VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

KNIHOVNA PRO RENDEROVÁNÍ OPENSTREETMAP NA MOBILU

LIBRARY FOR OPENSTREETMAP RENDERING ON SMARTPHONES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **DAVID VAĎURA**
AUTHOR

VEDOUCÍ PRÁCE **prof. Ing. ADAM HEROUT, Ph.D.**
SUPERVISOR

BRNO 2017

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Vaďura David**
Obor: Informační technologie
Téma: **Knihovna pro renderování OSM na mobilu**
Library for OSM Rendering on Smartphones
Kategorie: Počítačová grafika

Pokyny:

1. Prostudujtea popište problematiku zobrazování map na mobilních aplikacích. Zaměřte se na práci s vektorovými mapami.
2. Prostudujte a popište Open Street Maps a související nástroje použitelné na mobilních zařízeních.
3. Definujte funkčnost přenositelného systému pro zobrazení vektorové reprezentace OSM na mobilních zařízeních.
4. Implementujte navrženou funkčnost a testujte je na mobilních zařízeních různých platforem.
5. Vytvořte demonstrační aplikaci či aplikace a předveďte funkčnost vytvořeného řešení.
6. Zhodnoťte dosažené výsledky a navrhněte možnosti pokračování projektu; vytvořte plakátek a krátké video pro prezentování projektu.

Literatura:

- dle pokynů vedoucího

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3, značné rozpracování bodu 4.

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

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

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

Vedoucí: **Herout Adam, prof. Ing., Ph.D.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 17. května 2017

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

L. S.

doc. Dr. Ing. Jan Černocký
vedoucí ústavu

Abstrakt

Cílem mé práce bylo navrhnout a implementovat knihovnu pro vykreslování vektorových map na mobilních telefonech. Zdrojem dat pro knihovnu je projekt OpenStreetMap. Knihovna musí zahrnovat tyto funkce: načítání a cachování dlaždic, podpora online a offline map, vykreslování všech druhů geometrie a možnost definovat vlastní styl zobrazené mapy. Navržená architektura využívá hybridní přístup pro zobrazení mapy. Základní prvky jsou po načtení vykreslené do textury. Vrstvy obsahující text nebo ikony jsou vykreslené v reálném čase nad bázovou mapou. Díky tomu se mohou otáčet a text je čitelný i po přiblížení. Výsledná knihovna byla implementovaná v programovacím jazyce C++ s využitím OpenGL ES pro hardwarově akcelerované vykreslování. Funguje na platformách iOS a Android a je navržená takovým způsobem, aby byla jednoduše použitelná pro jedince i firmy.

Abstract

The aim of my work was to design and implement a vector map rendering library for mobile phones. The source of the data for the library is the project OpenStreetMap. The library must include these functions: loading and caching of tiles, support for online and offline maps, rendering of all geometry types and possibility to define custom map style. The proposed architecture uses a hybrid approach to presenting the map. Base features are rendered into texture upon loading them. Layers containing text or icons are rendered in real-time above the base map. This enables them to rotate and it ensures that the text stays readable even when zoomed in. The resulting library was implemented in C++ using OpenGL ES for hardware accelerated rendering. It works on both iOS and Android, and it is designed in such a way, that it can be easily used by individuals and companies.

Klíčová slova

mapa, vykreslování, OpenGLES

Keywords

map, rendering, OpenGLES

Citace

VAĎURA, David. *Knihovna pro renderování OpenStreetMap na mobilu.* Brno, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Herout Adam.

Knihovna pro renderování OpenStreetMap na mobilu

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením prof. Adama Herouta. Další informace mi poskytl Ing. Vladislav Skoumal. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
David Vaďura
13. května 2017

Poděkování

Chtěl bych poděkovat svému vedoucímu práce prof. Adamu Heroutovi, za jeho pomoc.

Také bych chtěl poděkovat Ing. Vladislavu Skoumalovi, se kterým jsem v průběhu práce konzultoval.

Obsah

Kapitola 1

Úvod

Mapy jsou dnes součástí mnoha mobilních aplikací, které denně použijí stovky tisíc uživatelů. Ačkoliv si to často neuvědomujeme, vykreslit dobře vypadající mapu na pomalém zařízení typu mobilní telefon tak, aby ji zároveň mohl uživatel pohodlně ovládat, je jeden z nejobtížnější úkolů před které se mohou vývojáři postavit. Naštěstí, stejně jako mnoha jiných úloh, i pro vykreslování map existují různé knihovny, které můžeme použít. U mnoha z nich jsem však časem zjistil že úplně nevyhovují mým požadavkům a to také vedlo ke vzniku mého tématu bakalářské práce.

Mým cílem bylo vytvořit knihovnu pro vykreslování map na mobilních telefonech, která bude zároveň splňovat několik požadavků:

- Zobrazení vektorových map.
- Práce s online i offline zdroji map.
- Komplexní možnosti definice stylu výsledné mapy.
- Podpora pro systémy Android a iOS.

Jako implementační jazyk jsem zvolil C++. Pro multiplatformní vývoj je tento jazyk výborná volba z několika důvodů. Obě cílové platformy zahrnují nástroje pro kompilaci, ladění a profilování kódu napsaného v C++. Díky tomu je velké množství kódu napsáno jen jednou a je potřeba jen dopsat patřičné „obalové“ třídy pro každou platformu, aby mohli další programátoři použít knihovnu z jazyka který se na dané platformě používá (Java, Objective-C, Swift atd.).

Nejsložitější část této práce bylo vykreslování, což je také část, kterou vidí cílový uživatel a proto musí vypadat výsledek dobře. Na mobilních zařízeních (ale ne jen tam) je velmi populární rozhraní OpenGL, respektive jeho mobilní verze OpenGL ES. Protože je podporované na obou platformách a existuje velké množství materiálů, které se OpenGL věnují, zvolil jsem právě toto API. Jak probíhá vykreslování, je podrobně popsáno v kapitole 3.

1.1 Projekt OpenStreetMap

Spolu s klesající cenou zařízení s podporou GPS se začala objevovat potřeba kvalitních mapových materiálů. Protože byl proces vytváření podkladů zpočátku velmi nákladný, většina z nich byla proprietární a nedostupná pro běžné vývojáře. To se změnilo v roce 2004, kdy Steve Cost založil projekt OpenStreetMap [10]. Podstata tohoto projektu je kolaborace

mezi jeho uživateli; každý může do map přidávat vlastní naměřená data a aktualizovat ta existující. Výstupem projektu jsou kromě vyrenderované mapy i podkladová data, která mohou být následně využita v dalších projektech.

