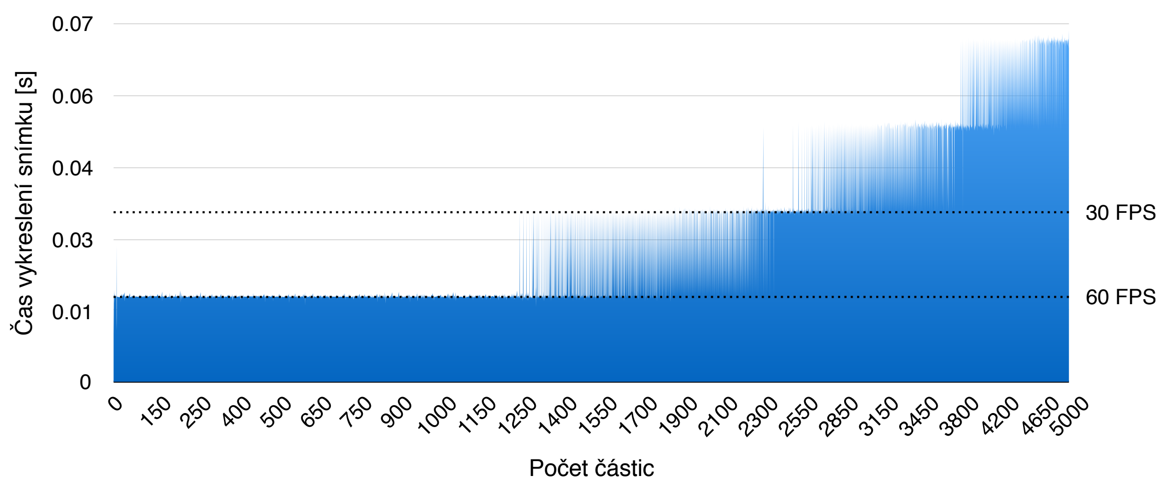
Obrázek 7.2: Závislost času zpracování snímku na typu úlohy pro 100 vykreslení v rámci snímku.

Částicový systém

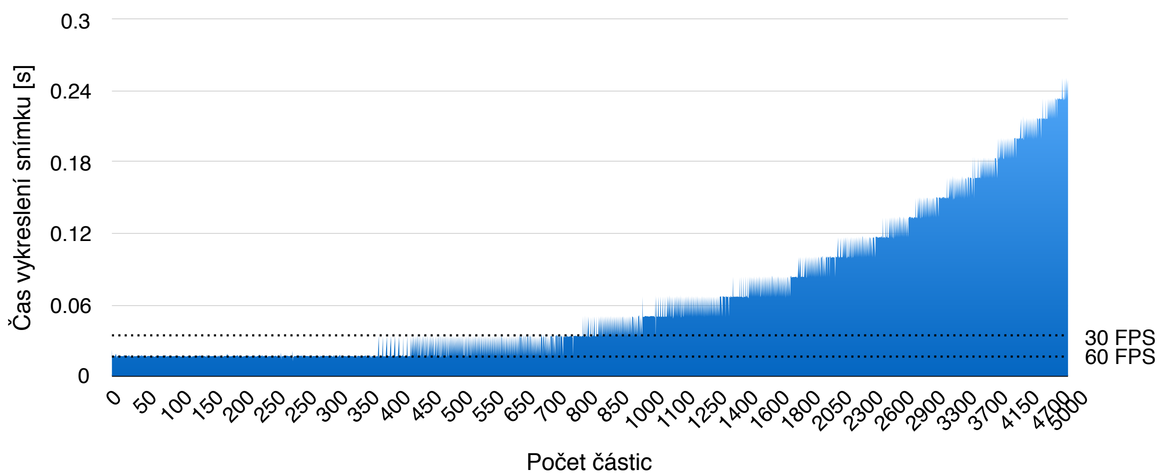
Měření výkonu při vykreslování částicového systému je od prvního experimentu odlišné. V testovací aplikaci je implementován částicový systém, který jako částice používá model koule o 2160 vrcholech.

Při běhu aplikace jsou neustále generovány další částice rychlostí 50 částic za jednotku času, která bude při všech experimentech rovna dvou vteřinám.

