Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informatiky a kvantitativních metod

Grafická knihovna OpenGL ES na platformě Android

Bakalářská práce

Autor: Michal Kolomazník
Studijní obor: AI3

Vedoucí práce: Ing. Bruno Ježek, Ph.D.

Hradec Králové Srpen 2018

Prohlášení:

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a s použitím uvedené literatury.

……

V Hradci Králové dne 12.8.2018 Michal Kolomazník

Poděkování:

Děkuji vedoucímu bakalářské práce Ing. Brunovi Ježkovi, Ph.D. za vedení práce, užitečné rady a čas věnovaný konzultacím.

Anotace

Tato práce se zabývá vývojem grafických aplikací pro mobilní platformu s operačním systémem Android za pomoci grafické knihovny OpenGL ES a zjištěním technických omezení mobilní platformy. Práce obsahuje informace o potřebných technologiích, problémech, které se při vývoji často objevují a také podrobný postup vedoucí k vytvoření vlastních aplikací. Dále sleduje rozdíly mezi knihovnami OpenGL pro desktop a OpenGL ES pro Android. Součástí práce jsou také postupy použití různých grafických funkcí za použití OpenGL ES na mobilních zařízeních a ukázkové úlohy testující jejich použitelnost. Dále práce obsahuje testy zjištěných omezení na mobilních zařízeních v porovnání s desktopovými systémy a testy výkonosti na obou platformách.

Annotation

Title: Graphics library OpenGL ES on Android platform

This thesis deals with the development of graphical applications for mobile platform with Android operating system using OpenGL ES graphic library and detection of technical limitations of mobile platform. The work contains information about the necessary technologies, the problems that are often encountered during development and detailed steps leading to the creation of your own applications. It also tracks the differences between OpenGL for desktop and OpenGL ES for Android libraries. Part of the thesis are also procedures how to use various graphical functions using OpenGL ES on mobile devices and sample tasks testing their usability. The work also contains tests of detected limitations on mobile devices in comparison to desktop systems and performance tests on both platforms.

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

Vývoj informačních technologií se v současné době ubírá směrem k minimalizaci. Trendem jsou chytré telefony a další mobilní zařízení s výpočetní silou mnohokrát větší než počítač, který dostal prvního člověka na měsíc. Nárůst výkonu umožnil provádění výpočetně náročnějších operací, které jsou dnes nutné pro chod většiny grafických aplikací, ať už se jedná o hru s realistickou grafikou nebo zobrazení rozsáhlých map či plánů budov. To vše je dnes možné s využitím nejnovějších technologií a postupů. Většina mobilní zařízení dnes používá jeden ze tří operačních systémů a to Android, iOS nebo Windows Phone. Největší zastoupení mezi zařízeními má právě systém Android, proto je tvorba aplikací soustředěna především zde.

Pro tvorbu grafických aplikací jsou používány speciální nástroje a knihovny. Jedním z takovýchto nástrojů je DirectX od společnosti Microsoft, který obsahuje mnoho nástrojů související nejen s počítačovou grafikou. Nevýhodou DirectX však je jeho svázanost se systémem Microsoft Windows. Tento problém řeší multiplatformní nástroj OpenGL, který je možné využít na téměř jakékoli platformě s použitím libovolného programovacího jazyka. Existují navíc nástroje velmi podobné OpenGL, které je možné využít nejen pro desktopovou platformu ale také pro weby nebo mobilní aplikace. Další možností je Vulkan, zamýšlený jako nástupce OpenGL a označovaný jako grafické API nové generace. Vulkan však podporují pouze novější systémy. Největší počet mobilních zařízení proto používá OpenGL.

Práce se zabývá možnostmi návrhu mobilních aplikací využívajících grafickou knihovnou OpenGL s hardwarovou akcelerací. Je členěna do pěti kapitol. První kapitola popisuje technologie a postupy využívané při tvorbě grafických aplikací. Další kapitoly shrnují specifikace knihovny OpenGL ES určené pro mobilní zařízení a srovnávají ji s knihovnou OpenGL navrženou pro desktopové systémy. Dále je rozebrán postup tvorby grafické aplikace pro mobilní zařízení. Závěrem jsou navrženy a provedeny testy možností grafické knihovny OpenGL ES pomocí ukázkových úloh a testy omezení a výkonů běžných zařízení.

2 Technologie

Příchod minimalizace technologií přinesl výrazné zmenšení zařízení, což umožňuje jejich snadné přenášení. Vznikla tak takzvaná „mobilní“ zařízení. Jejich menší velikost, přenositelnost a manipulativnost jim také umožňuje využívat různé typy senzorů, které by u desktopových počítačů neměly využití. Druhou rychle rozvíjející se technologií mobilních zařízení je podpora zobrazování a možnosti grafického subsystému. V následujících kapitolách bude popsán vývoj i aktuální stav mobilních technologií.

2.1 Mobilní zařízení

Mobilních zařízení existuje celá řada, ale tvorba aplikací je hlavně soustředěna na takzvaných „chytrých zařízeních“ v angličtině „smart devices“. „*Chytré zařízení je elektronické zařízení (mobilní telefon, tablet) určené pro přístup k funkcím umístěným lokálně na zařízení nebo vzdáleně na serveru. Zařízení bývají propojena s dalšími zařízeními pomocí bezdrátových sítí. Jejich hlavními charakteristikami je mobilita (jsou navržené pro lehké přenášení lidmi), dynamické zpřístupňování služeb (mohou se zaměřovat na nejbližší dostupné poskytovatele služeb) a přerušovaný přístup k prostředkům (nelze se spoléhat na připojení k síti).*“ [1] Zařízení tvoří tři části procesor s operačním systémem (OS), senzory a grafický podsystém. Operační systém se stará mimo jiné o spouštění programů na mobilním zařízeních a přidělení systémových prostředků běžícím úlohám. Tyto úlohy poté mohou využívat data ze senzorů zařízení, které slouží pro snímání stavu okolního prostředí. Úlohy pak pomocí grafického podsystému vykreslí uživatelské rozhraní na obrazovku zařízení.

Vývoj chytrých zařízení umožnil vývojářům vizuálních aplikací zaměřit se i na náročnější grafické vizualizace jako je realistické osvětlení, odlesky a různé optické jevy. Začaly se objevovat specializované GPU (Graphics Processing Unit) čipy, určené pro urychlení grafických výpočtů, které umožnili tvořit aplikace s vysokou výpočetní náročností jejich operací. Parametry GPU se však liší v závislosti na jejich výrobci. Kvůli rozdílům mezi grafickými čipy byla vyvinuta grafická knihovna OpenGL ES, která standardizuje rozhraní pro komunikaci s těmito

čipy. Jedná se o verzi vycházející z OpenGL knihovny určené pro desktopové platformy a upravené tak, že obsahuje jen nezbytné funkce nutné pro mobilní zařízení.

Součástí mobilních zařízení jsou také senzory, které rozšiřují možnosti moderních zařízení o další funkce. Tyto senzory mohou být rozděleny do tří kategorií. První z nich jsou senzory pohybu, které snímají zrychlení a rotaci zařízení. Patří sem akcelerometry (senzory zrychlení) a gyroskopy (snímají naklonění). Druhou kategorií jsou snímače okolního prostředí a patří sem světelná, tlaková (barometry) a teplotní čidla (termometry). Poslední kategorií jsou snímače polohy a zahrnuje magnetické senzory (určení polohy na základě geomagnetického pole) a proximity senzory používané k detekci blízkých předmětů (přiložení ucha k telefonu).

2.1.1 Android

Systém Android je dnes nejrozšířenější operační systém pro mobilní zařízení, využívá jej 85 % smartphonů prodaných v roce 2017 [2]. Druhým operačním systémem byl ve stejném roce iOS od společnosti Apple se 14,7 % [2]. Oba systémy tak pokrývají přibližně 99,7 % procent všech prodaných smartphonů, proto je vliv ostatních v historii využívaných operačních systémů, jako například Windows Phone, dnes již nepodstatný. V listopadu 2017 systém Android využívalo 75,9 % (2,3 miliard) aktivních zařízení z celkového počtu 3,1 miliard [3].

Systém Android je založen na Linuxovém jádře a vyvíjen společností Google. Systém je navržen pro celou škálu dotykových zařízení včetně tabletů, smartphonů, smartwatch a mnoha dalších. Android od verze 1.6 začal podporovat zařízení všech tvarů a velikostí (podpora více rozlišení a poměrů stran) a navíc s verzí 3.0 nabízí lepší uspořádání prvků pro větší obrazovky tabletů.

Android nabízí prostřednictvím aplikací mnoho funkcí pro snadné ovládání zařízení a umožňuje tak mimo jiné snadnou práci s dokumenty nebo bleskovou komunikaci pomocí bezdrátových technologií. Vývoj systému postupně umožnil využití funkcí jako převod hlasu na text (verze 2.1), ovládání hlasem (verze 2.2) nebo zobrazení virtuální reality (verze 7.1). Verze systému Android jsou uvedeny v tabulce (Tab. 1) spolu s jejich krycími jmény a podporovanou verzí OpenGL ES.

V současné době (Leden 2018) drtivá většina mobilních zařízení používá minimálně verzi Android 2.3 [4]. Nejnovější verzí je Android 8.0 přinášející funkci Picture-in-picture, která umožňuje uživateli nechat aplikaci ve zmenšeném okně, zatímco využívá aplikaci jinou. Tato funkce se využívá zejména pro přehrávání videa.

Tab. 1: Verze operačního systému Android zdroj vlastní vychází z [[5], [6], [7], [8], [9]]

Verze	Krycí jméno	API	OpenGL ES
1.0		1	1.0, 1.1
1.1		2	
1.5	Cupcake	3	
1.6	Donut	4	
2.0	Eclair	5-7	
2.2	Froyo	8	2.0
2.3	Gingerbread	9, 10	
3.0	Honeycomb	11-13	
4.0	Ice Cream Sandwich	14, 15	
4.1	Jelly Bean	16-18	3.0 (API 18)
4.4	KitKat	19, 20	
5.0	Lollipop	21, 22	3.1
6.0	Marshmallow	23	
7.0	Nougat	24, 25	3.2
8.0	Oreo	26, 27	

2.1.2 Omezení

Lehká přenositelnost mobilních zařízení je vyvážena několika omezeními. Na rozdíl od klasických počítačů mají mobilní zařízení výrazně menší displeje, což přináší mnohé problémy z hlediska interakce s aplikací. Většina nových zařízení nemá k dispozici fyzickou klávesnici, proto musí být veškeré ovládání pomocí dotyků na dotykové obrazovce. Je důležité zvolit dostatečně velké ovládací prvky a rozmístit je dost daleko od sebe, aby nedocházelo ke stisknutí jiného tlačítka, než uživatel zamýšlel. Dále je vhodné omezit textové vstupy, protože psaní na dotykovém displeji bývá velmi zdlouhavé.

Kvůli menší velikosti zařízení je omezen také hardware, čímž je výrazně snížen maximální výkon zařízení. Je potřeba mnohem větší optimalizace než na desktopech a je k dispozici mnohem menší velikost paměti. Většina dat by tak měla

být uložena pomocí co nejmenších datových typů. Je také vhodné omezit počet výpočtů a operací, které aplikace vykonává a šetřit tím baterii. Maximální velikost aplikace pro Android platformu (Google Play) je 50 MB, pro novější verze Android 100 MB. S tímto omezení je potřeba počítat při vývoji aplikací a použití jakýchkoliv souborů s texturami nebo modely objektů. Google Play navíc nabízí možnost přidat k aplikaci dva rozšiřující soubory s maximální velikostí 2 GB, které mohou být jednoduše staženy spolu s aplikací. Tyto limitace lze však obejít stažením souborů z jakékoliv URL adresy přímo aplikací nebo umístěním souborů na zařízení uživatelem.

Konkrétní funkce, které jednotlivá zařízení poskytují, samozřejmě závisí na nainstalované verzi systému, dostupných knihovnách a rozšířeních těchto knihoven. Aplikace využívající senzorů budou navíc vyžadovat, aby zařízení využívaný senzor vůbec obsahovalo.

2.2 OpenGL

OpenGL je grafické rozhraní (API) pro tvorbu 2D a 3D aplikací počítačové grafiky dostupná na 3 nejdominantnějších desktopových systémech Windows, Linux a macOS [10]. Umožňuje vykreslování základních geometrických elementů, jako jsou jednotlivé body, úsečky, trojúhelníky nebo jejich struktury. Spojením více základních tvarů lze vytvořit a následně zobrazit povrchový model 3D objektu. Jednotlivé objekty jsou reprezentovány pomocí seznamu vrcholů (vertex buffer) a informací o tom, jak tyto vrcholy vytvářejí elementy (index buffer). Vedle souřadnice pozice vrcholy obsahují většinou i další atributy jako jsou informace o barvě, normálách a texturových souřadnicích.

V 80. a 90. letech 20. století byl velký problém vývoje software, který umožňoval práci s různými grafickými zařízeními. Vznikl standard OpenGL, snažící se přinést jednotný jazyk pro komunikaci s grafickým hardwarem. Kontrolu nad standardem OpenGL získala skupina Khronos Group, která pravidelně vydává nové verze standardu [10]. Členové skupiny jsou výrobci grafických karet a operačních systémů a v poslední době i mobilních zařízení jako například společnosti Intel Corporation, AMD, NVIDIA, Apple Inc., Google, Microsoft Corporation a mnoho dalších [11].

Zobrazovací systém v OpenGL je navržen tak, aby umožnil hardwarovou implementaci a akceleraci na jednotce grafického procesoru (Graphic Processing Unit – GPU). Výrobci grafických procesorů tvoří vlastní implementace OpenGL ovladačů (driverů), které překládají OpenGL příkazy na příkazy jednotlivých GPU. Vykreslovací příkaz je zpracován grafickým systémem v zobrazovacím řetězci (rendering pipeline). Knihovna OpenGL umožňuje vykreslování pomocí pevného nebo programovatelného zobrazovacího řetězce. Více informací o zobrazovacím řetězci je uvedeno v následující kapitole.

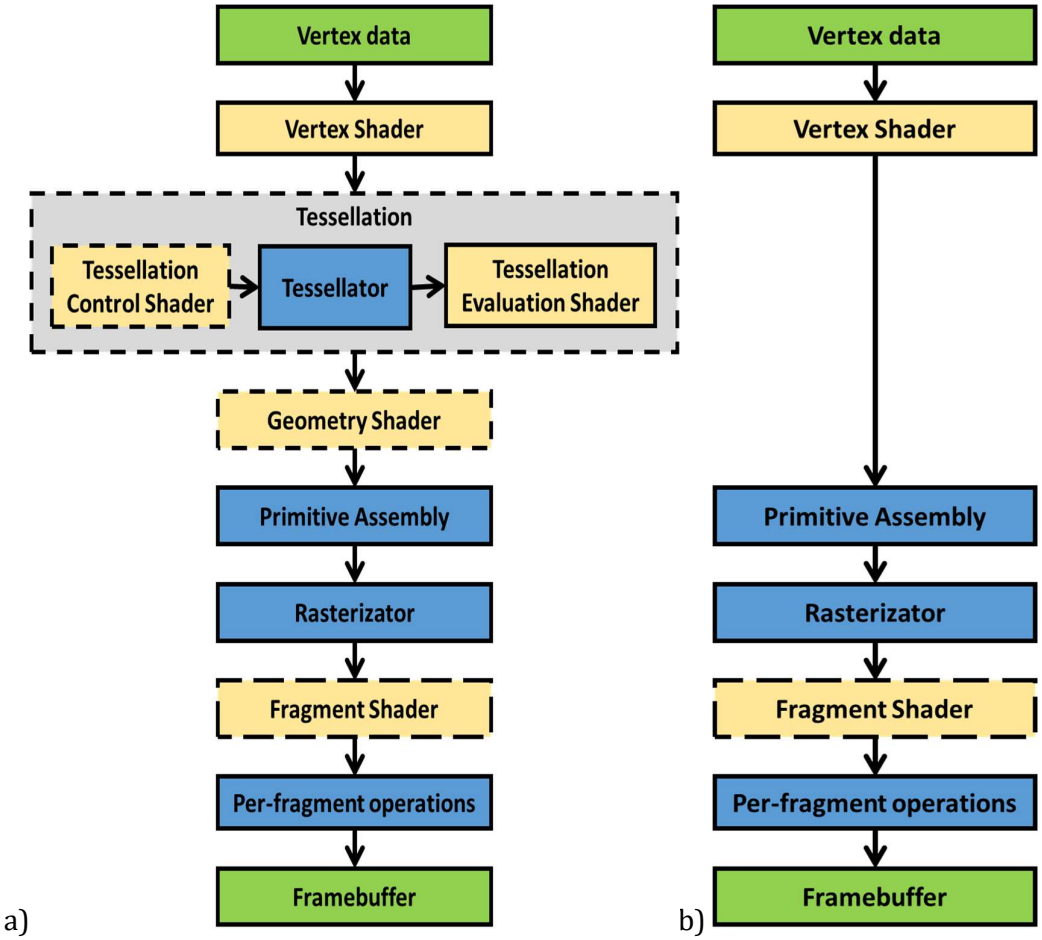
2.2.1 Zobrazovací řetězec v OpenGL

V začátcích OpenGL byl k vykreslování využíván pouze pevný zobrazovací řetězec (fixed pipeline). Jedná se o sekvenci kroků, které jsou postupně vykonány před vykreslením výsledného obrazu, kde na každém kroku je vykonán nějaký výpočet a výsledek je odeslán dalšímu kroku ve vykreslovacím řetězci k dalšímu zpracování. V pevném řetězci byli výpočty v jednotlivých krocích pevně dané vestavěnými hardwarově implementovanými funkcemi a bylo umožněno do řetězce pouze vkládat vlastní data a lehce ovlivňovat konfiguraci řetězce. Části pevného řetězce byly později nahrazeny programovatelnými jednotkami, avšak sekvence kroků zůstala víceméně stejná. Z OpenGL ES byl pevný řetězec zcela odstraněn a v OpenGL zůstal pouze jako součást kompatibility módu. Programovatelný řetězec umožňuje pozměnit výpočty jednotlivých kroků pomocí speciálních programů – shaderů.