Kromě mapy na webových stránkách OSM [23] jsou k dispozici ke stažení samotné podklady, které mapu tvoří. Základní formát, ve kterém jsou nabízené, je XML, které se k aplikaci, která zobrazuje mapu v reálném čase nehodí; k dispozici jsou ale i ve formátu PBF. Detaily různých formátů a jejich výhody a nevýhody jsou v sekci 2.3.

1.2 Existující řešení

Knihoven pro vykreslování map na mobilních telefonech existuje dnes mnoho. Velká část, kterou jsem při výzkumu hned vyloučil, jsou rastrové knihovny, protože se velmi odlišují od cíle mé práce.

1.2.1 Google Maps a MapKit

Pravděpodobně dnes nejpoužívanější jsou řešení nativní pro dvě cílové platformy Android a iOS, Google Maps [6] (dostupné jako knihovna dokonce na obou zmíněných platformách), respektive MapKit [16]. Důvod, proč je dnes vidíme v tolika aplikacích, je ten, že jsou zdarma a jejich implementace do aplikace je velmi jednoduchá a dobře zdokumentovaná. Mapy jsou v obou případech samozřejmě proprietární a vlastní je daný výrobce, tedy Google či Apple.

Modifikace těchto map je většinou omezena na vlastní anotace nad základní mapou. To znamená, že tvůrce aplikace může přidávat vlastní body zájmu (Points of Interest) nebo geometrické tvary. Google Maps navíc narozdíl od MapKit umožňují také upravit samotný vzhled mapy.

Hlavní nevýhodou je v obou případech nemožnost využít je bez připojení k internetu. Protože to může být podmínka při vývoji mnoha aplikací, jsou vývojáři odkázáni na konkurenční knihovny.

1.2.2 Další knihovny

Pravděpodobně největším konkurentem pro Google a Apple v oblasti mapování na mobilních zařízeních je společnost Mapbox [14]. Mezi jejich produkty patří mimo jiné turn-by-turn routing, satelitní snímky a hlavně samotná knihovna Mapbox GL Native [15]. Ta uspokojuje prakticky všechny mnou vytyčené požadavky a nabízí také vynikající podporu ze strany společnosti Mapbox. Zároveň je open-source a její uživatelé se tak mohou podílet na opravě chyb. Hlavním problémem ale zůstává zpoplatnění, které je vztažené na mapové podklady, se kterými knihovna pracuje. Mapbox vyvýjí vlastní mapy založené na OSM. Ačkoliv jsou zdarma k použití pro nekomerční aplikace, cena stoupá velmi rychle pro jakékoliv komerční využití.

Další knihovna, která se na trhu objevila nedávno, je Tangram-es. Je vyvíjená společností Mapzen [17] a jedná se o port knihovny určené pro webové prohlížeče. Jelikož je knihovna ve vývoji oproti konkurenci relativně krátce, některé podstatné funkce stále chybí (například offline mapování). V budoucnu se ale může potenciálně stát důležitým konkurentem pro Mapbox a ostatní knihovny.

Samozřejmě najdeme mnoho další knihoven, které ale nesplňují všechny podmínky, které jsem si určil. Jak je vidět v tomto souhrnu, pokud se chystáme vyvíjet mobilní aplikaci,

která bude obsahovat v nějaké formě i mapu, není příliš z čeho vybírat. To je jeden z důvodů proč jsem se rozhodl vytvořit tuto práci.

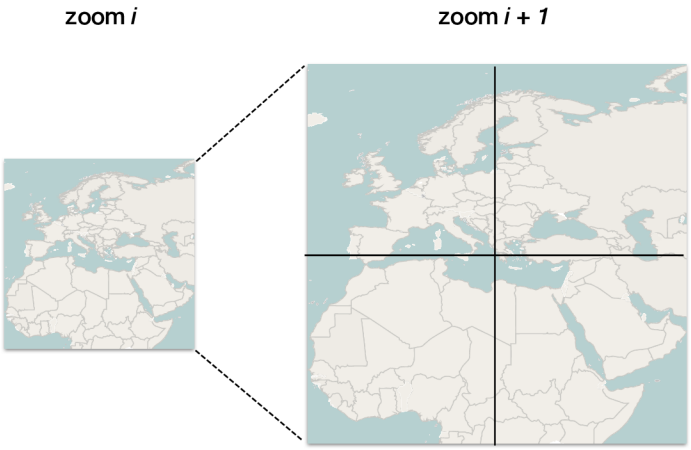
Kapitola 2

Metody zobrazování map

2.1 Struktura mapy

Mapa se v podání OSM (stejný formát ale používají i další mapy, jako Google Maps [7]) skládá z dlaždic (tile). Každá dlaždice je málá část výsledné mapy. Druhý důležitý princip elektronických map jsou úrovně zvětšení (zoom levels). OSM je rozdělené do 20 úrovní [24]. Mapa na 0. úrovni je nejvíc oddálená a typicky zobrazuje pouze kontinenty a oceány. Celá je přitom obsažena na jedné dlaždici. Spolu se stoupající úrovní zvětšení se zvyšuje počet dlaždic které jsou potřeba vždy čtyřnásobně oproti předchozí úrovni, takže při maximální úrovni zvětšení je celá zeměkoule rozdělená na více než 274 miliard dlaždic.

Jak z předchozích odstavců vyplívá, každý tile můžeme jednoznačně identifikovat pomocí tří souřadnic. K x a y přibyla ještě třetí souřadnice dále označovaná jako *zoom*.



Obrázek 2.1: Virtuální mapa je nejčastěji tvořena úrovněmi (zoom_level), které se skládají z rostoucího počtu dlaždic (tile).

Rozdělení mapy na dlaždice a použití schématu kde každou dlaždici popisujeme pomocí tří souřadnic má za následek několik věcí:

- Pokud potřebujeme jen mapu města, nemusí databáze obsahovat prvních 9 úrovní, a tím se změnší velikost dat. To stejné platí pro mapy, které zobrazují například pouze kontinenty a nepotřebují velké množství detailů.