Graf na obrázku 7.3 zobrazuje závislost času potřebného k vykreslení snímku na počtu částic v zobrazované scéně. Použitým zařízením je v tomto případě iPhone SE. Při porovnání s grafem 7.4 se také nyní potvrzuje vyšší výkon prvního zařízení. Použitou knihovnou je v obou případech OpenGL ES ve verzi 3.0.

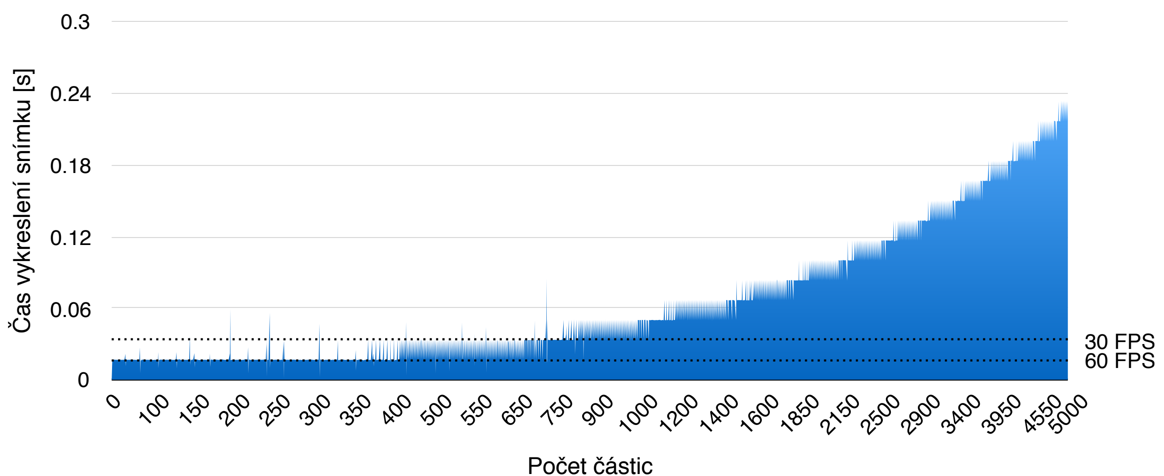


Obrázek 7.3: Závislost času potřebného k vykreslení snímku na počtu částic v zobrazované scéně. (OpenGL ES - iPhone SE) [2]



Obrázek 7.4: Závislost času potřebného k vykreslení snímku na počtu částic v zobrazované scéně. (OpenGL ES - iPad Air 2) [2]

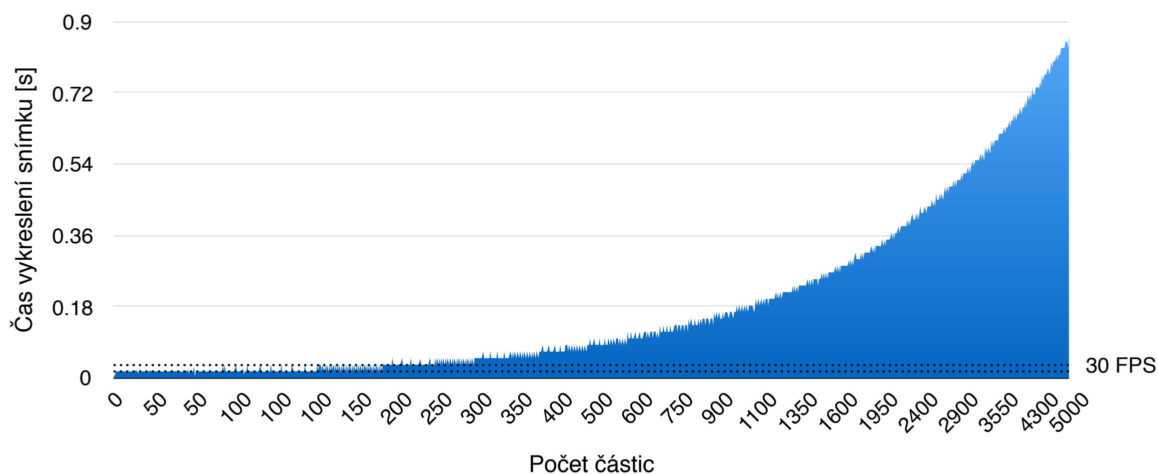
Následující dva grafy zobrazují stejnou závislost s tím rozdílem, že použitou knihovnou je nyní Metal. Z času potřebného k vykreslení jednotlivých snímku již lze rozpoznat, že výkonejší zařízení je použito v případě obrázku 7.5 a iPad je použit v obrázku následujícím 7.6.