Prvním krokem v řetězci OpenGL je příprava dat vrcholů, které mají být vykresleny a odeslání dat dalšímu kroku. Poté dochází ke zpracování dat vrcholů, které mohou být násobeny transformačními maticemi. Transformované vrcholy jsou složeny do sekvence geometrických primitiv (trojúhelník, quad, pás, vějíř). Následně může být provedeno dělení na menší primitiva (tzv. teselace), případně generování nových primitiv a opětovné sestavení. V dalším kroku probíhá dodatečné zpracování zahrnující například ořezání, dehomogenizaci a transformaci do prostoru okna. Transformovaná data jsou následně rozdělena na fragmenty pomocí rasterizace. Takto vytvořené fragmenty odpovídají pixelům umístěným v rastru průmětny a jsou zpracovávány ve fragment shaderu. Kromě pozice XY a

hloubky Z mohou být pro každý fragment připojeny i informace o barvě, textuře případně dalších zvolených atributech. Každý pixel tak může být pře vykreslením ještě dále upraven na základě výpočtů osvětlení, průhlednosti nebo mapování textury. Před samotným vykreslením je ještě provedeno několik testů včetně depth-testu, stencil testu, scissor testu, kterými je určeno, zda je pixel viditelný vzhledem k umístění kamery a ostatních objektů. Nakonec je každý pixel vykreslen.

Přehled kroků je vidět na obrázku (Obr. 1a). Oranžově podbarvené kroky je možné upravit pomocí shaderů při použití programovatelné pipeline. Čárkovanou čarou ohraničené kroky jsou nepovinné a je možné je za určitých podmínek přeskočit. Zeleně jsou označena data.



Obr. 1: Zobrazovací řetězec v a) OpenGL 4.0 a b) OpenGL ES 3.0

Řetězec v OpenGL ES 3.0 (Obr. 1b) je dost podobný tomu v OpenGL s tím rozdílem že chybějí Tessellation a Geometry shadery. V programovatelném OpenGL ES řetězci je možné upravit výpočty Vertex a Fragment shaderu.

V nejnovější verzi OpenGL ES 3.2 byli přidány i chybějící geometry a teselační shadery čímž bylo dosaženo velkého přiblížení k funkcionalitě OpenGL.

2.2.2 Shadery

Shaderový procesor představuje programovatelnou jednotku GPU, na které se vykonává shaderový program jako část programovatelného vykreslovacího řetězce. Shader je zkrácené označení jak pro vlastní hardwarovou jednotku, tak pro prováděný program. Existuje několik typů shaderů, každý určený pro specifický krok v rendrovacím řetězci. V OpenGL ES 2.0 jsou to Vertex shader a Fragment shader. OpenGL navíc obsahuje dva Tessellation shadery, Compute a Geometry shader.

Shadery se programují pomocí jazyka, který řídí chování programu a výpočet výsledného obrazu. Aplikace využívající knihovnu OpenGL používají OpenGL Shading Language (GLSL). Zdrojové shaderové programy musí být před jejich použitím zkompilovány pomocí speciálních příkazů OpenGL.

Android studio soubory s touto příponou automaticky rozpozná a nabídne instalaci pluginu se syntaxí shaderů. Samotný kód využívá syntaxi programovacího jazyka C. Každý ze shaderů je specializován pro jiné výpočty v zobrazovacím řetězci.

Vertex shader

Vertex shader je první shader v zobrazovacím řetězci. Po zavolání jakéhokoli vykreslovacího příkazu v OpenGL zpracovává jednotlivé vrcholy a může být využit například k transformaci vrcholů, k výpočtu per-vertex osvětlení, nebo k přípravě dat pro pozdější zpracování. Před samotným použitím vertex shaderu musí uživatel nejprve přiřadit shaderu atributy. Nejčastějšími používanými vstupy jsou souřadnice, normály a texturové souřadnice vrcholů. Výstupy z tohoto shaderu jsou posílány dalšímu kroku v řetězci. Vzhledem k tomu že některé kroky jsou nepovinné, může to být v tomto pořadí Tessellation control shader, Tessellation evaluation shader, Geometry shader nebo krok Vertex post processing. Výstupem může být barva nebo souřadnice do textury vrcholu. Dále existuje několik vestavěných výstupů jmenovitě například povinná výstupní hodnota určující pozici transformovaného vrcholu (`gl_Position`).

Tessellation control shader

Tato část vykreslovacího řetězce je nepovinná a nemusí být vůbec vykonána. Tessellation control shader (TCS) rozhoduje jaká teselace bude aplikována na jednotlivá geometrická primitiva. Teselace je proces ve kterém jsou geometrická primitiva rozdělena na menší primitiva přidáním vrcholů do každé části. Vstupy TCS jsou data vrcholů z vertex shaderu a informace o velikosti primitiv, která jsou vykreslována. Určitý počet vrcholů je pak spojen do pole o velikosti určené vstupním primitivem. Každá tato část je rozdělena podle požadované úrovně teselace přidáním nebo odebráním vrcholů. Proto se velikost výstupního pole bude lišit od vstupního. Výstupem z TCS jsou pole vrcholů tvořící jednotlivé části, která jsou předána Tessellation evaluation shaderu.

Tessellation evaluation shader

Dalším nepovinným krokem v řetězci je Tessellation evaluation shader (TES). Pokud je tato část aktivní bude navíc vykonán pevně daný krok teselace generování primitiv. V tomto kroku jsou geometrická primitiva vytvořená v teselátoru na základě úrovně teselace vypočtené buď v TCS nebo defaultně zvolené, pokud TCS není aktivní. Pokud není aktivní TES neproběhne žádná část teselace. V samotném TES probíhá výpočet nových hodnot v jednotlivých vrcholech (pozice, normály, souřadnice do textury) nově vytvořených primitivů z předchozích kroků teselace pomocí interpolace. Vstupy TES jsou data o vrcholech buď přímo z vertex shaderu nebo z TSC zpracovaná. Výstupem jsou data vypočítaná data vrcholů.

Geometry shader

Po teselaci (pokud je aktivní) následuje Geometry shader. Jedná se o další nepovinný krok vykreslovacího řetězce. V Geometry shaderu (GS) mohou být jednotlivá primitiva, skládající se z určitého počtu vrcholů určitého typu, transformována na jiný počet primitiv nebo jiný typ primitiv. Například jeden bod na vstupu (points) může být transformován na tři trojúhelníky (triangle_strip). Toho se využívá při práci s částicovými systémy. GS může být také využit pro vypočtení dalších informací o vrcholech. Vstupem do GS je jedno geometrické primitivum

s daty všech jeho vrcholů z vertex shaderu nebo data primitiva určeného jako výsledek teselace (TES). Výstupem GS je 0 nebo více primitiv, která mohou být jiného typu než vstupní primitivum. Výstupní primitiva však musí všechna mít jeden stejný typ.

Fragment shader

Posledním shaderem v řetězci je Fragment shader, který zpracovává fragmenty pixelů geometrických primitiv vytvořené v rasterizačním kroku řetězce. Fragment shader často využívá hodnoty interpolované pro určitý fragment a používá se k výpočtu per-fragment osvětlení a výsledné barvy pixelů fragmentu. Vstupem shaderu jsou tedy údaje o vrcholech tvořících fragment, včetně barvy a texturové souřadnice. Na pixely je následně aplikována textura a na základě osvětlení a barvy je vypočtena výsledná hodnota barvy pixelu. Výstupem je tedy barva pixelů ve fragmentu (ve starších verzích shaderů `gl_FragColor`, v novější je možno deklarovat vlastní proměnnou). Vestavěným výstupem tohoto shaderu je proměnná `gl_FragDepth` obsahující hodnotu hloubky fragmentu. Krok zpracování ve fragment shaderu je za určitý podmínek možné přeskočit zkrácením zobrazovacího řetězce při použití tzv. transform feedback.

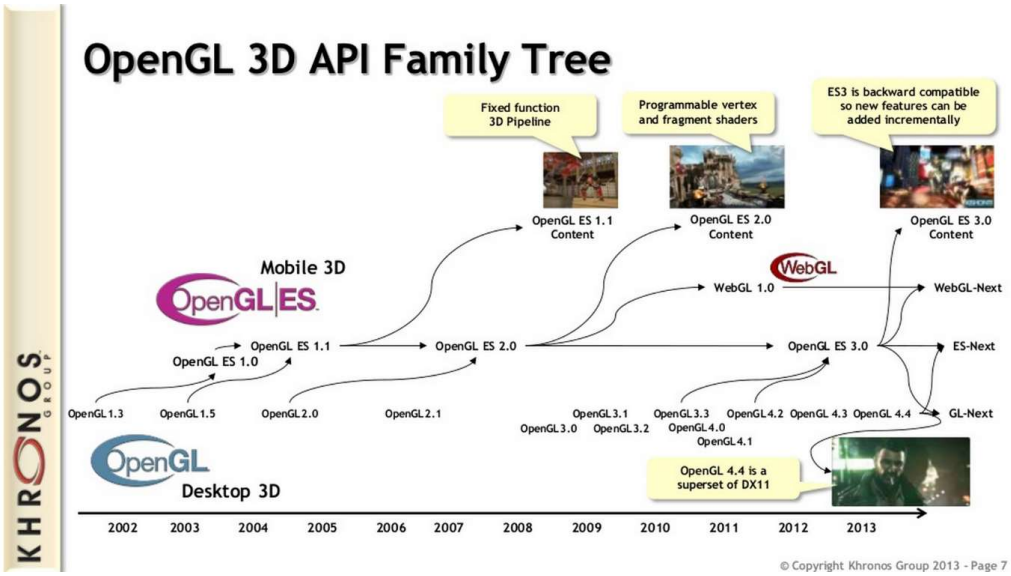
Compute shadery

Zvláštním případem shaderu jsou Compute shadery, které nejsou součástí vykreslovacího řetězce. Jsou použity pro pomocné výpočty, které je možno vykonat mimo vykreslovací řetězec, umožňují tak zrychlit libovolný výpočet pomocí vícevláknového paralelního zpracování. Vstupy ani výstupy z těchto shaderů nejsou spojeny s dalšími kroky řetězce, ale s buffery přímo v grafické paměti.

2.2.3 Typy OpenGL

OpenGL je navrženo speciálně pro desktopové systémy pro tvorbu grafických aplikací, jiné platformy využívají různé vývojové větve této knihovny. Nejvýznamnějšími větvemi jsou WebGL pro webové aplikace a OpenGL ES pro aplikace na mobilních zařízeních. Za zmínku také stojí knihovna OpenAL, používaná k tvorbě audio aplikací a velmi podobná OpenGL. Existuje mnoho podobných

knihoven pro různé typy aplikací, například knihovna OpenVG slouží pro práci s 2D vektorovou grafikou.



Obr. 2: Vývojové větve OpenGL [12]

OpenGL ES

Knihovna OpenGL ES (OpenGL for Embedded Systems) je vývojovou větví knihovny OpenGL pro mobilní platformu, kde je využívána pro tvorbu grafických aplikací. První verze OpenGL ES byla napsaná s podobnou funkcionalitou jako OpenGL 1.3. Podobně jako OpenGL bylo cílem této knihovny vytvořit více platformní API pro sjednocenou komunikaci s grafickým hardwarem různých mobilních zařízení.

U této platformy jsou problémem menší paměťové možnosti, proto knihovna obsahuje pouze funkce klíčové pro její fungování. Knihovna je dostupná na obou vedoucích operačních systémech mobilních zařízení Android a iOS. OpenGL ES od verze 2.0 obsahuje programovatelný zobrazovací řetězec. OpenGL ES je díky těmto ořezáním rychlejší oproti desktopové OpenGL [14]

WebGL

WebGL je v podstatě JavaScriptová API založená na specifikaci OpenGL ES 2.0. Používá se pro zobrazování 3D grafiky ve webových prohlížečích. Záměrem při tvorbě WebGL bylo vytvořit platformě nezávislý standard, použitelný

bez nutnosti instalace plug-inu do prohlížeče. Dnes je použití WebGL podporováno nejnovějšími verzemi všech známých moderních prohlížečů jako jsou Google Chrome, Mozzila Firefox, Opera, Safari nebo Microsoft Edge. Příkladem aplikací s využití WebGL jsou Google Maps (interaktivní mapa) nebo Google Body (interaktivní zobrazení lidského těla).

2.2.4 Vývoj knihovny OpenGL

Pro grafickou aplikaci je důležitá zejména verze knihovny OpenGL ES podporovaná operačním systémem. Není ovšem garantovaná přítomnost nejnovější verze této knihovny. Například pokud má zařízení nainstalovaný Android verze 6.0, nemusí to nutně znamenat, že je k dispozici OpenGL ES 3.1, proto je nutné vždy programově kontrolovat dostupnou verzi knihovny.

Vývoj aplikací využívajících OpenGL se výrazně liší v závislosti na verzi této knihovny, pro kterou je aplikace vytvořena. Také se liší verze shaderů GLSL, které tyto knihovny využívají. Je důležité zmínit, že pro OpenGL ES existují oddělené verze někdy označované jako GLSL ES (jedná se o GLSL 100 a 300).

OpenGL

První verze OpenGL obsahovaly pouze pevný zobrazovací řetězec a neumožňovali měnit průběh jednotlivých výpočtů. Od verze 2.0 však OpenGL uvedlo možnost úpravy výpočtů pomocí specializovaného programovacího jazyka shaderů (GLSL) a zavedením programovatelného zobrazovacího řetězce. V prvních verzích se jednalo o vertex a fragment shadery. Programovatelnost shaderů ale neznamená možnost úpravy pořadí jednotlivý kroků v řetězci, ale pouze změnu jejich průběhu. V této verzi (2.0) bylo také nově možné využívat textury jiných velikostí než násobky dvou (u starších verzí nutnost). S novými verzemi OpenGL vycházeli i nové verze jazyka GLSL (Tab. 2). Od verze 3.1 byla odstraněna funkcionalita související s pevným zobrazovacím řetězcem.

Dalším důležitým krokem bylo představení Geometry shaderů ve verzi 3.2. Postupně byli přidávány další shadery, každý pro úpravu specifického kroku řetězce. Ve verzi 4.0 byli přidány oba Tessellation shadery a jako další byl přidán Compute shader ve verzi 4.3.

Tab. 2: Vývoj knihovny OpenGL [13]

GLSL	OpenGL	Poznámky
110	2.0	Jiné textury než pouze násobky dvou
120	2.1	
130	3.0	
140	3.1	Odstraněna funkcionalita pevné pipeline
150	3.2	Přidání Geometry shaderů
330	3.3	Sjednoceno číslování verzí OpenGL a GLSL Unifikované shadery
400	4.0	Přidání teselátoru a obou teselačních shaderů
410	4.1	
420	4.2	
430	4.3	Přidání Compute shaderu
440	4.4	
450	4.5	
460	4.6	

OpenGL ES

První verze OpenGL ES využívaly pevný zobrazovací řetězec, který byl později nahrazen programovatelným řetězcem v ES 2.0 (dnes nejrozšířenější verze – viz Tab. 3). Od verze ES 2.0 je tedy možné využívat vertex a fragment shadery napsané pomocí GLSL. Podobně jako v OpenGL byla většina funkcí souvisejících s pevným řetězcem ve verzi ES 2.0 knihovny odstraněna.

Tab. 3: Rozšířenost knihovny OpenGL ES [4]

OpenGL ES	Využití v zařízeních
2.0	37.0 %
3.0	45.4 %
3.1	17.7 %

OpenGL ES 3.0 přineslo plnou podporu textur jiných velikostí než násobky dvěma a také transform feedback. Verze 3.1 navíc přivedla Compute shader i na mobilní platformu. Další verze OpenGL ES spolu s korespondujícími verzemi GLSL a nutnou verzí Android je uvedena v tabulce (Tab. 4). Doposud poslední verze 3.2 přinesla i chybějící Geometry a Tessellation shadery.

Tab. 4: Vývoj knihovny OpenGL ES

GLSL	OpenGL ES	Poznámky	Android
100	2.0	Programovatelné shadery (vertex a fragment)	2.2 (API 8)
300	3.0	Sjednoceno číslování verzí OpenGL ES a GLSL Transform feedback Jiné textury než pouze násobky dvou	4.3 (API 18)
310	3.1	Přidání Compute shaderu	5.0 (API 21)
320	3.2	Přidání Geometry a obou Teselačních shaderů	7.0 (API 24)

3 Porovnání OpenGL ES a OpenGL

Přestože oba standardy OpenGL a OpenGL ES mají mnoho společného, využívání grafického subsystému na mobilních zařízení přináší řadu specifik. Pro práci s jakoukoliv verzí knihovny OpenGL ES je vhodné znát její specifikace. Veškeré specifikace jsou zveřejněny na stránkách skupiny Khronos: https://www.khronos.org/. Tato kapitola bude zaměřena hlavně na specifikace verzí 2.0 knihovny OpenGL ES a rozdíly mezi možnostmi této knihovny a knihovny OpenGL.