- Implementace a použití tohoto schématu je velmi jednoduché. Komunikace mezi klientskou aplikací a serverem probíhá na základě jednoduchých požadavků, takže je snadné změnit poskytovatele mapových dat.

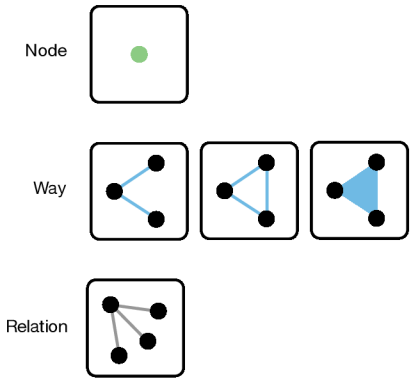
2.1.1 Elementy OSM mapy

Základní komponenty, ze kterých se skládá každá OSM mapa, jsou tři následující:

Node představuje bod na mapě. Pomocí bodu lze popsat mnoho různých prvků, jako bod zájmu, lavičku, telefonní budku apod.

Way je složený element, který se skládá ze 2 až 2000 dalších elementů, nejčastěji bodů (node). Může tvořit složitější tvary jako cesty nebo oblasti.

Relation je struktura popisující vztah mezi několika jinými elementy (které mohou být i další *relation*), které se mohou ve vztahu vyskytovat i vícekrát. Význam vztahu je popsán pomocí elementu **tag** (viz níže).



Obrázek 2.2: OSM data mohou obsahovat tyto druhy elementů.

Tyto elementy mají z pohledu mapy grafický význam. Společně tvoří geometrické tvary jako čáry, polygony a multi-polygony. Pro zobrazení mapy je ale potřeba znát i další informace o každém jejím prvku. K tomuto účelu slouží v OSM **tagy**. Každý tag představuje klíč a hodnotu, a popisuje nějakou vlastnost daného prvky. Nejčastější využití je klasifikace prvků mapy (např. silnice 1. třídy, silnice 2. třídy, řeka a mnoho dalších), k tomuto účelu je komunita OSM dohodnutá na běžných kombinacích klíčů a hodnot. Každý uživatel ale může přidávat prvkům libovolný počet nových tagů pro další použití.

2.2 Srovnání rastrových a vektorových map

Existují dvě kategorie uložení mapových dat, které jsou dnes běžně používané:

Rastrové mapy jsou tvořeny obrázky, které jsou předgenerované například na serveru, který je hostuje. Problém tohoto přístupu je poté hlavně v přibližování mapy. Je očividné, že pokud jsou popisky vykreslené předem, při přiblížení začne velmi rychle docházet k rozostření a nakonec i k nečitelnosti textu. Z dnešního pohledu je na mobilních telefonech, kde se klade velký důraz na uživatelovu zkušenost (User Experience), tento druh map méně vhodný.

Alternativní přístup je oddělit data, která musí zůstat zaostřená i při manipulaci s mapou, jako jsou popisky nebo ikony, a ty vykreslit poté nad zdrojovou mapou. Tímto přístupem získáme dobře čitelnou mapu, která nevíc podporuje i rotaci.

Vektorové mapy se v dnešní době stávají velmi populárními. Data pro takovou mapu jsou do cílového zařízení přenášena jako geometrie (čáry, polygony, body) a ta je vykreslena pomocí hardwarové akcelerace za běhu na uživatelově zařízení.

Tento přístup má hned několik výhod oproti rastrovým mapám. Vektorové mapy mohou být úspornější co do velikosti dat (není to ale pravidlo). Druhá výhoda je možnost stylizace zdrojové mapy podle potřeb uživatele. To zahrnuje možnost zvolit si za běhu, jaké vrstvy budou zobrazeny, aniž bychom museli stahovat další data. Stejné podklady můžeme navíc využít pro aplikace se zcela různými požadavky, například aplikaci pro vodáky nebo turistickou mapu Prahy, aniž bychom museli upravovat zdroj dat.

Pro tuto práci jsem zvolil právě druhou kategorii, zejména kvůli popsaným výhodám. Zobrazení takové mapy je zcela jistě složitější než pouhé zobrazení obrázků rastrové mapy. K tomuto účelu jsem zvolil OpenGL ES API které popisuje sekce 3.

2.3 Formáty vektorových map

U rastrových map je formát uložení obrázek, práce s nimi je tedy velmi jednoduchá. Pro vektorové mapy je ale princip o dost složitější. Protože se jedná v podstatě pouze o popis geometrických tvarů a dodatečné informace, volba formátu uložení je daleko širší. Přesto těch nejpoužívanější formátů je pouze několik a popsané jsou zde:

XML není z mnoha důvodů pro distribuci map online vhodný (velká datová náročnost, pomalý rozbor atd.). Proto se také k těmto účelům nepoužívá, zmíněný je proto, že jsou v tomto formátu uložená zdrojová data projektu OSM.

GeoJSON [5] je formát, který je uložený v JSON souboru, kterému navíc definuje zvláštní strukturu, která musí být dodržena. Stejně jako u XML se jedná o textový formát a proto je více datově náročný.

Protocol buffer [8] je dnes moderní serializační formát vyvíjený firmou Google. Umožňuje ve speciálním souboru (.proto) definovat schéma zpráv které chceme serializovat nebo deserializovat, a z tohoto souboru vytvořit objekty v programovacím jazyce vyvíjené aplikace. Jeho využití se nakonec rozšířilo i do map a například OSM si tak můžeme stáhnout i ve formátu PBF.

MVT [18] není formát jako takový, ale specifikace uložení vektorových map ve formátu Protocol bufferů definovaný společností Mapbox. Jeho využití je dnes ale mnohem širší.

Pro svoji práci jsem nakonec zvolil formát MVT. V tomto formátu jsou data dostupná hned z několika míst, například již zmiňovaná společnost Mapzen nabízí online mapové podklady zdarma. Nejvíce jsem pracoval s OpenMapTiles [22], které ke stažení nabízí mapy vytvořené z několika zdrojů včetně OSM pro celou zeměkouli, jednotlivé státy nebo města.