Obrázek 7.5: Závislost času potřebného k vykreslení snímku na počtu částic v zobrazované scéně. (Metal - iPhone SE) [2]

Při porovnání stejných zařízení a různých knihoven je jasně patrný výkonostní rozdíl ve prospěch knihovny OpenGL ES. V případě použití zařízení iPhone SE a knihovny OpenGL ES je možné při 30 snímcích za sekundu vykreslovat cca. 2550 částic. Oproti tomu na stejném zařízení lze při použití knihovny Metal dosáhnout pouze asi osmi set částic.

Při použití tabletu (iPad Air 2) je i zde vidět vyšší výkon knihovny OpenGL ES. Čas vykreslení snímku při cílovém množství 5000 částic při použití OpenGL ES je cca. 0.24 vteřiny, resp. $\sim$ 0.86 vteřiny při použití knihovny Metal.



Obrázek 7.6: Závislost času potřebného k vykreslení snímku na počtu částic v zobrazované scéně. (Metal - iPad Air 2) [2]

[2]Osa X u grafů 7.3, 7.4, 7.5, 7.6 není lineární, ale její intervaly jsou závislé na počtu snímků v jednotlivých fázích běhu aplikace.

Kapitola 8

Závěr

Cílem práce bylo navrhnout a implementovat jednoduchou aplikaci s jejíž pomocí lze měřit výkon vykreslování na mobilních zařízeních. Po potvrzení vykreslování po dlaždicích v OpenGL ES kontextu byl za pomoci obou knihoven implementován částicový systém, který tento cíl naplňuje. Implementovaný částicový systém je možné jednoduše modifikovat. Změnou jedné konstanty lze dosáhnout jiného množství generovaných částic. Stejně jednoduše je možné změnit i střední dobu generování částic. Obdobně nenáročná je i záměna modelu použitého pro částici.

Výsledky měření scény na dvou různých zařízeních prokázaly převahu OpenGL ES oproti knihovně Metal. Ačkoli jsem experimentoval s různými parametry systému závěry byly stejné. Například na zařízení iPhone SE bylo možné pomocí OpenGL ES plynule vykreslovat téměř čtyřnásobné množství částic.

V této práci lze nadále pokračovat několika směry. Nabízí se přidání dalších testovacích scén a tedy pokračování v porovnávání výkonu. Ovšem jeden z cílů této práce, konkrétně ten definovaný v úvodní kapitole, byl pomoci vývojářům učinit informovanější rozhodnutí o tom kterou knihovnu zvolit. Ačkoli by se po shlédnutí výsledků experimentů mohlo zdát logické pokračovat v další práci s vítěznou knihovnou OpenGL ES, tak osobně bych rád pokračoval spíše v práci s novější z dvojice testovaných. Motivací skrytou za tímto rozhodnutím je vyšší stupeň integrace do systému iOS a také modernější rozhraní.

V případě, že by tato práce sloužila jako odrazový můstek pro následné pokračování pouze s knihovnou Metal, tak bych považoval za přinejmenším zajímavé pokusit se obohatit částicový systém o vzájemnou interakci jednotlivých částic pomocí obecných výpočtů na grafickém čipu.

Literatura

[1] Apple Inc. *Concurrency Programming Guide* [online]. 2012 Dostupné z: <https://developer.apple.com/library/ios/documentation/General/Conceptual/ConcurrencyProgrammingGuide/Introduction/Introduction.html#//apple_ref/doc/uid/TP40008091>.

[2] Apple Inc. *Cocoa Fundamentals Guide* [online]. 2013 Dostupné z: <https://developer.apple.com/legacy/library/documentation/Cocoa/Conceptual/CocoaFundamentals/CocoaDesignPatterns/CocoaDesignPatterns.html#//apple_ref/doc/uid/TP40002974-CH6-SW1>.

[3] Apple Inc. *Programming with Objective-C* [online]. 2014 Dostupné z: <https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html#//apple_ref/doc/uid/TP40011210-CH1-SW1>.

[4] Apple Inc. *Metal Programming Guide* [online]. 2015 Dostupné z: <https://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/MetalProgrammingGuide/Introduction/Introduction.html>.

[5] Apple Inc. *Metal Shading Language Guide* [online]. 2015 Dostupné z: <https://developer.apple.com/library/ios/documentation/Metal/Reference/MetalShadingLanguageGuide/>.