3.1 Kompatibilita

Specifikace jednotlivých verzí OpenGL ES bývají založeny na specifikacích OpenGL. Většinou jsou v OpenGL ES některé OpenGL funkce odebrány nebo naopak přidány jejich vhodnější verze pro mobilní platformu. Aplikace napsané pro OpenGL ES mohou být snadno přenositelné na OpenGL, opačný přesun ale nemusí být úspěšný. Například specifikace OpenGL ES 1.1 byla založena na specifikaci OpenGL 1.5 a OpenGL ES 2.0 je na úrovni OpenGL 2.0. (Tab. 5).

Pozdější verze desktopové knihovny OpenGL navíc obsahují plnou kompatibilitu s některými verzemi OpenGL ES pro zjednodušení vytváření desktopových verzí aplikacích původně napsaných pro použití s OpenGL ES. OpenGL 4.3 poskytuje kompatibilitu s OpenGL ES 3.0 a s OpenGL 4.5 byla přidána kompatibilita s OpenGL ES 3.1. Nejnovější verze OpenGL ES zatím není plně kompatibilní s desktopovou verzí.

Tab. 5: Kompatibilní verze OpenGL a OpenGL ES [15]

OpenGL ES	Podkladová OpenGL	OpenGL kompatibilita
1.0	1.3	
1.1	1.5	
2.0	2.0	4.1 - plná kompatibilita
3.0		4.3 - plná kompatibilita
3.1		4.5 - plná kompatibilita
3.2		

3.2 Paměťové nároky

Povaha mobilních zařízení samozřejmě znamená menší množství dostupné paměti. Proto je v OpenGL ES vhodné použití nejmenší možnou velikost datových atributů s ohledem na potřebnou přesnost. Například pro ukládání hodnot souřadnic v textuře většinou stačí *GL_UNSIGNED_SHORT* a pro vrcholy je možné použít *GL_OES_vertex_half_float*.

Jednou z dalších používaných praktik je použití bufferů pro ukládání dat vrcholů. Právě díky bufferům jsou všechny hodnoty uloženy na jednom místě v paměti a ušetří se tak čas potřebný k nalezení jednotlivých hodnot. Díky široké škále mobilních zařízení, která mohou aplikaci spouštět, je také nutné přizpůsobit přesnost jednotlivých proměnných datových typů `int` nebo `float` v shaderech podle možností každého z nich. Toto je v OpenGL ES docíleno pomocí kvalifikátorů přesnosti, které umožňují rozdělit zařízení do třech kategorií na vysokou, střední a nízkou přesnost (`highp`, `mediump`, `lowp`). Pomocí větvení je tak možné vybrat požadovanou přesnost podle informací zjištěných ze zařízení. Tyto kvalifikátory lze použít také v OpenGL, avšak nemají tu žádné využití.

V OpenGL ES také není nutná funkce `dispose` – V OpenGL používaná pro uvolnění paměti shaderů nebo textur. V OpenGL ES se o vše stará třída `GLSurfaceView` (uvolnění paměti shaderů, textur). Je však vhodné uvolňovat paměť dříve – paměť mobilních zařízení je omezená a není vhodné ji zabírat objekty, které už nejsou nadále potřeba.

3.3 Shadery

Knihovna OpenGL ES využívá vlastní verze jazyka shaderů (GLSL) oddělené od OpenGL verzí. Hlavním milníkem důležitým při tvorbě shaderů pro OpenGL ES byl přechod z verze ES 2.0 (GLSL 100) na verzi ES 3.0 (GLSL 300). Tato novější verze totiž změnila označení shaderových proměnných. Vstupy vertex shaderu už nadále nejsou označeny `attribute`, protože všechny vstupy shaderů jsou nově značeny `in`. Naopak pro všechny výstupy shaderů vzniklo nové označení `out`. Označení `varying`, dříve výstup vertex shaderu/vstup fragment shaderu, bylo nahrazeno značením `out` ve vertex shaderu a značením `in` ve fragment shaderu. Podobná změna nastala také

na desktopech v OpenGL 3.1 (GLSL 140), kdy byla označení `varying` a `attribute` odstraněna i z této knihovny.

Další změnou bylo například nahrazení funkcí pro práci s texturami `texture2D()`, `textureCube()`, `texture2D()` apod. obecnější verzí této funkce `texture()` v OpenGL ES 3.0 (GLSL 300). V desktopové verzi jsou kvůli zpětné kompatibilitě od verze OpenGL 3.3 dostupné obě verze této funkce.

3.4 Funkční omezení

Mnoho funkcí běžně používaných na desktopovém OpenGL není součástí OpenGL ES 2.0, některé jsou dostupné pomocí rozšíření. Přístup k rozšířením však pomocí jazyka java není jednoduchý a je nutné použít přemostění do nativního kódu. Třída GLES20, která obsahuje veškerou ES 2.0 funkcionalitu, obsahuje pouze některá rozšíření. Zbylá rozšíření musí být dynamicky namapována použitím jazyka C. Není například možné používat index buffery hodnot o velikosti jiné než unsigned byte nebo short. Toto omezení lze zmírnit použitím OpenGL ES rozšíření `OES_element_index_uint extension`, které přidává podporu unsigned int. OpenGL ES neobsahuje definici `GL_QUAD` pro reprezentaci čtverců.

Dalším větším omezením je podpora float textur, kterou lze získat pouze použitím rozšíření `OES_texture_float`. Jádro OpenGL ES 2.0 (ES 2.0 bez rozšíření) dále neobsahuje podporu 3D textur (pouze pomocí `OES_texture_3D`). Je silně doporučeno nepoužívat textury jiné velikosti než „power of two“. Použitím takových textur by mohlo dojít k pomalejšímu zpracování a některé starší grafické čipy takovéto textury vůbec nezobrazí. Podpora komprimovaných textur je garantována opět pouze rozšířením (`EXT_compressed_ETC1_RGB8_sub_texture`). OpenGL ES 2.0 nepodporuje ETC2 kompresy textur.

Shadery jsou omezeny například chybějící proměnnou `gl_FragDepth` používanou k modifikaci z-bufferu ve fragment shaderu (`EXT_frag_depth`). Ve fragment shaderu také není možné získat jednotlivé levely detailu textury. Je to umožněno pouze ve vertex shaderu pomocí funkce `texture2DLod`. OpenGL ES shadery také nepodporují implicitní typové konverze, je nutné přetypovávat explicitně například pomocí `float(2)`.

3.4.1 Vestavěné konstanty

Maximální počty proměnných v OpenGL ES 2.0 shaderech (GLSL 100) jsou závislé na konkrétní zařízení (implementace funkcí OpenGL ES konkrétním grafickým hardwarem), proto je důležité sledovat kolik informací je do shaderů posíláno. Maximální počet vstupních atributů, které může využít vertex shader je určen implementačně-závislou proměnnou `GL_MAX_VERTEX_ATTRIBS` (v OpenGL ES 2.0 musí být tato hodnota minimálně 8), podobně je omezen počet uniform vectorů, které může vertex shader využívat, omezen implementačně-závislou proměnnou `GL_MAX_VERTEX_UNIFORM_VECTORS`. Seznam dalších implementačně závislých proměnných a jejich význam je dostupný v Tab. 6.

Hodnoty implementačně závislých proměnných lze zjistit pomocí dotazů. Funkce glGetIntegerv umožňuje získat hodnoty OpenGL stavových proměnných. Následuje ukázka výpisu hodnoty proměnné `GL_MAX_VERTEX_ATTRIBS`.

```
int[] query = new int[1];
GLES20.glGetIntegerv(GLES20.GL_MAX_VERTEX_ATTRIBS,query, 0 );
Log.i(TAG,"Maximum Vertex attributes: "+ Integer.toString(query[0]));
```

Tab. 6: Konstanty v OpenGL ES

Vestavěné Konstanty v OpenGL ES 2.0	Min. hodnota	Význam
GL_MAX_VERTEX_ATTRIBS	8	Maximální počet atributů využitelných vertex shaderem
GL_MAX_VERTEX_ UNIFORM_VECTORS	128	Maximální počet uniform vektorů využitelných vertex shaderem
GL_MAX_VARYING_VECTORS	8	Maximální počet Varying vektorů, kterými vertex shader předává data fragment shaderu
GL_MAX_VERTEX_ TEXTURE_IMAGE_UNITS	0	Maximální počet textur přístupných vertex shaderu.
GL_MAX_COMBINED_ TEXTURE_IMAGE_UNITS	8	Celkový počet textur přístupných GL (vertex a fragment shaderu)
GL_MAX_TEXTURE_ IMAGE_UNITS	8	Maximální počet textur přístupných fragment shaderu
GL_MAX_FRAGMENT_ UNIFORM_VECTORS	16	Maximální počet uniform vektorů využitelných fragment shaderem
GL_MAX_RENDERBUFFER_SIZE	1	Maximální počet vykreslovacích bufferů přístupných fragment shaderu
GL_MAX_VIEWPORT_DIMS		Maximální dimenze viewportu
GL_MAX_TEXTURE_SIZE	64	Maximální velikost obrázku textury
GL_MAX_CUBE_MAP_ TEXTURE_SIZE	16	Maximální velikost obrázku cube map textury

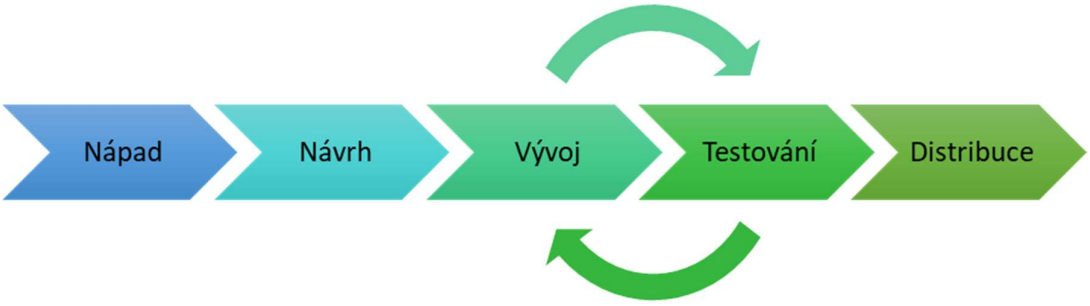
4 Návrh grafické aplikace pro Android OS

Pro využití OpenGL ES na platformě Android je nutné znát postupy tvorby aplikace pro tento systém. Je nejprve nutné vytvořit samotnou aplikaci a naučit se s ní pracovat a až poté je možné využívat v ní knihovnu OpenGL ES.

Vývoj každé aplikace je rozsáhlý proces začínající nápadem a končící uvolněním hotového díla do světa. Prvním krokem při vývoji jakéhokoli aplikačního softwaru (zkráceně aplikace) je nápad a s tím také souvisí určení hlavního účelu softwaru a cílové skupiny uživatelů. Je také nutné zvolit formu, kterou bude aplikace šířena, a tedy zvolit operační systém, na který se zaměří, přičemž je také důležitá verze operačního systému, která ovlivňuje funkce, které systém poskytuje.

Dalším krokem je typicky návrh vzhledu uživatelského rozhraní, který zahrnuje rozmístění ovládacích prvků a výběr funkcí, které by aplikace měla nabízet. Je nezbytné si uvědomit, že každý typ zařízení nabízí jiné ovládací prvky. Na přenosných zařízeních nejspíše budou k dispozici dotykové ovládací prvky, případně různé senzory, jimiž jsou dnešní zařízení vybavena. Tomu je potřeba přizpůsobit uživatelské rozhraní a mechanismy aplikace.

Po dokončení návrhu je vhodné přejít k samotnému vývoji aplikace. Pro vytvoření spustitelné aplikace potřebujeme v prvé řadě odpovídající vývojové prostředí, v závislosti na cíleném zařízení a operačním systému. V tomto prostředí poté probíhá samotné psaní kódu aplikace a další důležitá součást vývoje – testování kódu, jež může probíhat pomocí emulátoru nebo na reálném zařízení. Tento cyklus je opakován, nejlépe po každé změně v kódu, dokud není vytvořen finální produkt.



Obr. 3: Proces vývoje aplikace

Po dokončení je aplikace vypuštěna do světa pomocí příslušné distribuční služby. V tomto stádiu by měla být kompletní, avšak často se stává, že během

provozu jsou uživateli objeveny nedostatky a je nutné je opravit. Pro každou opravu se tak znovu opakuje vývojový cyklus.

4.1 Android Aplikace

Aplikace vytvářené v této práci budou zaměřeny na operační systém Android využívaný mobilními zařízeními (přesněji Android verze 4.0.3. a vyšší). Lze tedy očekávat práci hlavně s dotykovým displejem a je nutné tomu přizpůsobit ovládání. Jako vývojové prostředí bude použito Android Studio.

Pro vytvoření grafické aplikace pomocí OpenGL ES navíc musíme vybrat vhodnou verzi této knihovny. V současné době všechna prodaná zařízení podporují nejméně verzi 2.0 (Tab. 3), která přinesla možnosti programovatelného zobrazovacího řetězce, i proto je vhodné staršími verzemi se nezabývat.

4.1.1 Předpoklady

Pro vytváření grafických aplikací je v první řadě nutné znát základy programování v nějakém programovacím jazyku. V této práci je použit jazyk Java, který je podporován Android Studiem. Je také potřeba vědět, jak pracuje aplikace pro Android a jak zpracovávat vstupy z mobilního zařízení. Pro vytváření grafických aplikací je také nutné znát postupy v počítačové grafice a vědět, jak funguje knihovna OpenGL. Část potřebných znalostí lze získat v této práci, avšak je nutné mít potřebné základy.

4.2 Vývojové prostředí

4.2.1 Příprava prostředí – Android Studio

Nejpoužívanějším vývojovým prostředím (IDE – Integrated Development Environment) pro operační systém Android bylo v minulosti prostředí Eclipse s nainstalovaným pluginem ADT. Později (listopad 2014) však bylo nahrazeno novým prostředím od společnosti Google s názvem Android Studio, které je v současné době považováno za oficiální IDE pro platformu Android. Android Studio je dostupné na operačních systémech Windows, Mac, Linux a podporuje vývoj na všechny verze systému Android. Pro účely této práce byl použit operační systém Windows 10 (64-bit).

4.2.2 Instalace

Cílem Android Studia je co nejvíce ulehčit práci vývojářům, a navíc zrychlit celý proces vývoje.

Prvním krokem při přípravě Android Studia je stažení instalačního souboru na adrese: https://developer.android.com/studio/index.html. Instalace obsahuje vše nutné k vývoji aplikace včetně Android SDK a Android Virtual Device. Android SDK (software development kit) obsahuje vše potřebné k tvorbě aplikací pro Android (vyžadované knihovny, ukázkový kód, dokumentaci) určité verze. Android Virtual Device umožňuje testovat vytvořené aplikace přímo na počítači pomocí virtuálního zařízení (tablet, mobil) bez nutnosti připojení vlastního zařízení. Po prvním spuštění Android Studia je zkontrolována instalace Android SDK a vše je připraveno.

4.2.3 Vytvoření projektu

Důležitou věcí při vytváření projektu je volba zařízení, na kterém vytvořená aplikace poběží (chytrý telefon/tablet, hodinky, TV, obrazovka v autě). U každého zařízení je také nutné vybrat minimální verzi Android, kde bude spustitelná. Android Studio s výběrem verze dost pomáhá a obsahuje aktuální informace o podílu používaných verzí. V současné době 100 % zařízení používá verzi Android 4.0.3 a vyšší což odpovídá API 15. Dále je možné vybrat jednu z přednastavených obrazovek, čímž je možné ušetřit práci (např. Fullscreen Activity vytvoří aplikaci v maximalizovaném okně, doporučuji Empty Activity).

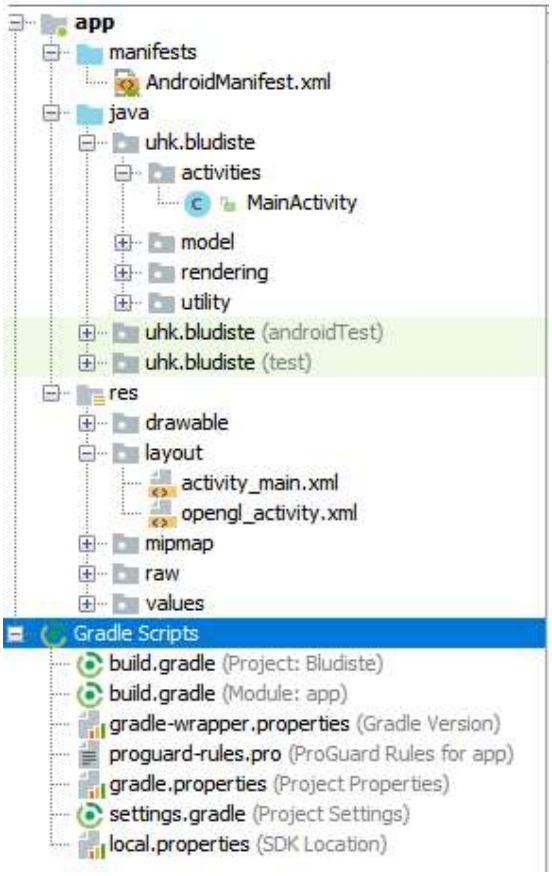
4.2.4 Pomocné knihovny

Android studio podporuje tvorbu 2D a 3D aplikací pomocí knihovny OpenGL ES, která je součástí instalace. Knihovna je dostupná v několika verzích, lišících se dostupností některých funkcí a efektivitou využití systémových zdrojů. V současnosti je nejnovější verzí OpenGL ES 3.2, která je podporována pouze novějšími zařízeními (Android 7.0 a výše). Pro účely této práce je použita verze OpenGL ES 2.0, protože je podporována většinou zařízení na trhu a výsledné aplikace tak budou spustitelné na co největším počtu zařízení.