2.4 Zeměpisné souřadné systémy

Souřadné systémy se v geografii používají k jednoznačnému určení polohy libovolného místa na povrchu země. Takovýchto systémů existuje celá řada, ten nejčastěji používaný popisuje polohu pomocí tří souřadnic: zeměpisná šířka (tj. úhlová vzdálenost od rovníku), zeměpisná délka (tj. úhlová vzdálenost od nultého poledníku) a poslední je nadmořská výška.

2.4.1 Projekce

Vzhledem k tomu že právě popsané geografické souřadnice mají tři dimenze, ale typická mapa má dimenze pouze dvě, musí nejdřív dojít k projekci těchto souřadnic do jiného, dvourozměrného systému. Touto projekcí jsou převedeny všechny významné body na dvourozměrnou rovinu, kterou lze následně libovolným způsobem zobrazit. Projekcí existuje více a jsou mezi nimi významné rozdíly ve výsledném zkreslení (projekci bez zkreslení z principu provést nelze).

Podle druhu mapy, která je vytvářena, se volí projekce, která způsobuje nejmenší zkreslení přípustné pro daný účel. Webové mapy (např. Google Maps, OSM a další) nejčastěji používají **Sférickou Mercator** projekci [25]. Její použití je důležité hlavně při vytváření podkladových dat pro vykreslování dalšími knihovnami. Pro vykreslování už není nutné projekci znát, protože všechny souřadnice jsou už upravené. Často je ale nutné provést z klientské aplikace projekci zpět do geografických souřadnic, například pokud uživatel klikne na mapu a chce znát adresu daného domu. Ve vykreslovaných materiálech taková data nemusejí být, a jsou načítána z externí služby.

Kapitola 3

Vykreslování pomocí OpenGL

3.1 Co je to OpenGL

OpenGL je standard vyvíjený konsorciem Khronos Group Inc. [11] zahrnující firmy napříč průmyslem, které se společně podílejí na definování otevřených standardů, mezi které patří právě i OpenGL a OpenGL ES.

Standard OpenGL definuje aplikační rozhraní (API) pro renderování 2D a 3D grafiky na grafickém procesoru (GPU). Toto API dnes patří mezi jedno z nejrozšířenějších, podporované nejen na počítačích, ale také na mobilních telefonech. Je důležité si uvědomit, že tento standard definuje pouze dostupné funkce a konstanty, samotná implementace závisí na výrobcích hardware. To je také důvod, proč jsou některé části OpenGL méně využívané, protože jsou mezi implementacemi drobné rozdíly. Pro OpenGL dále existuje řada rozšíření (extensions) která mohou přidávat funkcionalitu specifickou pro daného výrobce, nebo přidávat možnost používat funkce z novější verze API, než které hardware podporuje (např. `OES_vertex_array_object`).

Toto API je stále ve vývoji a od vytvoření v roce 1992 prodělalo mnoho změn a bylo vydání několik hlavních verzí (tou poslední je 4.5). Dne 16. února 2016 vydala Khronos Group první verzi API nové generace Vulkan [26]. Mezi jeho cíle patří například lepší spolupráce mezi CPU a GPU a menší vytížení procesoru než je tomu například u OpenGL. Ačkoliv budou nadále OpenGL a Vulkan ko-existovat, pro budoucí aplikace se bude pravděpodobně přecházet z OpenGL na nová moderní API jako je Vulkan nebo Metal.

3.2 Stav OpenGL na mobilních telefonech

Pro mobilní zařízení byl definován standard OpenGL ES, jehož první verze byla postavena na OpenGL 1.0. Ačkoliv poslední vydaná verze je 3.2, v mojí práci jsem se rozhodl využít verzi 2.0. Hlavním důvodem je vysoká podpora, která na Androidu podle statistik Googlu tvoří všechna zařízení [19]. Pro platformu iOS nejsou takto detailní statistiky k dispozici, ale OpenGL ES 2.0 je základní profil na všech dnešních zařízeních s iOS a z toho můžeme usoudit, že je podpora stejně vysoká.

Ve verzi OpenGL ES 2.0 byl stejně jako v OpenGL 3.2 odebrán pevný vykreslovací řetězec. Ten byl nahrazen plně programovatelným vykreslovacím řetězcem. Vykreslování probíhá v několika fázích, přičemž pro každou můžeme definovat vlastní program vykonávaný přímo na GPU – **shader**. Tyto programy se píší ve specifickém jazyce zvaném

OpenGL Shading Language (GLSL) [21]. Pokud chceme vykreslit obraz do framebufferu, musíme definovat minimálně dva shadery: **vertex shader** a **fragment shader**.

3.3 Základní principy OpenGL

3.3.1 OpenGL kontext

K tomu aby bylo možné volat funkce OpenGL je zapotřebí vytvořit OpenGL kontext. Ten představuje vlastně celý stav, ve kterém se OpenGL v danou chvíli nachází, například do jakého bufferu probíhá vykreslování, vlastní OpenGL objekty, které se budou vykreslovat a mnoho dalších věcí. Každý kontext je přitom spjatý s jedním vláknem, typicky s tím, ve kterém je vytvořen.

Tento kontext za programátora nejčastěji vytváří systém aniž by musel provádět nějaké zvláštní akce. Na iOS k tomuto účelu existuje třída `GLKView`, která zjednodušuje inicializaci OpenGL. Druhá možnost je vytvořit kontext a framebuffer manuálně a použít běžné `UIView`. V takovém případě musí být vrstva daného view typu `CAEAGLLayer`.

I na platformě Android existuje třída, která představuje plochu, na kterou můžeme pomocí OpenGL vykreslovat geometrii. Tento koncept je rozdělen do tříd `GLSurfaceView` a `GLSurfaceView.Renderer`.

GLSurfaceView vlastní plochu (surface) do které provádí OpenGL vykreslování. Podporuje kontinuální režim (překreslení v pravidelných intervalech) nebo režim na vyžádání (překreslení pouze na základě požadavku od aplikace).

GLSurfaceView.Renderer je interface který obsahuje několik metod k implementaci. V nich se provádí konkrétní OpenGL operace pro vykreslování. Vytvoření třídy, která tento interface implementuje, se děje nejčastěji v `GLSurfaceView` při jeho inicializaci.