[6] Apple Inc. *OpenGL ES Programming Guide for iOS* [online]. 2015 Dostupné z: <https://developer.apple.com/library/ios/documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/Introduction/Introduction.html>.

[7] Christ, M.; Moore, W. *Metal* [online]. 2014 Dostupné z: <https://www.objc.io/issues/18-games/metal/>.

[8] Cozzi, P.; Riccio, C. *OpenGL Insights*. Taylor & Francis, 2012. ISBN 9781439893760.

[9] Davis, J. *Understanding OpenGL ES: Multi-thread and multi-window rendering - Imagination Blog* [online]. 2013 Dostupné z: <http://blog.imgtec.com/powervr/understanding-opengl-es-multi-thread-multi-window-rendering>.

[10] Gamma, E.; Helm, R.; Johnson, R.; aj. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994. ISBN 9780321700698.

[11] Ginsburg, D.; Purnomo, B.; Shreiner, D.; aj. *OpenGL ES 3.0 Programming Guide*. OpenGL, Pearson Education, 2014. ISBN 9780133440126.

[12] Hector, T. *Vulkan: Scaling to multiple threads - Imagination Blog* [online]. 2015 Dostupné z: <http://blog.imgtec.com/powervr/vulkan-scaling-to-multiple-threads>.

[13] Imagination Technologies Limited. *A look at the PowerVR graphics architecture: Deferred rendering - Imagination Blog* [online]. 2016 Dostupné z: <http://blog.imgtec.com/powervr/the-dr-in-tbdr-deferred-rendering-in-rogue>.

[14] Imagination Technologies Limited. *A look at the PowerVR graphics architecture: Tile-based rendering - Imagination Blog* [online]. 2016 Dostupné z: <http://blog.imgtec.com/powervr/a-look-at-the-powervr-graphics-architecture-tile-based-rendering>.

[15] Imagination Technologies Limited. *PowerVR Hardware Architecture Overview for Developers* [online]. 2016 Dostupné z: <http://cdn.imgtec.com/sdk-documentation/PowerVR+Hardware.Architecture+Overview+for+Developers.pdf>.

[16] Moore, W. *Metal by Example: High-Performance Graphics and Data-Parallel Programming for IOS*. Warren Moore, 2015. ISBN 9780996312301.

[17] Navik, A. P.; Zaveri, M. A.; Murthy, S. V.; aj. *Emerging Research in Computing, Information, Communication and Applications: ERCICA 2015, Volume 1*. Shetty, R. N. and Prasad, N.H. and Nalini, N. New Delhi: Springer India, 2015. Microbenchmark Based Performance Evaluation of GPU Rendering. Dostupné z: <http://dx.doi.org/10.1007/978-81-322-2550-8_39>. ISBN 9788132225508.

[18] Porcino, N.; Palandri, R.; Roberts, C. *Managing 3D Assets with Model I/O* [online]. 2015 Dostupné z: <https://developer.apple.com/videos/play/wwdc2015/602/>.

[19] Reeves, W. T. : Particle Systems a Technique for Modeling a Class of Fuzzy Objects. *ACM Trans. Graph.*, roč. 2, č. 2, Duben 1983: s 91–108, ISSN 0730-0301.

[20] Rideout, P. *iPhone 3D Programming: Developing Graphical Applications with OpenGL ES*. O'Reilly Media, 2010. ISBN 9781449390624.

[21] Sandmel, J. *Working with Metal – Overview* [online]. 2014 Dostupné z: <http://devstreaming.apple.com/videos/wwdc/2014/603xx33n8igr5n1/603/603_working_with_metal_overview.pdf>.

[22] Shiffman, D. *The Nature of Code*. D. Shiffman, 2012. ISBN 9780985930806.

[23] The Khronos Group Inc. *OpenGL ES 3.1* [online]. 2015 Dostupné z: <https://www.khronos.org/registry/gles/specs/3.1/es_spec_3.1.pdf>.

[24] The Khronos Group Inc. *Vulkan Overview February 2016* [online]. 2016 Dostupné z: <https://www.khronos.org/assets/uploads/developers/library/overview/vulkan-overview.pdf>.

[25] Wyman, C.; Foley, T.; Sellers, G.; aj. An Overview of Next-generation Graphics APIs. 2015.