4.2.5 Struktura projektu

Android studio po vytvoření projektu automaticky vytvoří modul „app“, který obsahuje veškeré zdrojové kódy aplikace. Moduly se používají pro rozdělení projektu na verze pro různá zařízení, kód je tak stále seskupený pod jedním projektem, ale je možné kompletně změnit funkce na základě potřeby jednotlivých zařízení.

Android studio implicitně nezobrazuje soubory ve složkách, ale rozděluje soubory do skupin podle jejich účelu. Na nejvyšším stupni hierarchie jsou umístěny jednotlivé moduly v projektu a také skupina „Grandle Scripts“ obsahující veškeré soubory potřebné k sestavení projektu.



Obr. 4: Struktura projektu v Android studiu

Modul Android aplikace je rozdělen do tří hlavních skupin: manifest, java, res. Skupina manifest obsahuje soubor AndroidManifest.xml se všemi informacemi o konfiguraci aplikace včetně názvu aplikace, vyžadované verze OpenGL, požadavků

na funkce telefonu a názvu aktivity, která bude spuštěna při spuštění aplikace. Skupina java obsahuje veškeré zdrojové soubory rozdělené do jednotlivých balíčků. Defaultně jsou ve skupině java také vloženy balíčky pro unit testy. Další skupina res obsahuje všechny ostatní zdroje aplikace, například shadery, textury, zvukové soubory, ale hlavně složku layout obsahující uspořádání uživatelského rozhraní aplikace. Je také vhodné do projektu přidat složku Assets (kliknout pravým na modul – app→New→Folder→Assets Folder) a používat ji místo složky res. Složka assets totiž navíc umožňuje přístup pomocí názvu souboru namísto čísla ID používaného ve složce res.

4.2.6 Export aplikace a digitální podpis

Systém Android nedovolí instalaci nepodepsaných aplikací, proto je nutné aplikaci digitálně podepsat certifikátem – vyžadováno pro nahrání na Google Store. Jedná se o připojení informací o autorovi aplikace případně zodpovědné organizaci. Pro testování aplikace Android Studio v debug módu generuje tzv. debugovací certifikát, který umožňuje spustit jinak nepodepsanou aplikaci, avšak pro nahrání na většinu distribučních služeb tento certifikát nestačí.

Android Studio umožňuje vytvoření podepsané aplikace následujícím způsobem. Nejprve je nutné vytvořit klíč a soubor keystore do kterého se uloží. Součástí klíče je také certifikát s informacemi o autorovy aplikace. Klíč a keystore je také možné vytvořit pomocí aplikace Google Play. Poté už stačí jen při exportu aplikace zvolit soubor keystore a zadat odpovídající heslo.

Tvorba podepsané aplikace:

1) Build → Generate Signed APK.
2) Create new / Choose existing
    a. Vyplňte údaje (hesla klíče a keystore by měla být odlišná)→OK
3) Potvrďte předvyplněné údaje o klíči/keystore (při použití klíče z Google Play je nutné ručně vybrat klíč a zadat hesla) → Next
4) Vyberte typ sestavení projektu a umístění kam se uloží soubor .apk → Finish

4.3 Uživatelské rozhraní

Úspěch aplikace je založen zejména na přehlednosti a intuitivnosti ovládacích prvků tvořících rozhraní mezi uživatelem a aplikací. Je nutné přizpůsobit tyto prvky velikosti zařízení, na kterém aplikace poběží.

Android Studio nabízí jednoduchý způsob navržení jednotlivých obrazovek zobrazených v aplikaci. Součástí projektu jsou xml soubory, reprezentující rozmístění všech prvků na obrazovce. Každá aktivita (obrazovka) má svůj vlastní xml soubor, který je možné upravovat buď přímo textově úpravou xml, nebo pomocí grafického editoru. V editoru je možné na model obrazovky přetahovat jednotlivé komponenty a rozmístit je podle potřeby.

4.4 Vývoj

Hlavním prvkem každé android aplikace jsou aktivity, které reprezentují jednotlivé obrazovky, mezi kterými je možno přepínat. Typicky jsou to oddělené části aplikace, každá aktivita musí být zanesena v souboru AndroidManifest.xml pomocí tagu activity. Aktivita, která má být spuštěna při startu aplikace musí být označena pomocí následujícího kódu.

```
<activity android:name="packageName.ActivityName">
    <intent-filter>
       <action android:name="android.intent.action.MAIN" />
       <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

Samotná aktivita je reprezentována třídou, která dědí ze třídy `android.app.Activity`. Tato třída by měla obsahovat vlastní implementaci funkcí `onCreate()`, `onResume()` a `onPause()`. Funkce `onCreate()` je zavolána pokaždé když je aktivita vytvořena a měla by obsahovat kód vybírající rozložení aplikace `setContentView(R.layout.layoutName)` a tvorbu prvků uživatelského rozhraní. Funkce `onResume()` a `onPause()` jsou volány pokaždé, když je aplikace přesunuta do/z pozadí otevřením jiné aplikace.

4.4.1 Tvorba OpenGL ES prostředí v Android aplikaci

O veškerou interakci OpenGL ES s aplikací se stará třída `android.opengl.GLSurfaceView`. Je vhodné vytvořit si vlastní implementaci této třídy jejím děděním. Ve třídě je důležité přepsat funkci `onTouchEvent()` sloužící k obsluze událostí spojených s dotykem obrazovky pomocí notace `@Override`. Dále je potřeba vytvořit si vlastní třídu `CustomRenderer` starající se o vykreslování (děděním třídy `android.opengl.GLSurfaceView.Renderer`). Toho lze nejlépe docílit přepsáním metody `setRenderer()` následujícím způsobem.

```
Public void setRenderer(package.CustomRenderer renderer) {
     this.renderer = renderer;
     super.setRenderer(renderer);
}
```

Třída `CustomRenderer` sestává ze tří stěžejních metod. V první řade se jedná o funkci `onSurfaceCreated()` volanou při vytvoření instance třídy. Funkce `onDrawFrame()` je volána pro vykreslení obrazovky několikrát za vteřinu. A funkce `onSurfaceChanged()` je volána při změně šířky obrazovky.

S takto připravenými třídami je možné přidat do funkce `onCreate()` ve vybrané třídě aktivity kód vytvářející instanci nově třídy `CustomGLSurfaceView`. Tuto třídu je také možné přidat přímo do rozložení, která využívá aktivita (soubor layoutName.xml ve složce res/layout).

```
<fullPackage.CustomGLSurfaceView
     android:id="@+id/customSurfaceView"
     android:layout_width="match_parent"
     android:layout_height="match_parent">
</fullPackage.CustomGLSurfaceView>
```

Ve funkci `onCreate()` v aktivitě poté stačí pouze najít v rozložení prvek `customSurfaceView`, nastavit OpenGL ES kontext na požadovanou verzi a přiřadit ke kontextu vlastní `Renderer`.

```
// Pick layout
setContentView(R.layout.layoutName);
// Find GLsurfaceView in layout
sample_GL_View = findViewById(R.id.customSurfaceView);
// Create OpenGL ES 2.0 context.
sample_GL_View.setEGLContextClientVersion(2);
// Set the renderer
sample_GL_View.setRenderer(new CustomRenderer());
```

4.4.2 Inicializace OpenGL ES

Třída `CustomRenderer` popsaná v předchozí kapitole provádí veškerá volání OpenGL ES funkcí. Pro správné fungování grafické aplikace většinou potřebují nějaká externí data, ať už to jsou obrázky textur, soubory s daty 3D objektů nebo soubory s shaderovým kódem. Načítání externích dat většinou probíhá pouze jednou při startu aplikace, a to ve funkci `onSurfaceCreated`. Aplikace by měla nejprve zjistit jaká je podporovaná verze shaderů a podle toho načíst příslušné soubory se shadery. Dalším důležitou věcí, která by měla v této části být provedena je tvorba bufferů pro uložení geometrie a dat spojených s vrcholy.

Pro účely vývoje je vhodné vypsat do konzole například informace o verzi shaderů nebo OpenGL podporovanou zařízením. Dále proběhne načtení případných textur, 3D objektů nebo dalších požadovaných souborů a inicializace proměnných na počáteční hodnoty. Je zde také možné například uložit pozice uniform proměnných používaných shadery anebo nastavit stavové proměnné OpenGL, které nemusí být měněny při každém vykreslení obrazovky. Příklad obsahu funkce `onSurfaceCreated` je vidět v následujícím bloku kódu.

```
GLES20.glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
OGLUtils.shaderCheck(maxGlEsVersion);
GLUtils.printOGLparameters(maxGlEsVersion);

if (OGLUtils.getVersionGLSL(maxGlEsVersion)<300)
   shaderProgram = ShaderUtils.loadProgram(activity, maxGlEsVersion,
   "shaders/lvl1basic/p03texture/p01intro/textureOld");
else
   shaderProgram = ShaderUtils.loadProgram(activity, maxGlEsVersion,
   "shaders/lvl1basic/p03texture/p01intro/texture");

createBuffers();
locMat = GLES20.glGetUniformLocation(shaderProgram, "mat");
// load texture
texture = loadTexture("textures/mosaic.jpg");
cam = cam.withPosition(new Vec3D(5, 5, 2.5))
     .withAzimuth(Math.PI * 1.25).withZenith(Math.PI * -0.125);
GLES20.glEnable(GLES20.GL_DEPTH_TEST);
```

4.4.3 Grafické objekty – buffery

Pro reprezentaci grafických objektů v OpenGL ES jsou používány tzv. buffery. Buffer obsahuje pole s přesně určeným pořadím prvků vykreslitelné pomocí

OpenGL. Jedná se o vertex buffery obsahující data vrcholů a index buffery obsahující čísla vrcholů v pořadí, ve kterém mají být vykresleny. Prvky tohoto pole představují jednotlivé složky pozice, barvy, normály nebo souřadnice v textuře jednotlivých vrcholů vykreslovaných tvarů. Pořadí i počet prvků určují data, která mají být reprezentována. Například reprezentace trojúhelníku se třemi barevnými vrcholy bude vypadat následujícím způsobem.

```
float[] vertexBufferData = {
     -1, -1,     0.7f, 0, 0,
     1,  0,      0, 0.7f, 0,
     0,  1,      0, 0, 0.7f
};
short[] indexBufferData = { 0, 1, 2 };
```

OpenGL ES 2.0 obsahuje omezení maximální velikosti prvků index bufferu, proto je použit pouze datový typ `short`. Pro vertex buffery se nejčastěji využívá typ `float`. Pro větší počty vykreslovaných objektů je vždy lepší používat co nejmenší počet vertex bufferů a vyhnout se tak „drahému“ přepínání mezi jednotlivými buffery. Použití index bufferů umožňuje vytáhnout jednotlivé vrcholy tvořící objekt přímo z vertex bufferu.

Předtím, než OpenGL může data vykreslit, musejí buffery mít alokovanou paměť. K tomuto účelu slouží programový balíček `java.nio`. Důležitý je také datový typ bufferu, protože tvorba přímých bufferů v OpenGL ES 2.0 je umožněna pouze pro Byte buffery. Tvorba přímých bufferů jiných datových typů je umožněna pouze naplněním bufferu požadovaného typu `byte` bufferem. Také je důležité si uvědomit, že jde o načtení dat z pole datového typu `float` do bufferu typu `byte`, proto musíme při tvorbě bufferu násobit délku pole číslem 4(v kódu tučně). Datový typ `float` je reprezentován 32bity paměti nebo také 4 byty. Podobně například `short` odpovídá 2 bytům. Následující úryvek kódu zobrazuje vytvoření vertex bufferu. Tvorba index bufferu je identická až na datový typ, který bude `short` (délka pole je násobena dvěma).

```
private int[] vertexBuffer = new int[1], indexBuffer = new int[1];
FloatBuffer vertexBufferBuffer = ByteBuffer         //Short
     .allocateDirect(vertexBufferData.length * 4)   //2
     .order(ByteOrder.nativeOrder()).asFloatBuffer();
vertexBufferBuffer.put(vertexBufferData);
vertexBufferBuffer.position(0);
```

Takto připravené buffery lze jednoduše propojit s OpenGL ES. Nejdříve OpenGL ES vygeneruje požadovaný počet bufferů a uloží odkazy na ně do pole vloženého jako parametr funkce `glGenBuffers`. Poté je pomocí tohoto odkazu vybrán buffer funkcí `glBindBuffer`. Následuje načtení dat z připravených bufferů do bufferů spravovaných OpenGL pomocí funkce `glBufferData`. Při práci s OpenGL buffery je používán parametr `GL_ARRAY_BUFFER` pro práci s vertex buffery a `GL_ELEMENT_ARRAY_BUFFER` pro práci s index buffery. Velikost bufferu opět závisí na velikosti použitého datového typu v bytech. Poslední parametr funkce určuje použití dat bufferu a může nabývat hodnot: `GL_STREAM_DRAW` (data budou měněna pouze jednou a poté pouze několikrát použita), `GL_STATIC_DRAW` (data budou měněna pouze jednou a poté mnohokrát použita), nebo `GL_DYNAMIC_DRAW` (data budou měněna několikrát a poté mnohokrát použita).

```
private int[] vertexBuffer = new int[1];
GLES20.glGenBuffers(1, vertexBuffer, 0);
GLES20.glBindBuffer(GLES20.GL_ARRAY_BUFFER, vertexBuffer[0]);
GLES20.glBufferData(GLES20.GL_ARRAY_BUFFER,vertexBufferData.length*4,
    vertexBufferBuffer, GLES20.GL_STATIC_DRAW);
```

Vykreslení bufferů probíhá ve funkci `onDrawFrame` a začíná výběrem bufferu pomocí `glBindBuffer` (zde využijeme odkaz vytvořený při generování bufferů OpenGL), poté je vybrán atribut vrcholu, se kterým se bude pracovat (`glEnableVertexAttribArray`), a nakonec jsou nastaveny parametry atributu jako je jeho typ nebo délka pomocí funkce `glVertexAttribPointer`. Jednotlivé atributy odkazují na proměnné shaderu a přistupujeme k nim pomocí jejich pozice v paměti získané funkcí `glGetAttribLocation`. Podobně postupujeme pro všechny atributy ve vertex bufferu. Důležité jsou zejména poslední dva parametry funkce `glVertexAttribPointer`, které určují velikost dat jednoho vrcholu a vzdálenost od začátku dat vrcholu v bytech (pozice – 2 elementy typu `float` – 8 bytů, barva – 3 elementy `float` – 12 bytů, celkem 20 bytů).

```
int locPosition = GLES20.glGetAttribLocation(shaderProgram,
    "inPosition");
int locColor = GLES20.glGetAttribLocation(shaderProgram, "inColor");
GLES20.glBindBuffer(GLES20.GL_ARRAY_BUFFER, vertexBuffer[0]);
GLES20.glEnableVertexAttribArray(locPosition);
// shader variable ID, number of components, type of components,
// normalize?, size of a vertex in bytes, location of first component
GLES20.glVertexAttribPointer(locPosition, 2, GLES20.GL_FLOAT, false,
    20, 0);
GLES20.glEnableVertexAttribArray(locColor);
// color information starts at 0 + 8(position) = 8
GLES20.glVertexAttribPointer(locColor, 3, GLES20.GL_FLOAT, false,
    20, 8);
GLES20.glBindBuffer(GLES20.GL_ELEMENT_ARRAY_BUFFER, indexBuffer[0]);
```

Následně můžeme přepnout na index buffer a vykreslit data geometrie pomocí `glDrawElements`. Funkce akceptuje pouze datový typ `GL_UNSIGNED_BYTE`, `GL_UNSIGNED_SHORT` nebo `GL_UNSIGNED_INT` (vyžaduje podporu zařízení).

```
GLES20.glBindBuffer(GLES20.GL_ELEMENT_ARRAY_BUFFER, indexBuffer[0]);
// draw - MODE(triangles), number of vertexes, type of vertexes,
    // location of indices
GLES20.glDrawElements(GLES20.GL_TRIANGLES,3,GLES20.GL_UNSIGNED_SHORT,
    0);
```

4.4.4 Shadery

Zavoláním vykreslovacích funkcí OpenGL ES jako `glDrawElements` jsou vložená data zpracována shadery. Program, který shadery vykonají, je nejdříve potřeba načíst a zkompilovat. Kód shaderu je ve většině případů uložen v externím souboru, ale je možné jej definovat přímo v programu proměnnou typu `String`. Načtení kódu programu z externího souboru v projektové složce assets je provedeno pomocí `context.getAssets().open(streamFileName)`. Tato funkce vyžaduje kontext aktuální Aktivity a otevře vstupní proud dat pomocí názvu cesty k souboru. Tento proud je možné například pomocí `BufferedReaderu` a funkce `readLine` postupně uložit do proměnné datového typu `String` následujícím způsobem. Vstupní proud dat musí být po ukončení práce uzavřen.