Sdílený kontext

Navrhovaná architektura nevyužívá pouze kreslení na hlavním vlákně, ale využívá také jedno vlákno v pozadí na kterém se provádí časově náročné aktivity OpenGL. Problém ovšem je, že OpenGL kontext je vždy navázán na jedno vlákno, a proto musí dojít k vytvoření druhého kontextu. To ovšem nestačí, protože dva takovéto kontexty by měly své vlastní prostředky (textury, buffer objekty, shader programy atd.). Naštěstí k tomuto účelu existuje koncept sdílených kontextů na obou platformách.

Android Na Androidu není postup vytvoření takového kontextu příliš dobře zdokumentovaný. Vytvořit druhý kontext je možné například z nativního C++ pomocí funkcí z hlavičkového souboru `egl.h` [1].

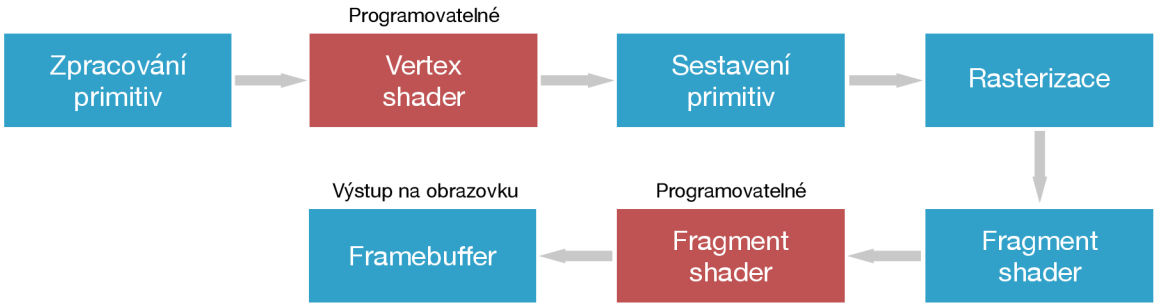
iOS Pro sdílení OpenGL objektů mezi více kontexty se používá `EAGLSharegroup`. Postup přidání obou kontextů do skupiny je jednoduchý – EAGLContext obsahuje konstruktor který jako druhý parametr přijímá právě objekt typu `EAGLSharegroup`, kterému se předá skupina z nějakého dalšího kontextu.

3.3.2 Programovatelný vykreslovací řetězec

V prvních verzích OpenGL (ES) probíhalo vykreslování primitiv předem definovaným způsobem. Této sekvenci kroků se říkalo **pevný vykreslovací řetězec**. Přestože šlo konfigurovat chování pomocí různých parametrů, v každém kroku docházelo k přesně definovaným

operacím. Od verze OpenGL 3.2 (respektive OpenGL ES 2.0) došlo k odstranění **pevného vykreslovacího řetězce** a jednotlivé fáze vykreslování byly nahrazeny programovatelnými kroky.

Přechod na **programovatelný řetězec** přinesl do OpenGL velkou flexibilitu. Fáze jako zpracování vrcholů nebo fragmentů probíhají jako vykonávání speciálního programu zvaného shader, přímo na GPU. Moderní grafický procesor může obsahovat několik stovek až tisíců jednotek pro vykonávání těchto programů, které pracují paralelně. Díky tomu je provedení některých operací, které se dříve prováděly na CPU, mnohonásobně rychlejší na grafickém procesoru.



Obrázek 3.1: Tento obrázek zobrazuje část programovatelného řetězce v OpenGL ES 2. Některé kroky jsou zjednodušené, například výstup z Fragment shaderu prochází několika dalšími operacemi, než se dostane do Framebufferu.

3.3.3 Shader programy

Pro tyto programy mají grafická API vždy speciální programovací jazyk, v případě OpenGL je to GLSL. Po spuštění klientské aplikace se shadery zkompilují a nahrají se na grafickou kartu. Každý shader obsahuje funkci *main*, ve které musí nastavit svůj výstup, který je předem daný pro každou fázi vykreslování. Jazyk GLSL samozřejmě podporuje základní datové typy jako *bool*, *int*, *float*, tak i složitější vektorové datové typy (*vec*3, *vec*4 atd.) a matice (*mat*3, *mat*4 atd.). Pro práci s texturami existují speciální typy tzv. **samplery**, které „obsahují“ textury různých rozměrů (1D, 2D, 3D).

GLSL navíc definuje několik speciálních kvalifikátorů, které slouží k předávání hodnot z klientské aplikace, nebo mezi různými fázemi[1]:

Attribute slouží k předání hodnot různých pro každý vrchol v bufferu který se vykresluje, proto ho lze použít pouze ve **Vertex shaderu**.

Uniform proměnné se nastavují většinou jednou po aktivaci shaderu. Jejich hodnota je proto stejná pro všechny vrcholy/fragmenty.

Varying označuje proměnné, které předávají hodnotu mezi dvěma fázemi vykreslování a proto musí v obou shaderech existovat proměnná se stejným názvem. U takto označených proměnných navíc automaticky dojde k interpolaci hodnot (u typů, kde je to možné), takže se používají například pro barevné přechody.

[1]Kvalifikátory **attribute** a **varying** byly v pozdějších verzích OpenGL odstraněny a místo nich existuje pouze kvalifikátor **in**. Podle toho, v jakém shaderu se proměnná nachází, se chová podobně jako jeden z předchozích dvou typů.

Každý program musí definovat minimálně dva shadery, než se cokoliv zobrazí na obrazovku:

Vertex Shader přichází na řadu jako první a provádí se pro každý vrchol, který je součástí vykreslované geometrie. Každý z vrcholů může mít více atributů jako je pozice, barva apod. Tělo programu provádí alespoň transformaci vrcholu z prostoru 3D světa do prostoru obrazovky (za pomocí transformačních matic), může však provádět i složitější transformace.

Fragment Shader navazuje na předchozí shader a provádí zpracování fragmentů (pixelů) výsledného obrazu. Vstupem jsou proměnné nastavené ve Vertex shaderu, které jsou lineárně interpolované mezi jednotlivými vrcholy. Výstupem této fáze je vždy výsledná barva daného fragmentu. Zde se provádí například i čtení barvy z textury.

Existují také další druhy shader programů, ale výše zmíněné jsou jediné dva druhy dostupné v OpenGL ES 2.0. Novější verze 3.2 přinesla několik shader programů, dostupných na desktopové verzi OpenGL už delší dobu: geometry shader, compute shader a tesselation shadery.