```
InputStream is = context.getAssets().open(streamFileName);
BufferedReader bufferedReader = new BufferedReader(
     new InputStreamReader (is, "UTF-8"));
String line;
final StringBuilder shader = new StringBuilder();
while ((line = bufferedReader.readLine()) != null) {
     shader.append(line);
}
is.close();
bufferedReader.close();
return shader.toString();
```

Po načtení kódu shaderu je možné vytvořit tzv. shader objekty. Pomocí funkce `glCreateShader` získáme odkaz na objekt shaderu. Funkce v parametru požaduje typ shaderu. Typem může v OpenGL ES 2.0 být pouze `GL_VERTEX_SHADER` nebo `GL_FRAGMENT_SHADER`. Dále načteme data s kódem shaderu do objektu shaderu pomocí dříve uloženého odkazu funkcí `glShaderSource` a kód zkompilujeme (`glCompileShader`). Takto připravíme vertex i fragment shader.

```
int vs = GLES20.glCreateShader(GLES20.GL_VERTEX_SHADER);
GLES20.glShaderSource(vs, shaderVertSrc);
GLES20.glCompileShader(vs);
```

Shader objekty musejí být přiřazeny k tzv. program objektu. Ten oba objekty spojí a vytvoří tak propojení mezi výstupem vertex shaderu a vstupem fragment shaderu. Funkce `glCreateProgram` vytvoří objekt programu a vrátí odkaz na tento objekt. Dále jsou pomocí funkce `glAttachShader` připojeny jednotlivé objekty shaderů k objektu programu pomocí odkazů získaných dříve. Po připojení všech objektů shaderů je program propojen (`glLinkProgram`).

```
int shaderProgram = GLES20.glCreateProgram();
GLES20.glAttachShader(shaderProgram, vs);
GLES20.glAttachShader(shaderProgram, fs);
GLES20.glLinkProgram(shaderProgram);
```

Takto vytvořený objekt programu je použit zavoláním funkce glUseProgram ve funkci `onDrawFrame`. Všechny následně volané GL příkazy budou při vykreslování používat tento objekt. Při procesu tvorby objektu programu a shader objektů je důležité zjišťovat, zda nedošlo k chybě, a pokud došlo zbavit se nefunkčních objektů. Doporučuji vyhledat dokumentaci k pochopení užití těchto funkcí (dokumentace OpenGL ES 2.0 funkcí: https://www.khronos.org/registry/OpenGL-Refpages/es2.0/xhtml/). Informace o shader objektu lze zjistit pomocí funkce

`glGetShaderiv` a informace o objektu programu podobně pomocí `glGetProgramiv`. Smazání objektů je provedeno pomocí `glDetachShader`, `glDeleteShader` a `glDeleteProgram`.

Vertex shader obsahuje tři typy proměnných, které je nutné navázat na data. Připojení vstupních proměnných vertex shaderu s označením `attribute` je vykonáno při tvoření objektů bufferů, `varying` proměnné jsou propojením výstupu vertex a vstupu fragment shaderu – toto propojení je vytvořeno při vytvoření objektu programu. `Uniform` proměnné jsou dostupné z vertex i fragment shaderu a jejich připojení probíhá většinou během vykreslovacího cyklu. Nejdříve je po spuštění aplikace získán odkaz na proměnnou shaderu pomocí `glGetUniformLocation`. Ve vykreslovacím řetězci je potom tímto odkazem proměnná naplněna pomocí funkce odpovídající datovému typu proměnné.

```
// in onSurfaceCreated
int locHeight = GLES20.glGetUniformLocation(shaderProgram, "height");
// in onDrawFrame
GLES20.glUniform1f(locHeight, height);
```

4.4.5 Textury OpenGL ES

Textury v OpenGL označují objekty s obrázky vykreslované na povrch geometrický těles nebo použité jako cíl vykreslení. Klasické použití spočívá v načtení dat obrázku z externího souboru a následné mapování pozic obrázku na existující geometrii pomocí texturových souřadnic. Pro načtení externího obrázku z projektové složky assets použijeme stejný postup jako při načítání souboru se shaderem s tím rozdílem, že vstupní proud dat bude použit jinak. Nejdříve použijeme funkci `glGenTextures` pro vytvoření objektu textury a získání odkazu na něj. Následně pomocí `android.graphics.BitmapFactory` dekódujeme vstupní proud dat jako Bitmap. Bitmap (`android.graphics.Bitmap`) je objekt, který systém android využívá pro rychlé načtení obrazových dat do OpenGL. `BitmapFactory` podporuje celou řadu formátů včetně JPG, PNG, GIF nebo BMP. Pomocí funkce `glBindTexture` vybereme odkaz na vytvořenou texturu. Všechny následně volané GL funkce budou ovlivňovat tuto texturu. OpenGL ES 2.0 podporuje dva druhy textur `GL_TEXTURE_2D` (dvourozměrná textura) nebo `GL_TEXTURE_CUBE_MAP` (6 spojených textur).

```
InputStream is = context.getAssets().open(fileName);
// texture handler
final int[] textureHandler = new int[1];
GLES20.glGenTextures(1, textureHandler, 0);
final Bitmap bitmap = BitmapFactory.decodeStream(is);
GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, textureHandler[0]);
```

U zvolené textury je možné nastavit typ filtrování, které OpenGL aplikuje při jejím vykreslení v menší nebo větší velikosti. Při zvětšování je použit parametr `GL_TEXTURE_MAG_FILTER` a při zmenšování textury je použit parametr `GL_TEXTURE_MIN_FILTER`. Dalšími parametry textur jsou `GL_TEXTURE_WRAP_S` a `GL_TEXTURE_WRAP_T` určující chování při situaci kdy textura nemá dostatečnou šířku nebo výšku (souřadnice textury mimo 0.0–1.0). Pomocí funkce texImage2D lze lehce načíst data bitmapu přímo do OpenGL. Po předání dat není bitmap dále potřeba a lze uvolnit jeho paměť. OpenGL také umí automaticky vygenerovat mipmapy textury, které jsou použiti pro zlepšení kvality při zvětšování/zmenšování textury. Nakonec vždy uzavřeme otevřený proud dat.

```
GLES20.glTexParameteri(GLES20.GL_TEXTURE_2D,
     GLES20.GL_TEXTURE_MIN_FILTER, GLES20.GL_LINEAR_MIPMAP_NEAREST);
GLES20.glTexParameteri(GLES20.GL_TEXTURE_2D,
     GLES20.GL_TEXTURE_MAG_FILTER, GLES20.GL_LINEAR);
GLUtils.texImage2D(GLES20.GL_TEXTURE_2D, 0, bitmap, 0);

// generate mipmaps
GLES20.glGenerateMipmap(GLES20.GL_TEXTURE_2D);
is.close();
bitmap.recycle();
return textureHandler[0];
```

Většina běžných formátů obrázků je oproti OpenGL převrácená na ose Y, proto je vhodné obrázky před použitím s OpenGL nejdříve překlopit. Nejjednodušší varianta je překlopení obrázků v grafickém editoru ještě před načtením aplikací. Dalšími variantami je úprava texturových souřadnic nebo převrácení bitmapu hned po načtení textury ze souboru. Je také možné převrátit souřadnice textury přímo v shaderech. Převrácení bitmapu podle osy Y je možné pomocí následujícího kódu.

```
Matrix matrix = new Matrix();
matrix.postScale(1, -1, bitmap.getWidth()/2f, bitmap.getHeight()/2f);
Bitmap.createBitmap(bitmap, 0, 0, bitmap.getWidth(),
                        bitmap.getHeight(), matrix, true);
```

Ve fragment shaderu je proměnná textury označena `sampler2D` nebo `samplerCube` pro odpovídající druhy textur. Před vykreslením objektu s aplikovanou texturou je nutné vybrat aktivní OpenGL slot textury (`glActiveTexture`), připojit k aktivnímu slotu dříve vygenerovaný odkaz na texturu (`glBindTexture`) a nakonec připojit proměnnou shaderu k nějakému slotu OpenGL (`glUniform1i`) pomocí odkazu na proměnnou shaderu.

```
GLES20.glActiveTexture(GLES20.GL_TEXTURE0);
GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, textureHandler[0]);
GLES20.glUniform1i(
    GLES20.glGetUniformLocation(shaderProgram, "texture"), 0);
```

4.5 Testování

Důležitou složkou tvorby každé aplikace je testování. Každou změnu v aplikaci je nutné otestovat spuštěním přímo na nějakém zařízení. Při tvorbě větších aplikací je počet testů v řádech tisíců, proto je důležité proces testování co nejvíce urychlit. Android studio nabízí emulátor, který dokáže simulovat řadu zařízení a přímo v počítači je tak možné vyzkoušet, jak se aplikace chová na různých typech tabletů nebo telefonů. Nevýhoda emulátorů je potřeba výkonného počítače a pomalý průběh testování (jedno spuštění může trvat několik minut). Nejlepším řešením je testovat přímo na reálném zařízení. V tomto případě stačí jen aplikaci přesunout na zařízení a spustit (spuštění většinou proběhne během několika sekund). Zařízení použité pro testování musí mít zapnutý Debug mode a povoleny instalace z neznámých zdrojů.

Pro testovací účely je v jazyce Java na desktopech často využívána funkce `System.out.println()` pro zobrazení potřebných informací v konzoli. Při programování pro systém Android za použití Android Studia není dostupná klasická konzole. Místo konzole je v Android Studiu použit nástroj Logcat (View→Tool Windows→Logcat). Výpis informací probíhá pomocí třídy `android.util.Log` a je tříděn do několika kategorií. Zpráva bude zařazena do kategorie podle použité funkce, například zprávy chyb by měly být vypisovány pomocí `Log.e(String, String)`. Typy zpráv: `Log.v` – verbose, `Log.d` – debug, `Log.i` – information, `Log.w` – warning, `Log.e` – error. První parametr těchto funkcí slouží k označení pro pozdější filtraci výstupu v nástroji Logcat. Typicky je v tomto parametru například

pojmenování třídy, ve které byl vypsán. Druhým parametrem je zpráva, která se objeví v nástroji Logcat.

4.6 Distribuce

Finální částí tvorby aplikace je její distribuce mezi budoucí uživatele. Pro systém Android je oficiální distribuční službou Google Play, vyvíjená a udržovaná firmou Google. Služba umožňuje nahrání a údržbu aplikací a získání zpětné vazby přímo od uživatelů. Před nahráním první aplikace je nutné uhradit jednorázový poplatek v hodnotě $25, kterým Google mírně filtruje zbytečné aplikace. Dále si Google účtuje 30 % za veškerý zisk z prodeje vašich aplikací. Česká republika patří mezi země, ve kterých Google dovoluje účtovat za nahrané aplikace peníze, je však možné aplikace poskytovat zcela zdarma a případně vydělávat reklamou umístěnou přímo ve vaší aplikaci. Google Play očekává aplikaci ve formátu .apk podepsanou certifikátem.

5 Výsledky testování tvorby grafické aplikace na mobilních zařízení

5.1 Parametry

5.1.1 Proč se bude testovat?

Úroveň kvality desktopových grafických aplikací vytvořila určitý standard. Uživatelé vyžadují stále vyšší a vyšší vizuální kvalitu aplikací a nespokojí se s aplikací, která tuto kvalitu neposkytuje. Tvůrci mobilních aplikací proto nechtějí zůstat pozadu a snaží se stejných efektů dosáhnout i s mnohem menšími výpočetními možnostmi. Z tohoto důvodu existuje snaha přepisovat aplikace původně vytvořené pro použití s OpenGL do podoby odpovídající OpenGL ES.

5.1.2 Co se bude testovat?

V průběhu testů bude testováno, zda je možné s použití OpenGL ES 2.0 vytvořit nebo napodobit stejné efekty jaké jsou dosažitelné pomocí OpenGL. Součástí práce jsou ukázky aplikací zobrazující možnosti grafické knihovny OpenGL ES. Tyto ukázky jsou založeny na příkladech používaných v předmětu PGRF3. Původní ukázky vznikly pro znázornění možností programovatelného grafického řetězce OpenGL a byli upraveny tak, aby byli použitelné na systému Android s knihovnou OpenGL ES. Dále budou testovány hodnoty implementačně závislých proměnných omezujících práci s OpenGL ES. Novější verze knihovny nebudou testovány z důvodu nepřístupnosti testovacích zařízení, které by je podporovaly.

5.1.3 Na čem se bude testovat?

Testy Basic a Advanced ukázek budou prováděny nejprve na desktopovém počítači a poté na mobilním telefonu využívající systém Android. Test implementačně závislých proměnných bude pro porovnání probíhat navíc také na tabletu.

Počítač – Lenovo ideapad 320-15IKB

- OS: Windows 10 Home 64-bit
- Maximální verze OpenGL: 4.5
- Procesor: Intel Core i7-7500U CPU @ 2,70GHz 2,90GHz
- Grafická karta: NVIDIA GeForce 940MX 4 GB
- Rozlišení: 1920x1080
- Paměť: 12 GB RAM, 1000 GB HDD + 128 GB SSD

Mobilní telefon – Blackview A5

- OS: Android 6.0
- Maximální podporovaná verze OpenGL ES: 2.0
- Procesor: MT6580 QuadCore 1,3GHz
- Rozlišení: 960x540
- Paměť: 1 GB RAM + 8 GB ROM

Tablet Lenovo A3500-FL

- OS: Android 4.4.2
- Maximální podporovaná verze: OpenGL ES 2.0
- Procesor: 4x ARM Cortex-A7 1,30Ghz
- Rozlišení: 800x1280, 215dpi
- Paměť: 1 GB RAM

5.2 Test ukázek

V tomto testu bylo testováno, zda je možné dosáhnout s knihovnou OpenGL ES stejných efektů, jakých je schopna knihovna OpenGL. Ukázky jsou rozděleny do dvou skupin: Basic a Advanced úlohy. Basic úlohy testují základní grafické funkce (vykreslení geometrických primitiv pomocí bufferů) a Advanced úlohy testují pokročilejší funkce (práce s volume texturami, teselace, geometry shader). Nejsou zde do detailu popsány postupy využívané úlohami, ale pouze rozdíly verze těchto ukázek pro OpenGL ES v porovnání s předlohovou verzí vytvořenou pro knihovnu OpenGL. Některé ukázky byly kvůli lepší viditelnosti znázorněny pomocí snímků obrazovky tabletu.

5.2.1 Basic úlohy

Basic úlohy jsou navrženy pro testování základní funkcionality knihovny OpenGL ES. Následující tabulka obsahuje seznam testovaných ukázek a jejich proveditelnost s knihovnou OpenGL ES 2.0.

Tab. 7: Funkčnost basic ukázek

Označení – název ukázky	**OpenGL ES 2.0**
B00 – Vykreslení trojúhelníku bez shaderů	Nefunkční
B11 – Vykreslení trojúhelníku pomocí Bufferu *uhk.android_samples.lvl1basic.p00.p01buffer*	Funkční
B12 – Buffer s atributem *uhk.android_samples.lvl1basic.p01start.p02attribute*	Funkční
B13 – Buffer s uniform proměnnou *uhk.android_samples.lvl1basic.p01start.p03uniform*	Funkční
B14 – Buffer s pomocí Utils *uhk.android_samples.lvl1basic.p01start.p04utils*	Funkční
B15 – Více shaderových programů *uhk.android_samples.lvl1basic.p01start.p05multiple*	Funkční
B16 – Vykreslení s depth testem *uhk.android_samples.lvl1basic.p01start.p06depthbuffer*	Funkční
B21 – Vykreslení krychle *uhk.android_samples.lvl1basic.p02geometry.p01cube*	Funkční
B22 – Vykreslovací módy *uhk.android_samples.lvl1basic.p02geometry.p02strip*	Částečně funkční
B23 – Vykreslení modelu ze souboru *uhk.android_samples.lvl1basic.p02geometry.p03obj*	Funkční s omezením
B31 – Vykreslení s texturou *uhk.android_samples.lvl1basic.p03texture.p01intro*	Funkční
B32 – Vykreslení s texturou pomocí Utils *uhk.android_samples.lvl1basic.p03texture.p02utils*	Funkční
B33 – Vykreslen několika textur na jeden objekt *uhk.android_samples.lvl1basic.p03texture.p03multiple*	Funkční
B41 – Vykreslení do textury *uhk.android_samples.lvl1basic.p04target.p01intro*	Částečně funkční
B42 – Vykreslení do textury pomocí Utils *uhk.android_samples.lvl1basic.p04target.p02utils*	Částečně funkční
B43 – Post-processing pomocí vykreslení do textury *uhk.android_samples.lvl1basic.p04target.p03postproces*	Částečně funkční

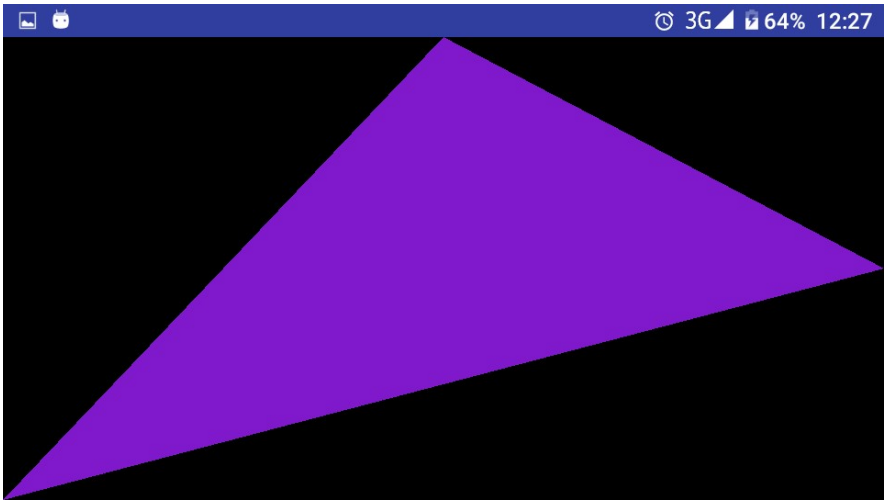
Úloha B00

První z úloh testuje vykreslení trojúhelníku pomocí funkcí pevného vykreslovacího řetězce. Ukázka je nefunkční z důvodu absence pevného řetězce v knihovně OpenGL ES.

Úloha B11

Úloha testuje vykreslení trojúhelníku pomocí programovatelného řetězce za použití bufferů v kombinaci s vertex a fragment shadery. S malými změnami byla úloha plně zprovozněna. První problém při tvorbě ukázky nastal při tvorbě bufferů. OpenGL ES neobsahuje funkcionalitu pro přímou alokaci paměti pro buffery s jiným datovým typem než `byte`, proto bylo nutné nejdříve alokovat paměť pro `byte` buffer (`java.nio.ByteBuffer`) s dostatečnou velikostí pro uchování `short` nebo `float` bufferů a poté buffery převést na požadovaný datový typ. Seřazení bytů může být rozdílné na různých zařízeních, a proto je nutné navíc před přiřazením `byte` buffer seřadit do nativního pořadí daného zařízení. Buffery dalších datových typů jako je `int`, `double`, `long` a `short` je nutné alokovat obdobně jako `float` buffer. Vytvoření `float` bufferu v OpenGL ES 2.0 je znázorněno v následujícím bloku kódu.

```
ByteBuffer bb = ByteBuffer.allocateDirect(data.length*4);
bb.order(ByteOrder.nativeOrder());              // 4 bytes per float
FloatBuffer buffer = bb.asFloatBuffer();
buffer.put(data); buffer.position(0);
```

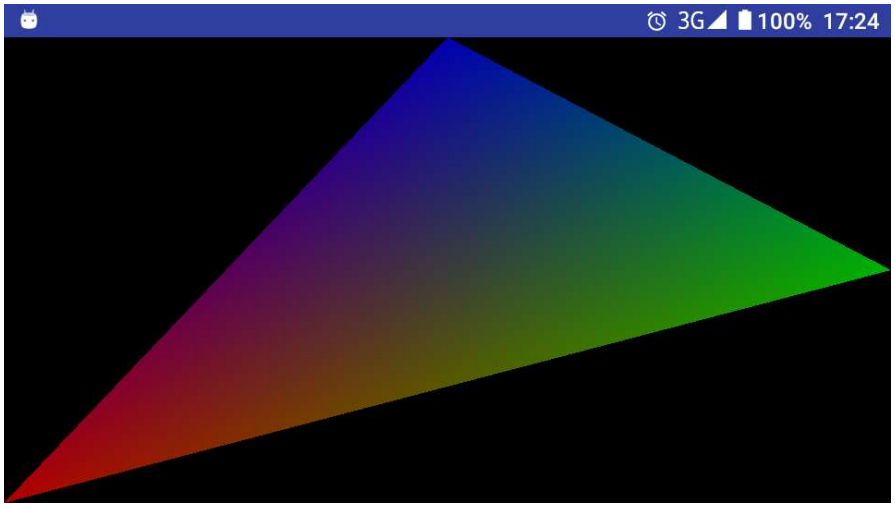


Obr. 5: Úloha B11 – Vykreslení trojúhelníku

Další důležitou změnou při přechodu na OpenGL ES bylo přepsání shaderů ukázek. Původní verze využívá pro psaní shaderů jazyk GLSL 150 a je proto nutné shadery přepsat na verzi GLSL 100 (pro OpenGL ES 2.0). Rozdíl spočívá hlavně ve změně označení vstupních a výstupních proměnných shaderů a v nutnosti označit přesnost atributů v GLSL 100 (Viz. kapitola 3.3). Dále je nutné smazat označení verze shaderu (`#version`), aby byl shader automaticky identifikován defaultní verzí GLSL 100. Jako výstupní proměnnou fragment shaderu je v GLSL 100 nutné použít pouze vestavěnou proměnnou `gl_FragColor`.

Úloha B12 a B13

Tyto úlohy navazují na předcházející úlohu a k vykreslení trojúhelníku přidávají další parametry – úloha B12 přidává proměnnou s barvou, úloha B13 přidává konstantu s časem vykreslení. Oproti původní verzi této úlohy je nutné upravit tvorbu bufferů a také změnit označení parametrů v shaderech obdobným způsobem jako v předcházející úloze. Po provedení požadovaných změn je úloha na mobilních zařízeních zcela funkční. Použití atributu při vykreslování trojúhelníku probíhá stejným způsobem na obou platformách (v GLSL 100 – značení proměnné `attribute`). V úloze B12 barva jednotlivých vrcholů nejprve beze změny projde vertex shaderem a potom je předána fragment shaderu, kde je interpolována po celém trojúhelníku. Úloha B13 přidává konstantu jejíž hodnota zůstává stejná pro všechny vrcholy v jednom vykreslení, lze tak měnit pozici například pozici vrcholů v závislosti na čase a rozpohybovat tak trojúhelník.



Obr. 6: Úloha B12 – Vykreslení trojúhelníku s atributem

Úloha B14

Úloha B14 vykonává stejné funkce jako úloha B13, tedy vykreslí pohybující se trojúhelník s interpolovanou barvou po celé ploše. Rozdíl oproti předcházející úloze spočívá v přesunutí a zobecnění opakujícího se kódu pro práci s buffery a shaderovým programem do programového balíčku oglutils, který je využit pro ulehčení práce s OpenGL ES v budoucích úlohách. Na rozdíl od předchozí úlohy byl kód vertex a fragment shaderů přesunut do externího souboru a je načítán pomocí oglutils. Oproti původním úlohám bylo v OpenGL ES potřeba při vytvoření OpenGL kontextu předat v parametru konstruktoru vykreslovací třídě `Renderer` odkaz na aktuální aktivitu (`android.app.Activity` - okno v systému Android) a číslo maximální podporované verze OpenGL ES (`int`). Úloha je zcela funkční na zařízení podporujících OpenGL ES 2.0.

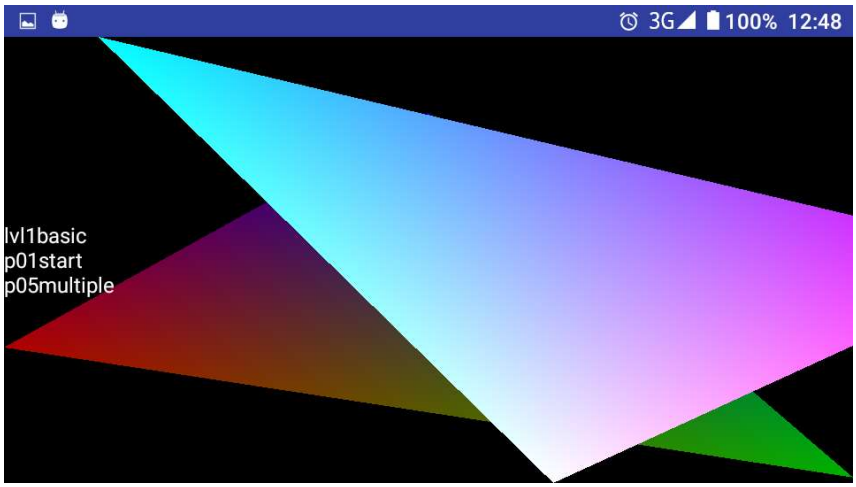
Většina funkcí balíčku oglutils je vytvořena pro práci různými verzemi knihovny OpenGL ES, proto je nutné třídě nějakým způsobem předat aktuálně používanou verzi při jejich volání. Oglutils podporuje OpenGL ES 2.0 a 3.0 s možností jednoduchého rozšíření o budoucí verze této knihovny. Balíček byl testován pouze s verzí 2.0 z důvodu nedostupnosti testovacího zařízení podporující novější verzi OpenGL ES. Některé funkce navíc pro vykonání vyžadují kontext aktuální aktivity

(například funkce pro čtení shaderu ze souboru). Oproti původní verzi ukázek bylo načítání souboru shaderu pozměněno následujícím způsobem.

```
InputStream is = context.getAssets().open(streamFileName);
```

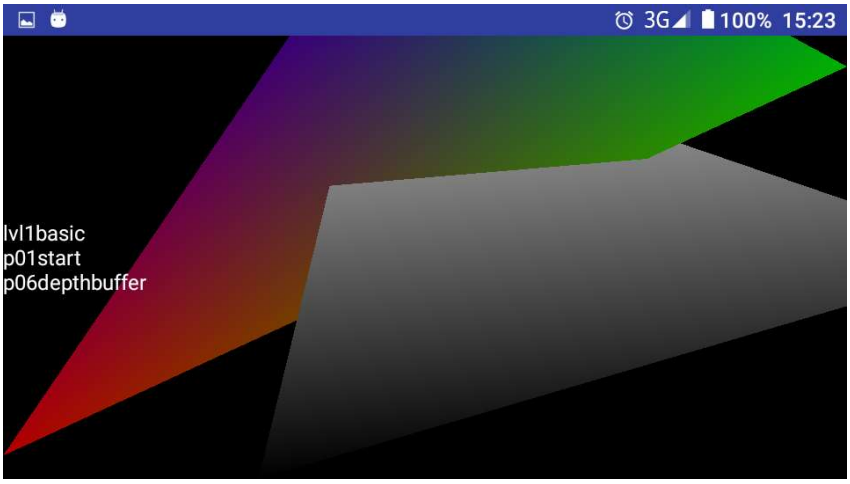
Úloha B15, B16

Během vykreslování lze přepínat mezi různými shaderovými programy, toto je testováno v úloze B15. Stejně jako v OpenGL je přepínání mezi nimi v OpenGL ES realizováno funkcí `glUseProgram(int program)`. Použití několika shaderů umožňuje vykreslovat různé geometrie jinými shaderovými programy s jinými grafickými efekty.



Obr. 7: Úloha B15 – Vykreslení s více shaderovými programy

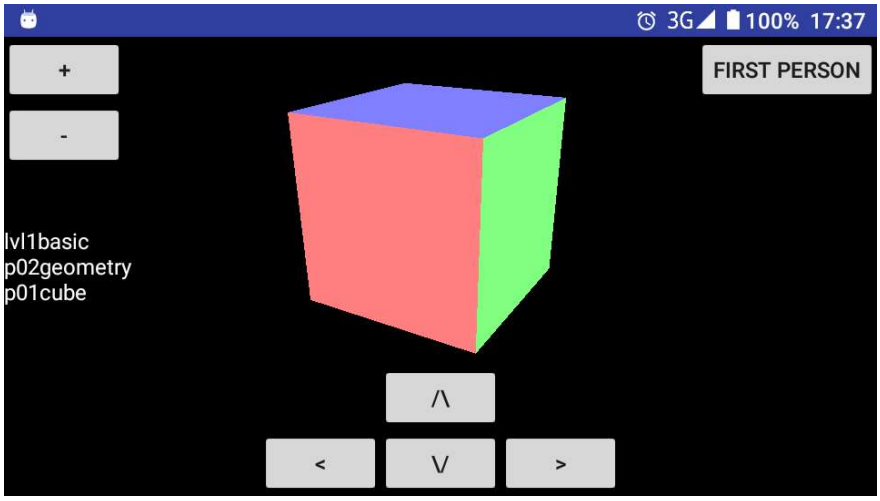
Úloha B16 testuje použití hloubkového testu. Téměř stejným zápisem jako v OpenGL je možné pomocí kódu `GLES20.glEnable(GLES20.GL_DEPTH_TEST);` zaktivovat provádění depth-testu. Obě úlohy jsou plně funkční.



Obr. 8: Úloha B16 – Vykreslení s depth bufferem

Úlohy B21, B22, B23

Následující trojice úloh testuje vykreslení složitější geometrie, módy vykreslování a také načtení geometrie z externího souboru. První z trojice úloh přidává ovladatelnou kameru a vykresluje namísto jednoho trojúhelníku krychli tvořenou trojúhelníky. Oproti původní desktopové úloze je na mobilním zařízení zcela odlišné ovládání. Kameru nelze ovládat myší a klávesami, ale je využit pouze dotykový displej, který se v podstatě chová identicky jako myš jen s jedním tlačítkem. Namísto kláves jsou použity prvky uživatelského rozhraní (`android.widget.Button`). O správu uživatelského rozhraní s ovládacími prvky včetně odchytu událostí s nimi svázanými se stará systém android a probíhá na jiném vlákně než OpenGL ES výpočty.



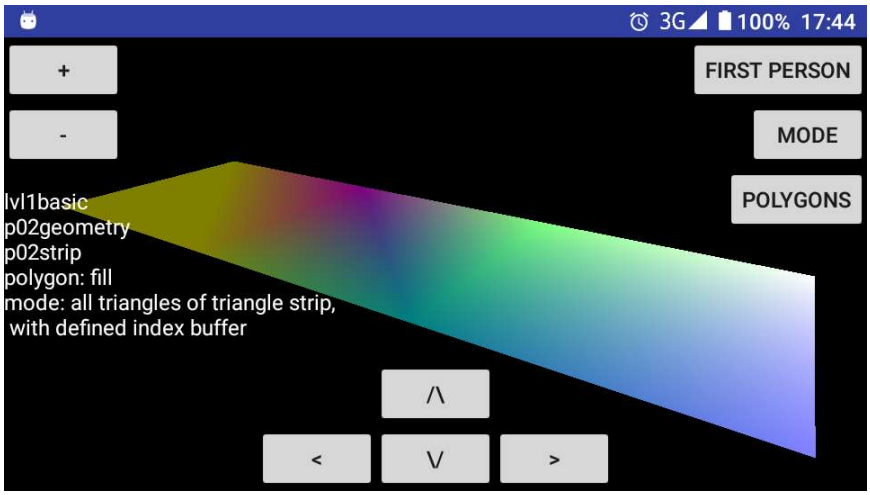
Obr. 9: Úloha B21 – Vykreslení krychle

Všechny reakce na události uživatelského rozhraní (listenery) je možné přepsat vlastní implementací třídy `GLSurfaceView` (v ukázkách třída `MyGLSurfaceView`). Jedná se hlavně o funkci `onTouchEvent`. V kódu ukázek jsou pro přehlednost do této třídy také přesunuty `OnClick` listenery tlačítek uživatelského rozhraní. Pokud je kombinovat práci mezi vláknem OpenGL a GUI vláknem je nutné využít k tomu určené funkce, které se postarají o spuštění kódu na správném vlákně. V úloze je takto spravován obsah `TextView` komponenty přímo z funkce `onDrawFrame` uvnitř třídy `Renderer`. Samotné vykreslení krychle funguje po patřičné úpravě index bufferu stejným způsobem jako v předcházejících úlohách. Úloha je plně funkční pouze s jiným způsobem ovládání.

```
// inside GUI class
queueEvent(new Runnable() {
    @Override
    public void run() {
        // code with OpenGL calls
    }
});
// inside rendering class
this.runOnUiThread(new Runnable() {
     public void run() {
        // code working with UI components
     }
});
```

Druhá z trojice úloh se zaměřuje na vykreslovací módy OpenGL a na rozdíl mezi vykreslením bez a s použitím index bufferu. V původní desktopové verze je možné použít funkci `glPolygonMode` na přepnutí mezi vyplněným a drátovým

módem vykreslení. Tato funkce v OpenGL ES chybí a je možné ji částečně napodobit použitím vykreslování `GL_LINES`, `GL_LINE_LOOP` nebo `GL_LINE_STRIP`. Původní úloha obsahovala pouze dva index buffery, jeden pro vykreslení geometrie pomocí triangle listu a druhý pro triangle strip. Nová úloha je rozšířena o další dva index buffery určených pro vykreslení drátového modelu pomocí line listu a line stripu. Vykreslení odpovídajícího drátového modelu bez index-bufferu by vyžadovalo oddělená data vrcholů odpovídající vykreslení pomocí `GL_LINES`, `GL_LINE_LOOP` nebo `GL_LINE_STRIP` (konkrétní způsob vykreslení by závisel na konkrétní vyžadované geometrii). Finální úloha obsahuje plně funkční vykreslení vyplněného modelu pomocí `GL_TRIANGLES` a `GL_TRIANGLE_STRIP` při použití index bufferu i bez jeho použití. Dále obsahuje funkční vykreslení drátového modelu pomocí `GL_LINES` a `GL_LINE_STRIP` s index bufferem. Jediným nedostatkem je nepřesné vykreslení drátového modelu bez index bufferu.



Obr. 10: Úloha B22 – Vykreslovací módy

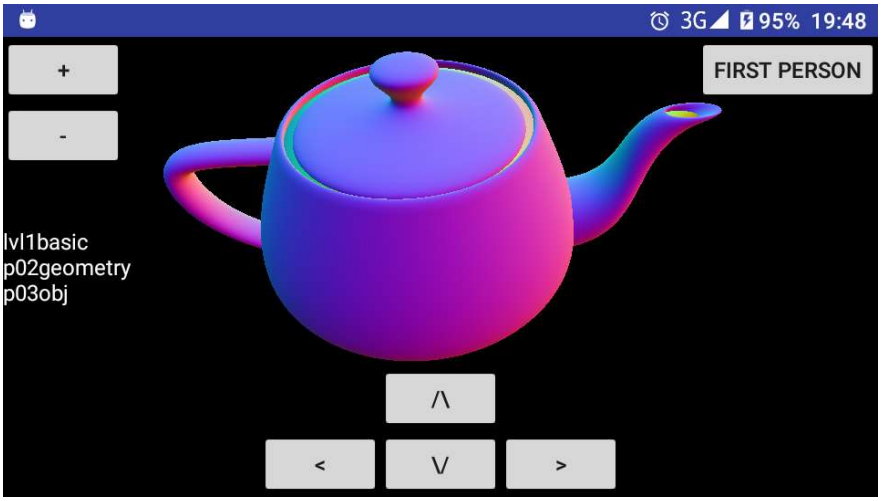
Třetí úloha s označením B23 pracuje s externím souborem s daty modelu s koncovkou .obj. Úloha využívá funkce z balíčku oglutils pro načtení souboru s daty modelu. Načtení souboru je pozměněno na následující kód.