3.3.4 Buffer objekty

Buffer objekty představují pojmenovanou část paměti alokovanou v paměti grafické karty. Mají mnoho využití, například pro uložení pozic vrcholů geometrie nebo uložení indexů trojúhelníků. Po úspěšném vytvoření **buffer objektu** ho lze navázát na různé cíle, které určují, jak bude dál použit.

Nejčastější použití je uložení samotné geometrie. V takovém případě je buffer navázán na cíl (*target*) `GL_ARRAY_BUFFER`. Vrcholy nejčastěji neobsahují pouze svou pozici, mohou obsahovat také barvu, texturovací souřadnice, normály pro výpočet osvětlení a další. Z tohoto důvodu je také důležité rozhodnutí, jak budou data v bufferu uložená za sebou, přičemž se často volí ze dvou způsobů:

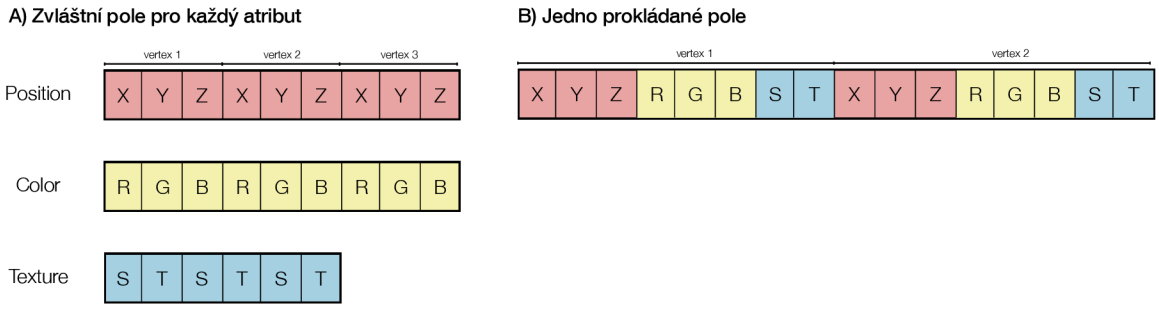
- Pro všechny vrcholy jsou jednotlivé druhy dat (pozice, barva apod.) uloženy za sebou.
- Pro každý vrchol jsou uloženy všechny informace hned za sebou, následují všechny informace pro druhý vrchol atd. Tomuto se také říká **prokládané** (interleaved) uspořádání.

V dnešní době je doporučován právě druhý způsob. Jeho výhoda, která je viditelná i z obrázku 3.2, je lepší lokalita paměti. Informace se čtou za sebou a není potřeba přeskakovat mezi vzdálenými místy v paměti (což s velkou pravděpodobností způsobí výpadek cache) pro každý vrchol.

Před použitím **vertex bufferu** pro vykreslení musí být grafické kartě popsán způsob uložení jednotlivých informací v bufferu tak, aby se správně předaly jako atributy vertex shaderu.

3.3.5 Textury

Obrazová data se v OpenGL ukládají do textur. Každá textura má definovaný barevný formát, v OpenGL ES jsou to hlavně **GL_RGB** a **GL_RGBA**. U obou formátů má každá barevná složka 8 bitů a jsou v poli uložené zasebou jako červená, zelená a modrá.



Obrázek 3.2: Srovnání mezi dvěma způsoby uložení dat používanými v OpenGL.

Existuje mnoho dalších formátů, které podporují až novější verze OpenGL, kde může být pořadí složek, ale i jejich velikost, jiné, např. **GL_BGRA**, **GL_RGB5** a další.

Pro nahrání textury na grafickou kartu je potřeba mít hodnoty jejích pixelů v poli v jednom ze zmíněných formátů. OpenGL samotné nenabízí žádné funkce na čtení obrázků, takže k tomuto účelu musí být použity externí knihovny. Na mobilních platformách je řešení jednoduché, protože Android i iOS nabízejí metody, jak načíst *PNG* obrázek a také z něj získat pole bytů.

Kreslení obarvených objektů ale není jediné využití textur. Mohou navíc sloužit také jako cíle kreslení. To se často využívá pro efekty které vyžadují více průchodů (multi-pass) tak, že se každý průchod vykreslí do textury, použije se v dalším průchodu a na obrazovku se vykreslí až výsledná textura. Další využití je pro takzvaný off-screen rendering, tedy vykreslování mimo obrazovku, například pokud je potřeba vykreslit více věcí současně a prezentovat až výsledný obrázek. Díky sdílenému kontextu může být obraz připravený na pozadí, aniž by vykreslování složitých objektů brzdilo hlavní vlákno.

3.4 Vykreslování primitiv

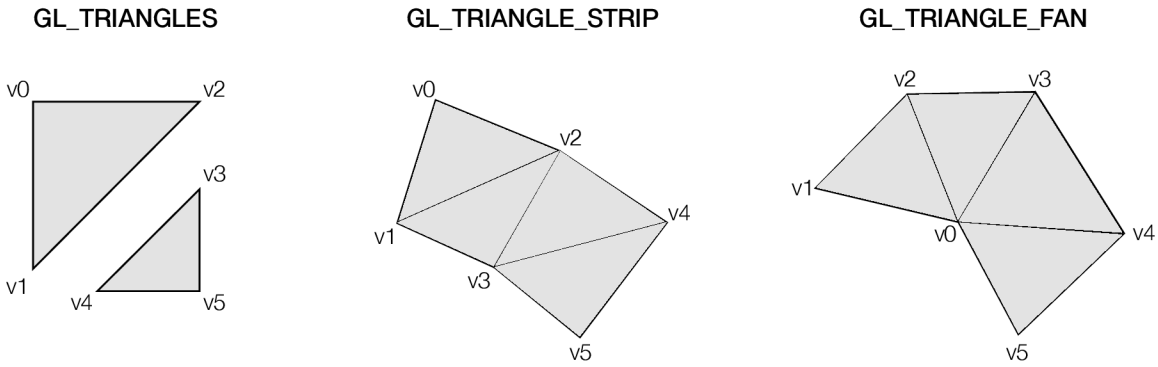
Jakékoliv vykreslování v OpenGL probíhá na úrovní tvarů nazývaných primitiva, které můžeme rozdělit do několika skupin: body (`GL_POINTS`), úsečky (`GL_LINES`), trojúhelníky (`GL_TRIANGLES`). Existuje také několik dalších, které jsou ale **deprecated** (např. `GL_QUADS`). Z tohoto seznamu vyplývá, že jakýkoliv složitější tvar se musí skládat z bodů, čar nebo trojúhelníků, aby se dal pomocí OpenGL vykreslit.