```
InputStream is = context.getAssets().open(modelPath);
```

Úloha funguje správným způsobem, ale při větším objemu dat v načítaném souboru je znatelná výrazná prodleva před prvním vykreslením modelu.

Tab. 8: Vykreslení větších modelů

Model	Prodleva	Počet plošek	Počet vrcholů na plošku	Počet parametrů vrcholů
Krychle	<0.001 s	12	3	3
Kachna	7 s	14128	3	2
Slon	27 s	13714	3	3
Čajová konvice	83 s	39800	3	3



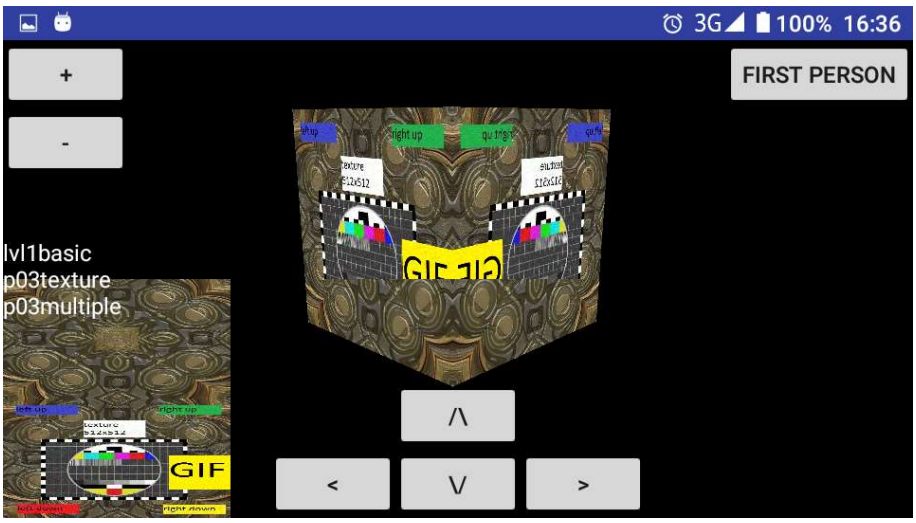
Obr. 11: Úloha B23 – Vykreslení modelu ze souboru

Úloha B31, B32

První ze dvojice úloh (B31) testuje načítání textury ze souboru a její aplikování na existující geometrii. Knihovna OpenGL ES přímo nepodporuje formáty jako JPEG nebo PNG. Android však poskytuje funkci pro načtení dat do OpenGL textury pomocí objektu `Bitmap`. Stačí tedy převést data textury pomocí třídy `BitmapFactory` a poté zavolat funkci `texImage2D`. Podrobný popis se nachází v kapitole 4.4.5. Úloha plně funguje na OpenGL ES – podařilo se vykreslit texturovanou krychli. Úloha B32 provádí stejné funkce jako předchozí úloha za pomocí oglutils. Oglutils navíc umožňují zvolit vnitřní formát ve kterém OpenGL uchová data textury a typ vstupních dat. V OpenGL ES 2.0 si musí vnitřní formát a typ odpovídat. Druhá úloha podle očekávání funguje stejným způsobem.

Úloha B33

Tato úloha testuje aplikaci dvou různých textur na jeden objekt. Ve vrchní polovině obrazovky je aplikována jedna textura a ve spodní polovině je aplikována druhá textura. Při přepsání shaderů na verzi GLSL 100 nastal problém ve fragment shaderu při dělení `height/2`, kdy proměnná `height` byla typu `float` (chyba dělení `float/int` nelze). GLSL 100 nepodporuje implicitní konverze datových typů, proto musí být konverze vyvolána explicitně pomocí `height/float(2)`. Po změnách viděných na předchozích úlohách a úpravě dělení je tato úloha plně funkční. V úloze byli testováno načtení formátů PNG, JPG, GIF, BMP.



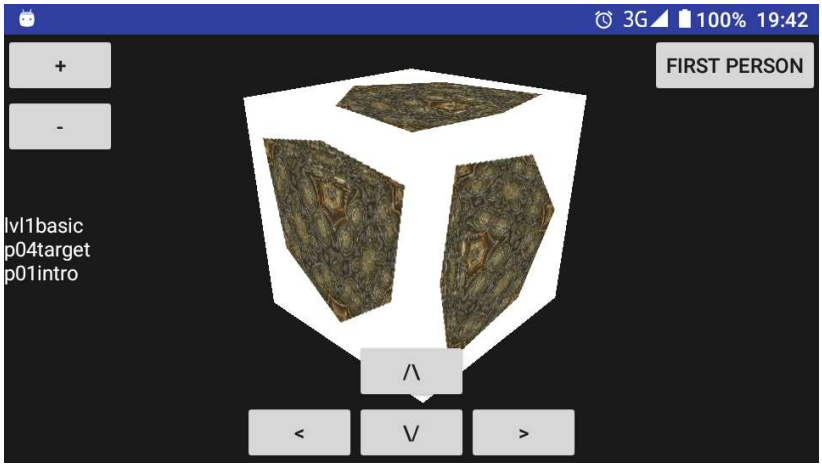
Obr. 12: Úloha B33 – Vykreslení dvou textur na jeden objekt

Úloha B41, B42

Úloha B41 testuje vykreslení do textury místo na obrazovku. Tato technika vyžaduje vytvoření oddělené textury a použití této textury jako cíle vykreslovacích příkazů OpenGL. Takto vykreslenou scénu lze poté znovu vykreslit a případně aplikovat nějaký druh grafických efektů. Lze takto několikanásobným průchodem aplikovat na scény post-processingové efekty. V úloze je nejdříve vykreslena texturovaná krychle, ale místo zobrazení na obrazovku je uložena do objektu textury. Scéna je pak vykreslena znovu, ale místo původní textury je na krychli aplikován nově vytvořený objekt textury. Celá scéna je tak vykreslena několikrát na každé straně krychle. Problém nastává při vykreslení depth bufferu do textury. OpenGL ES 2.0 v základu nepodporuje takovéto použití textury a je vyžadováno

rozšíření `OES_depth_texture`, které však nemusí být přítomné na starších zařízeních s OpenGL ES 2.0. U starších zařízení lze využít RenderBuffer, který však slouží k trochu jiným účelům.

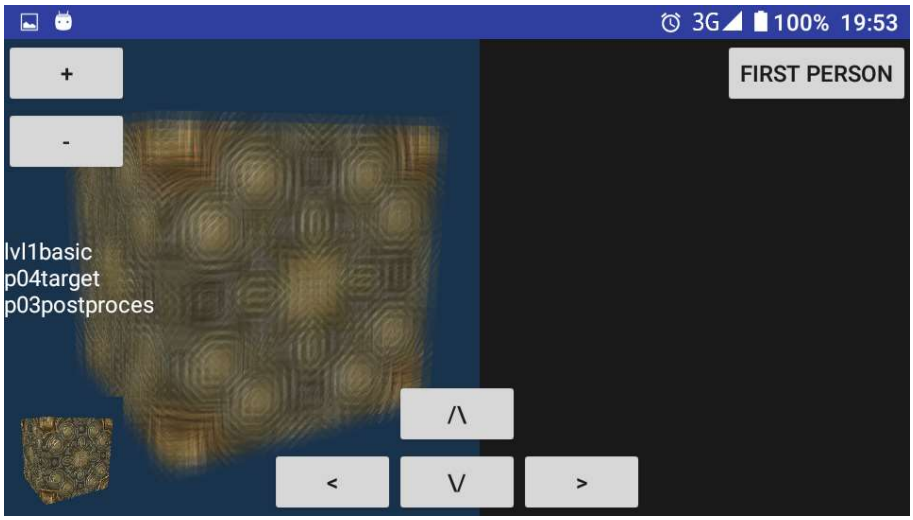
Úloha B42 přesouvá funkcionalitu v předchozí úloze do balíčku oglutils. Obě úlohy jsou funkční.



Obr. 13: Úloha B41 – Vykreslení do textury

Úloha B43

Tato úloha testuje post-procesingové efekty s využitím vykreslení do textury a následným průchodem druhým shaderem. Na mobilním zařízení se podařilo docílit rozmazání i převodu na stupně šedi stejným způsobem jako na desktopové verzi úloh, pouze se změněnou syntaxí.



Obr. 14: Úloha B43 – Post-processing

5.2.2 Advanced úlohy

Advanced ukázky testují pokročilé grafické efekty včetně využití compute, geometry a obou tessellation shaderů. V následující tabulce je znázorněn seznam testovaných ukázek a jejich proveditelnost s knihovnou OpenGL ES 2.0.

Tab. 9: Funkčnost advanced ukázek

Označení – Název Ukázky	**OpenGL ES 2.0**
A10 – Použití knihovny JOGL	Netestováno
A20 – Debugovací nástroje *uhk.android_samples.lvl2advanced.p02debug*	Částečně funkční
A31 – Testování editace textur *uhk.android_samples.lvl2advanced.p03texture.p01quad*	Částečně funkční
A32 – Krychlová textura *uhk.android_samples.lvl2advanced.p03texture.p02cubetexture*	Funkční
A33 – Objemová textura	Nefunkční
A34 – Filtrace textur, mipmap *uhk.android_samples.lvl2advanced.p03texture.p04filtering*	Částečně funkční
A41 – Uložení textury do souboru *uhk.android_samples.lvl2advanced.p04target.p01save*	Funkční
A42 – Editace dat textury *uhk.android_samples.lvl2advanced.p04target.p02draw*	Částečně funkční
A43 – Editace dat textury pomocí BufferedImage	Nefunkční
A44 – Vykreslení do více textur	Nefunkční
A45 – GPGPU *uhk.android_samples.lvl2advanced.p04target.p05gpgpu*	Funkční
A51 – Geometry shader	Nefunkční
A52 – Teselační shader	Nefunkční
A53 – Query	Nefunkční
A61 – Compute shader	Nefunkční
A62 – Hledání min. klíčové hodnoty pomocí compute shaderu	Nefunkční
A63 – Načtení textur do výstupního obrázku	Nefunkční
A64 – Compute shader ukázka hry	Nefunkční
A71 – Transform feedback z vertex shaderu	Nefunkční
A72 – Transform feedback z geometry shaderu	Nefunkční

Úloha A10

Tato úloha testuje použití knihovny JOGL s různými druhy práce s uživatelským prostředím jako je AWT, Swing nebo NEWT. Úloha nebyla testována z důvodu nedostatečné dokumentace použití knihovny JOGL na systému Android.

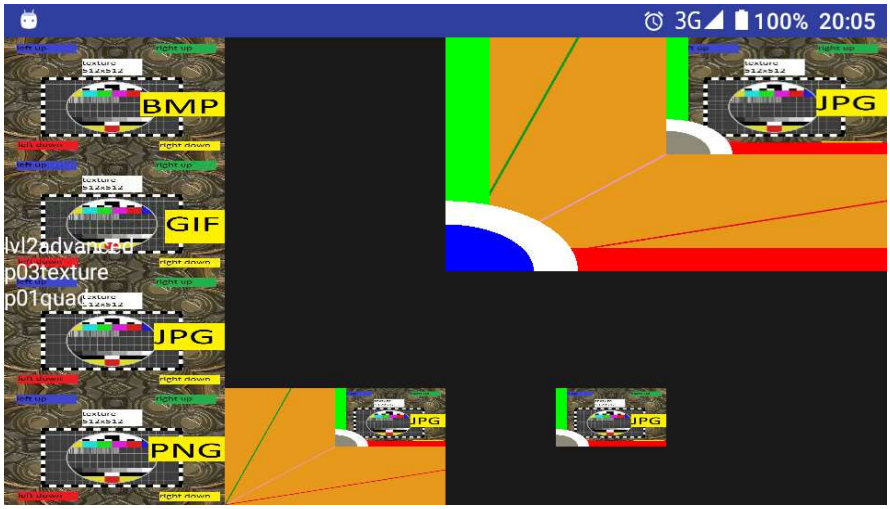
Úloha A20

Tato úloha testuje debugovací nástroje. V původních úloze jsou k dispozici tři debugovací módy: Debug, Trace a Individual. Mód Individual používá funkci `glError` po každém volání GL metod. Běžné použití této funkce je na konci vykreslovací smyčky nebo na místě, kde je očekávána chyba. Funkce vrací první chybu ve frontě všech chyb, proto by měla být volána ve smyčce, dokud nevrátí `GL_NO_ERROR`. Mód Debug mód využívá třídy `DebugGL2`, která obaluje volání GL zjišťování chyb, pokud nastane chyba vyvolá v místě chyby výjimku. Poslední z módů – Trace vypisuje jména volaných GL metod. Dva z těchto módů jsou součástí knihovny JOGL, která není v nových ukázkách využita, proto v nich funguje pouze mód Individual. Mód Trace lze nahradit na některých zařízeních v „Nastavení→Pro vývojáře→Povolit trasování OpenGL→Logcat“.

Úloha A31

Tato úloha testuje získání dat OpenGL textury, editaci těchto dat a následné uložení upravených dat zpět do původní OpenGL textury. Knihovna OpenGL ES 2.0 postrádá funkci `glGetTexImage` používanou desktopovým OpenGL k získání dat z připojené OpenGL textury. Jako náhradu je možné použít funkci `glReadPixels`, která vrací podobné informace, ale z frame buffer objektu (objekt využívaný pro vykreslování bez narušení aktuálně zobrazené obrazovky). V OpenGL ES 2.0 nelze přímo číst data textury, proto je textura nejdříve vykreslena do frame buffer objektu, poté čtena funkcí `glReadPixels` a nakonec je vykreslení přepnuto zpět do defaultního vykreslovacího objektu zobrazovaného na obrazovku. V následujícím úryvku kódu jsou do proměnné buffer načtena data textury.

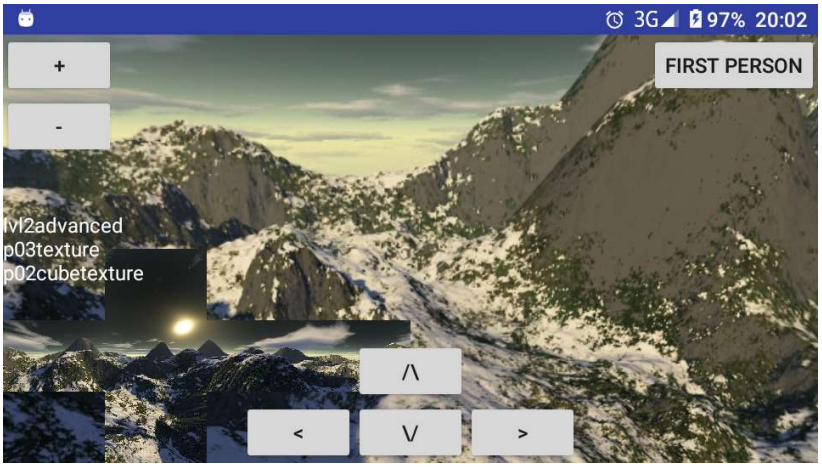
```
int[] frameBufferID = new int[1];
GLES20.glGenFramebuffers(1, frameBufferID, 0);
GLES20.glBindFramebuffer(GLES20.GL_FRAMEBUFFER, frameBufferID[0]);
GLES20.glFramebufferTexture2D(GLES20.GL_FRAMEBUFFER,
     GLES20.GL_COLOR_ATTACHMENT0, GLES20.GL_TEXTURE_2D,
     textureID[0], level);
GLES20.glReadPixels(0, 0, width, height, pixelFormat,
     pixelType, buffer);     //0 - default framebuffer
GLES20.glBindFramebuffer(GLES20.GL_FRAMEBUFFER, 0);
```



Obr. 15: Úloha A31 – Editace textur

Úloha A32, A33

První úloha (A32) testuje vykreslení cube map textury. Cube map textura se skládá ze 6 obrázků tvořících krychli. Druhá úloha (A33) testuje použití 3D (volume) textury. OpenGL ES 2.0 nepodporuje vykreslení 3D textur (pouze na vybraných zařízeních) proto byla zprovozněna pouze první z těchto úloh. Dalším zjištěným nedostatkem je chybějící parametr `GL_TEXTURE_WRAP_R`, sloužící k nastavení chování textury při použití texturových souřadnic mimo hodnoty 0…1 na ose R (třetí rozměr texturových souřadnic).



Obr. 16: Úloha A32 – Aplikace cube map textury

Úloha A34

Další z úloh testuje filtrační módy textur, druhy interpolace parametrů v shaderu a chování textury při použití texturových souřadnic mimo 0…1. Také je zde testován zápis a vykreslení jednotlivých levelů MIP map textury. Filtrační módy určují chování textury při zvětšování/zmenšování například při zobrazení textury ve větší vzdálenosti a jsou totožné v OpenGL i OpenGL ES. Módy chování textury při souřadnicích mimo 0…1 jsou v OpenGL ES pouze tři a to `GL_CLAMP_TO_EDGE`, `GL_REPEAT` a `GL_MIRRORED_REPEAT`. Chybí například `GL_CLAMP_TO_BORDER`.

Podporované filtry:

`GL_TEXTURE_MIN_FILTER` (zmenšení textury)

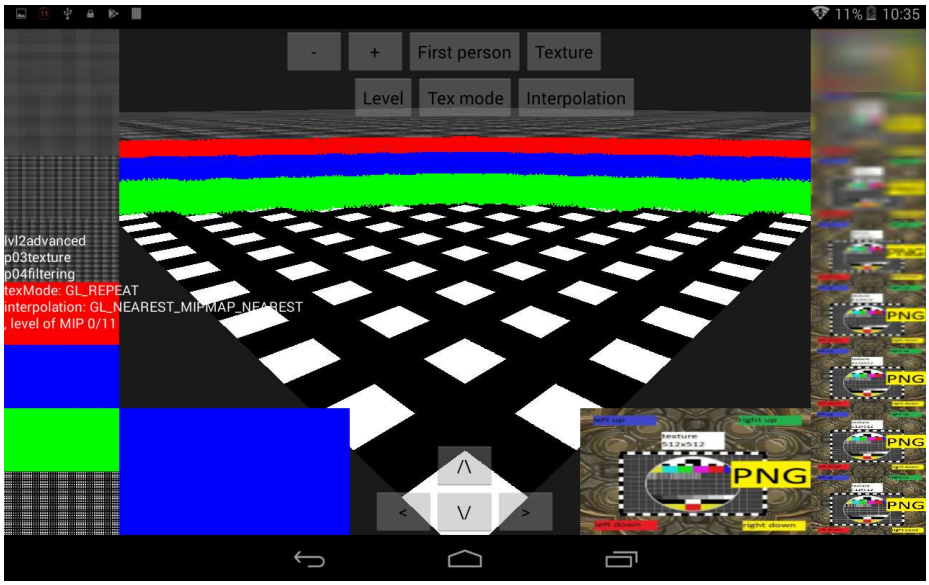
- `GL_NEAREST`
- `GL_LINEAR`
- `GL_NEAREST_MIPMAP_NEAREST`
- `GL_LINEAR_MIPMAP_NEAREST`
- `GL_NEAREST_MIPMAP_LINEAR`
- `GL_LINEAR_MIPMAP_LINEAR`

`GL_TEXTURE_MAG_FILTER` (zvětšení textury)

- `GL_NEAREST`
- `GL_LINEAR`

V desktopovém OpenGL je možné pomocí interpolační klasifikátorů flat (žádná interpolace), noperspective (lineární interpolace) a smooth (perspektivně korektní interpolace) ve fragment shaderu určit druh interpolace takto označených proměnných. V OpenGL ES 2.0 (GLSL 100) toto není možné, protože podle

specifikace musí být všechny varying proměnné fragment shaderu interpolovány perspektivně korektní interpolací.



Obr. 17: Úloha A34 – Filtrace textur, mipmap

Práce s levely MIP map je v OpenGL ES ve fragment shaderu možná pouze pomocí rozšíření `GL_EXT_shader_texture_lod`. V úloze jsou zobrazeny jednotlivé levely MIP map a lze mezi nimi přepínat právě díky tomuto rozšíření. Pro použití rozšíření je nutné ve fragment shaderu označit jeho použití pomocí `#extension GL_EXT_shader_texture_lod : enable`. Po aktivaci je možné použít funkci `texture2DLodEXT` k získání dat požadovaného levelu MIP map.

Úloha A41

Tato úloha využívá vykreslení barvy a hloubky do textury a testuje uložení textury do souboru. Pomocí objektu bitmap lze data textury kompresovat například do formátu PNG. Následuje zkrácený postup vytvoření a naplnění souboru v systému android. Je důležité, aby objekt Bitmap měl definovány rozměry odpovídající velikosti textury a formát dat Bitmapu.