V následujících podkapitolách je detailněji popsaný přístup k vykreslování tvarů, které jsou potřebné k zobrazení prakticky libovolné dvourozměrné mapy.

3.4.1 Indexování trojúhelníků

Důležitý koncept, který umožňuje ušetřit paměť a někdy také zrychlit proces vykreslování, je indexování trojúhelníků. Vykreslování trojúhelníků lze v OpenGL provést několika způsoby: jednotlivé trojúhelníky, pás nebo vějíř. Každý z případů používá sdílení jiného počtu vrcholů mezi následujícími trojúhelníky. Samostatné trojúhelníky jsou očividně nejvíc paměťově náročné, protože i dva trojúhelníky, které sdílí dva vrcholy, jsou popsány pomocí 6 vrcholů. Ačkoliv ostatní způsoby už jsou úspornější, je většinou nemožné složitou geometrii převést na pás nebo vějíř trojúhelníků.

Naštěstí existuje jednoduchý způsob jak tento problém obejít, zvaný vykreslování indexovaných trojúhelníků. Jsou k tomu potřeba dva buffery, jeden s vrcholy trojúhelníků, druhý s indexy vrcholů v prvním bufferu. Při vykreslování se vždy přečtou tři indexy z



Obrázek 3.3: Ukázka použití tří módů pro vykreslování trojúhelníků. Každý z nich se liší počtem vrcholů a trojúhelníků které z nich vzniknou.

index bufferu a vykreslí se trojúhelník z vrcholů na těchto indexech ve **vertex bufferu**. Díky tomu může být každý vrchol v bufferu uložen pouze jednou.

3.4.2 Vykreslování polygonů

Polygony jsou jeden ze dvou geometrických tvarů ve vektorových mapách (druhým jsou úsečky) a přestože vypadají na první pohled složitěji, jejich vykreslování je z principu jednodušší než vykreslování úseček. Pro polygony je použit vykreslovací mód GL_TRIANGLES, což znamená, že vstupní data musejí být trojúhelníky. Jak je získáme, je popsáno v sekci 4.5.1. Abych maximalizoval znovupoužití vrcholů, použil jsem navíc princip vykreslování indexovaných primitiv. Kromě samotného seznamu vrcholů (kde se žádný vrchol v ideálním případě nevyskytuje více než jednou) je na grafickou kartu nahrán seznam indexů. Každá trojice v tomto seznamu říká, jaké vrcholy použít k vytvoření jednoho trojúhelníka.

Tento přístup bohužel neumožňuje jednoduše přidat antialiasing. Jedním způsobem by bylo celý tvar nejdříve vykreslit mírně větší s barvou okraje a poté vykreslit znovu v normální velikosti. Problém s tímto přístupem je, že neumožňuje zohlednit díry v polygonech, které jsou běžně součástí. Druhý přístup, který se také často v grafice používá, je přeskreslit okraje pomocí standardních čar (GL_LINES). Tímto způsobem můžeme přidat nejen okraje (i když limitované, viz. 3.4.3) ale i jednoduchý anti-aliasing a navíc za relativně malou cenu snížení rychlosti vykreslení celého polygonu.

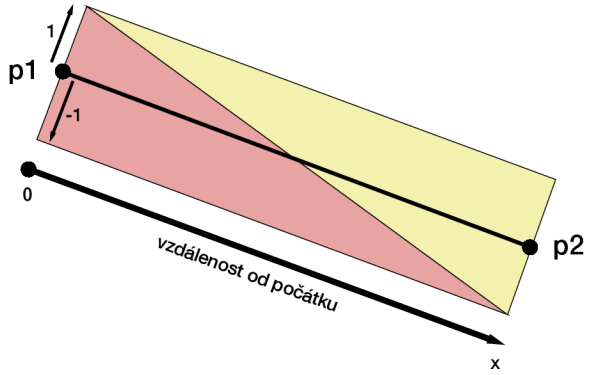
Shadery, které byly pro polygony použity, jsou minimální implementace, které v GLSL lze napsat. Vertex shader pouze transformuje pozici vrcholu ze světových souřadnic do ořezávacích souřadnic. Barva polygonu je předávána jako uniformní proměnná (celý polygon je jednobarevný). V budoucnu mám v plánu implementovat také polygony se vzory (obsaženými v textuře). K tomuto účelu stačí přidat atribut s UV souřadnicemi vrcholu a předat do fragment shaderu chtěnou texturu.

3.4.3 Vykreslování úseček

Vykreslování čar je v OpenGL většinou složitější, než se na první pohled zdá. Důvodem je, že mód `GL_LINES`, který je součástí standardu, má hned několik limitací, které jsou pro mapy problém. Největší z nich je maximální šířka vykreslované čáry, která je dána konkrétní implementací a může být na některém zařízení 1 pixel, na jiném 10 pixelů. Pro mapy očividně ani jedno nemusí být dostatečné a uživatelé této knihovny budou očekávat možnost nastavit si šířku libovolně, podle svých potřeb.

Z tohoto důvodu bylo potřeba zvolit jinou cestu k vykreslování úseček. Způsob, který se nabízí, je provést teselaci bodů úsečky a takto ji nejdřív převést na trojúhelníky. Tímto způsobem lze vykreslit čáru libovolné šířky nezávisle na hardware a zároveň na čáře provádět mnoho dalších modifikací v rámci shaderů. Při programování této části `Rendereru` jsem čerpal především z článků [28] a [29].

Každý vrchol čáry má čtyři atributy, které jsou vypočítané v kroku symbolizace (kapitola 4.5.1): pozice, normálna, směr normály a vzdálenost od počátku čáry. Z každého bodu jednotlivých úseček, které čáru tvoří, vznikají dva vrcholy, které vynásobí normálový vektor hodnotou 1, respektive -1. Touto hodnotou se ve vertex shaderu vynásobí normála a získáme směr kterým se bod posune. Ilustrace 3.5 zobrazuje, jak vypadají výsledné trojúhelníky.