```
Bitmap bitmap =
     Bitmap.createBitmap(width,height,Bitmap.Config.ARGB_8888);
bitmap.copyPixelsFromBuffer(dataBuffer);
File directory = context.getDir("OGLimages", Context.MODE_PRIVATE);
// Create directory
File file=new File(directory,fileName);
FileOutputStream fos = new FileOutputStream(file);
// compress BitMap
bitmap.compress(Bitmap.CompressFormat.PNG, 100, fos);
fos.close();
```

Úloha A42, A43, A44

První z úloh využívá vykreslení barvy a hloubky do textury a testuje editaci této textury a zobrazení výsledku. OpenGL ES 2.0 v základu neobsahuje podporu práce s hloubkovou texturou, proto byla editace úspěšná pouze u barevné části textury, a to obdobným způsobem jako v úloze A31. Úloha A43 testuje editaci textury pomocí tříd `BufferedImage` a `Graphics` z knihovny AWT. Android nemá knihovnu AWT, proto úloha nebyla zprovozněna. Úloha A44 testuje výstup z fragment shaderu do více textur. Toto v OpenGL ES není možné z důvodu chybějícího kvalifikátoru layout umožňujícího specifikovat více výstupních proměnných.

Úloha A45

Tato úloha testuje použití grafického procesoru k výpočtu běžných algoritmů. OpenGL ES lze k těmto účelům využít, avšak omezený výkon většiny zařízení a úloha běží bez větších problémů.

Nefunkční úlohy

Značná část úloh je nefunkčních z důvodu chybějících shaderů v knihovně OpenGL ES 2.0. Jsou to úlohy A51, A52 – chybějící geometry a tessellation shadery (přidány v OpenGL ES 3.2), úlohy A61 až A64 – chybějící compute shader (přidán v OpenGL ES 3.1) a úlohy A71, A72 z důvodu nedostupnosti transform feedbacku (přidán v OpenGL ES 3.0). Úloha A53 testuje použití occlusion query pro vypsání počtů právě renderovaných prvků a primitivů. OpenGL ES umožňuje práci s occlusion query pomocí rozšíření EXT_occlusion_query_boolean. Testovací zařízení neobsahují toto rozšíření, proto nebyla úloha testována.

6 Srovnání výkonů mobilních a desktopových aplikací

6.1 Test omezení

Test omezení zkoumá hodnoty implementačně závislých proměnných omezujících možnosti práce s knihovnou OpenGL ES. Vysvětlení významu proměnných se nachází v Tab. 6. V Tab. 10 lze vidět, že možnosti testovaného tabletu a mobilního telefonu jsou identické. V porovnání PC nabízí práci s mnohem větším počtem textur a také umožňuje vykreslovat textury s několikanásobně většími rozměry.

Tab. 10: Hodnoty implementačně-závislých proměnných

Název proměnné	Min.	Mobil	Tablet	PC
GL_MAX_VERTEX_ATTRIBS	8	16	16	16
GL_MAX_VERTEX_UNIFORM_VECTORS	128	256	256	1024
GL_MAX_VARYING_VECTORS	8	12	12	16
GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS	0	0	0	32
GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS	8	8	8	192
GL_MAX_TEXTURE_IMAGE_UNITS	8	8	8	32
GL_MAX_FRAGMENT_UNIFORM_VECTORS	16	256	256	1024
GL_MAX_RENDERBUFFER_SIZE	1	4096	4096	16384
GL_MAX_CUBE_MAP_TEXTURE_SIZE	16	4096	4096	16384
GL_MAX_TEXTURE_SIZE	64	4096	4096	16384
GL_MAX_VIEWPORT_DIMS		4096	4096	16384

6.2 Porovnání výkonů

Desktopové aplikace díky konstrukci desktopů mohou dosahovat mnohem větších výkonů než mobilní aplikace při vykreslování grafických objektů s obrovským množstvím vrcholů. Je pravidlem, že mobilní aplikace nepotřebují tak velké množství vrcholů k dosažení stejné kvality díky menším displejům. Při porovnání běžných aplikací s malým množstvím vykreslovaných vrcholů bude OpenGL ES pravděpodobně o dost rychlejší. Běžné desktopové aplikace s menším množstvím vykreslované grafiky totiž zpravidla nejsou tolik optimalizovány, protože díky výkonu desktopů to není potřeba. Při větším počtu vrcholů se objevují limity OpenGL ES 2.0. Maximální počet vykreslitelných vrcholů při jednom volání je totiž omezen datovým typem indexu. Index vrcholu může v OpenGL ES 2.0 mít pouze

datový typ `GL_UNSIGNED_SHORT`, s maximální hodnotou 65,535. V praxi sice neznamená, že nelze vykreslit vrcholů více (indexy mohou být při vykreslení použity vícekrát), ale není možné, aby hodnota indexu přesáhla požadovanou hodnotu 65,535. Vykreslování většího množství geometrie musí proto být děleno do jednotlivých vykreslovacích volání.

7 Shrnutí výsledků

Při zpracování této práce byla zjištěna funkčnost běžných grafických operací a omezení knihovny OpenGL ES 2.0 na platformě Android. Testováním byly zjištěny nedostatky knihovny ve verzi 2.0 zejména v oblasti zpracování textur. Knihovna plně podporuje pouze práci s 2D a cube map texturami. Hlavním nedostatkem této verze knihovny je nepřístupnost programovatelných tessellation, geometry a compute shaderů, které však byli přidány do pozdějších verzí knihovny. Dále byla zjištěna podpora práce s buffery a jejich vykreslení pomocí vertex a fragment shaderů. V OpenGL lze vykreslovat do textury a využít více průchodů pro dosažení post-processingových efektů.

8 Závěry a doporučení

Práce se zabývá možnostmi mobilních zařízení s operačním systémem Android se zaměřením na využití programovatelného zobrazovacího řetězce. Je podrobně popsána knihovna OpenGL a její specifika pro mobilní zařízení ve verzi OpenGL ES. Nejdříve je porovnávána dostupná funkcionalita knihovny OpenGL ES s desktopovou verzí OpenGL a poté jsou implementovány ukázky testující tuto funkcionalitu. Testování mimo jiné zahrnovalo testy vykreslení geometrie pomocí shaderů, práci s texturami nebo zjištění dostupných shaderů. Práce také obsahuje postupy tvorby grafických aplikací systému Android a principy používané při zobrazování grafiky. Dále byli testovány hodnoty omezující práci na běžných mobilních zařízení v porovnání s desktopy.

Grafické možnosti značné části dnes používaných mobilních zařízení jsou stále omezeny. Na většině zařízení není možné využívat teselace či geometrických shaderů. V OpenGL ES 2.0 jsou oproti desktopovým systémům velké nedostatky v pokročilejší grafické funkcionalitě. Tyto nedostatky jsou však stále snižovány a s nejnovější verzí této grafické knihovny se již mobilní platforma blíží možnostem desktopů. Stále však existují omezení paměti a výkonu. Menší velikost mobilních zařízení naštěstí dokáže dosáhnout podobné kvality jako desktopové systémy s mnohem menším výkonem. OpenGL ES 2.0 podporuje všechny potřeby základní grafické aplikace a vzhledem k zpětné kompatibilitě současných verzí OpenGL ES lze dále používat tuto verzi jak na starších, tak i na novějších zařízeních.

Hlavní omezení ve využití OpenGL na mobilních zařízení se ukázala při práci s texturami, kde byla znatelná hlavně chybějící podpora hloubkových textur nebo 3D textur. OpenGL ES 2.0 nabízí pouze omezenou práci s texturami, které nejsou „power of two“ – například chybějící podpora mipmap a pouze jeden wrap mód (`GL_CLAMP_TO_EDGE`).

Tato práce by mohla být bez větších potíží rozšířena použitím novější verze knihovny OpenGL ES. S odpovídajícím testovací zařízením by bylo možné otestovat úlohy, které byli nefunkční s verzí OpenGL ES 2.0.

9 Seznam použité literatury

[1] POSLAD, Stefan. Ubiquitous Computing: Basics and Vision. In: *Ubiquitous Computing* [online]. B.m.: John Wiley & Sons, Ltd, 2009 [vid. 2018-02-18], s. 1–40. ISBN 978-0-470-77944-6. Dostupné z: doi:10.1002/9780470779446.ch1

[2] IDC. *Smartphone OS Market Share, 2017 Q1* [online]. [vid. 2018-02-17]. Dostupné z: https://www.idc.com/promo/smartphone-market-share/os

[3] BERND VAN DER WIELEN. Insights into the 2.3 Billion Android Smartphones in Use. *Newzoo* [online]. 17. leden 2018 [vid. 2018-07-08]. Dostupné z: https://newzoo.com/insights/articles/insights-into-the-2-3-billion-android-smartphones-in-use-around-the-world/

[4] GOOGLE. Android Developers. *Dashboards* [online]. [vid. 2018-02-21]. Dostupné z: https://developer.android.com/about/dashboards/index.html

[5] OpenGL ES. *Android Developers* [online]. [vid. 2018-07-14]. Dostupné z: https://developer.android.com/guide/topics/graphics/opengl

[6] Android 2.2 Platform Highlights. *Android Developers* [online]. [vid. 2018-07-14]. Dostupné z: https://developer.android.com/about/versions/android-2.2-highlights

[7] Codenames, Tags, and Build Numbers. *Android Open Source Project* [online]. [vid. 2018-07-14]. Dostupné z: https://source.android.com/setup/start/build-numbers

[8] Android 7.0 for Developers. *Android Developers* [online]. [vid. 2018-07-14]. Dostupné z: https://developer.android.com/about/versions/nougat/android-7.0

[9] Jelly Bean. *Android Developers* [online]. [vid. 2018-07-14]. Dostupné z: https://developer.android.com/about/versions/jelly-bean

[10] KHRONOS GROUP. OpenGL. *OpenGL Overview* [online]. [vid. 2018-02-21]. Dostupné z: https://www.opengl.org/about/

[11] KHRONOS GROUP. The Khronos Group. *Khronos Members* [online]. 21. únor 2018 [vid. 2018-02-21]. Dostupné z: https://www.khronos.org/members/list

[12] BENSON TAO. Understand the mobile graphics processing unit. *Embedded Computing Design* [online]. 16. září 2014 [vid. 2018-07-15]. Dostupné z: http://www.embedded-computing.com/embedded-computing-design/understand-the-mobile-graphics-processing-unit

[13] OpenGL ES - The Standard for Embedded Accelerated 3D Graphics. *The Khronos Group* [online]. 19. červenec 2011 [vid. 2018-08-08]. Dostupné z: https://www.khronos.org/opengles/

[14] LEE, Hwanyong a Nakhoon BAEK. Implementing OpenGL ES on OpenGL. *Semantic Scholar* [online]. 2009 [vid. 2018-08-08]. Dostupné z: https://pdfs.semanticscholar.org/1675/7b4ea7495c1c916b7eb9383782f736145a7e.pdf

[15] KHRONOS GROUP. The Khronos Group. *Khronos OpenGL ES Registry* [online]. [vid. 2018-02-24]. Dostupné z: https://www.khronos.org/registry/OpenGL/index_es.php

10 Přílohy

1) CD obsahující
    - Text práce ve formátu .docx
    - Text práce ve formátu PDF
    - Projekt Android_samples.zip
    - Adresář Android_samples s ukázkami ve formátu .apk

Univerzita Hradec Králové
Fakulta informatiky a managementu
Akademický rok: 2017/2018

Studijní program: Aplikovaná informatika
Forma: Prezenční
Obor/komb.: Aplikovaná informatika (ai3-p)

Podklad pro zadání BAKALÁŘSKÉ práce studenta

PŘEDKLÁDÁ:	ADRESA	OSOBNÍ ČÍSLO
Kolomazník Michal	Lesná 69, Sosnová	I1500378

TÉMA ČESKY:

Grafická knihovna OpenGL ES na platformě Android

TÉMA ANGLICKY:

Graphics library OpenGL ES on Android platform

VEDOUCÍ PRÁCE:

Ing. Bruno Ježek, Ph.D. - KIKM

ZÁSADY PRO VYPRACOVÁNÍ:

Cíl: Prozkoumat oblast vývoje mobilních grafických aplikací s využitím grafické knihovny OpenGL ES a hardwarově podporovaným programovatelným zobrazovacím řetězcem.
Osnova:
1) Prozkoumat a popsat možnosti využití programovatelných grafických karet na mobilních zařízeních.
2) Zaměřit se na specifické vlastnosti knihovny OpenGL ES a rozdíly oproti standardní knihovně OpenGL.
3) Prozkoumat a popsat způsob vývoje grafické aplikace na platformě Android.
4) Navrhnout vhodné příklady pro znázornění omezení a specifik mobilní platformy a implementovat je.
5) Provést srovnání z hlediska výkonu a možností oproti desktopovým aplikacím.
6) Zhodnotit dosažené výsledky.

SEZNAM DOPORUČENÉ LITERATURY:

OpenGL ES Overview [online] https://www.khronos.org/opengles/

Lee, H. and N. Baek. Implementing OpenGL ES on OpenGL. in 13th International Symposium on Consumer Electronics. 2009 [online] https://pdfs.semanticscholar.org/1675/7b4ea7495c1c916b7eb9383782f736145a7e.pdf

OpenGL ES Specifications and Documentation [online] https://www.khronos.org/registry/OpenGL/index_es.php

Podpis studenta: Kolomazník ..... **Datum:** 20.12.2017

Podpis vedoucího práce: ..... **Datum:** 20.12.2017