Obrázek 3.4: Ilustrace zobrazuje úsečku, převedenou na dva trojúhelníky tak, aby mohla být zobrazena pomocí OpenGL. Kromě atributů pozice má každý vrchol také vzdálenost od počátku čáry (x) a směr posunutí vrcholu (1 nebo -1) po normále. Tyto hodnoty se použijí v shaderu pro vykreslení úsečky s libovolnou šířkou.

Poslední atribut, vzdálenost od počátku čáry, je použit k zobrazení vzorů, například čárkované nebo čerchované čáry. Na základě této hodnoty se vzorkováním textury vybraného vzoru zjistí hodnota průhlednosti tohoto pixelu. Textura pro vzor se vytvoří v kroku parsování stylu.

Výsledná pozice vrcholu se vypočítá podle následujícího vzorce (ve vertex shaderu):

$$v = ns\frac{w}{2} \tag{3.1}$$

Kde:

n je normála úsečky

s je směr posunutí, má hodnotu 1 nebo -1

w je požadovaná šířka úsečky

3.4.4 Vykreslování ikon

Ikony v mapách se používají především pro body zájmu (Points of interest, POI). Pro vykreslení ikony v OpenGL jsou dvě možnosti, pomocí bodů (`GL_POINT`) nebo trojúhelníků. Situace s `GL_POINTS` je podobná jako s `GL_LINES`. Používají se zřídka (například pro částicové systémy). Fungují na jednoduchém principu, geometrii definujeme jako jeden bod,

který grafický adaptér převede na čtverec. Opět bohužel nelze přesným způsobem definovat libovolné rozměry a tak se tato technika příliš nepoužívá.

Pro každou ikonu se v mé knihovně vytvoří čtverec pomocí dvou trojúhelníků. Na ty už je jednoduché namapovat správným způsobem texturu a zobrazit je.

3.4.5 Vykreslování textu

Zobrazení textu běžná grafická API nepodporují, proto museli programátoři přijít na jiný způsob jak text zobrazit. Naskýtají se dvě možnosti: teselace a zobrazení pomocí trojúhelníků, nebo rasterizace textu a zobrazení pomocí textury. Většina dnešních aplikací používá postup druhý, protože vykreslení textu skutečně vektorově je i na dnešním hardware v reálném čase neuskutečnitelné.

Typický postup vykreslování textu tedy zahrnuje několik kroků:

1. Znaky vybraného fontu (glyphy) se vykreslí v jedné nebo více velikostech do textury. K tomu většinou slouží nějaká knihovna.

2. Ve chvíli, kdy je potřeba zobrazit nějaký text, vytvoří se na daném místě čtverce s odpovídajícími velikostmi podle daných znaků textového řetězce.

3. Při vykreslování se na každý čtverec namapuje určitá část textury, která obsahuje znaky fontu.

Pomocí těchto kroků lze zobrazit text a libovolně ho transformovat jako jakoukoliv jinou OpenGL geometrii. Mezi dnes nejpoužívanější knihovny pro vygenerování atlasu z fontu patří FreeType [3].

Rozšířením tohoto způsobu jsou takzvané signed distance field (SDF). Při použití této techniky se do textury nevykreslí hodnoty pouze jedna (bílá) nebo nula (černá), ale pro každý pixel se spočítá vzdálenost k nejbližšímu okraji geometrie (v případě fontu vzdálenost k okraji glyphu). Na okraji znaku je hodnota nejčastěji 0,5 a směrem dovnitř roste k jedničce. V roce 2007 publikovala firma Valve dokument [30], ve kterém popisuje použití těchto polí při renderování decals a textu ve hrách. Výhodami této metody je ostřejší text při různých velikostech, možnost jednoduchým způsobem přidat obrys nebo i stín. Hlavní nevýhodou je potom to, že se stávají rohy znaků, které byly ostré, mírně zaoblené, při malých velikostech textu, jako například na mobilním telefon, ale rozdíl není znatelný.

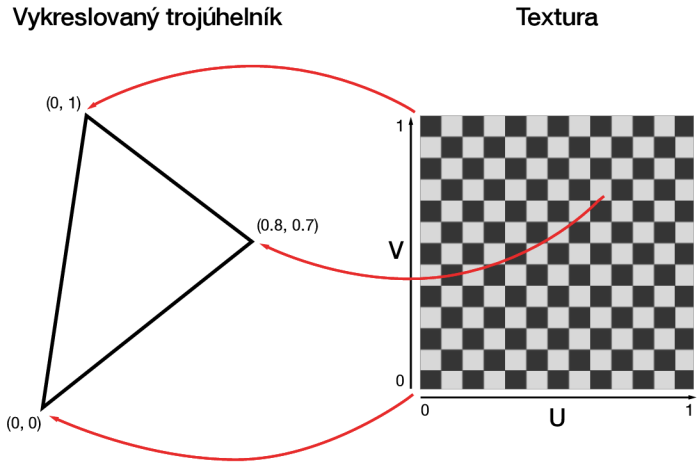
Obrázek 3.5: Příklad písmene „A“ v SDF textuře. Hodnota průhlednosti 0.5 odpovídá přibližně okrajům původního znaku. Jak je vidět, celý font se vejde do textury s relativně nízkým rozlišením (například 1024×1024). Přesto bude výsledný text i při zvětšení ostrý.

V této knihovně jsem použil právě SDF k vykreslování textu. Uživatel si zatím musí texturu s glyphy vygenerovat předem, ale k jejímu vytvoření jsou naštestí volně dostupné různé nástroje (například Hiero [9]). Fragment shader, který jsem vytvořil, umožňuje také definovat barevný obrys textu, který se často používá k odlišení textu od pozadí.

3.4.6 Nanesení textury

Textury popsané v sekci 3.3.5 obsahující obrázek mohou být dále naneseny při vykreslování na geometrii místo obyčejné barvy. K tomuto účelu jsou zavedené souřadnice, nejčastěji označované UV, které se na obou osách pohybují v rozsahu $< 0, 1 >$. Konvencí OpenGL je, že levý dolní roh má souřadnice $(0, 0)$.

Pokud je textura nastavená jako aktivní a její id je předáno do **fragment shaderu**, už stačí jen znát UV souřadnice pro každý vrchol vykreslované geometrie. Jak již bylo zmíněno v sekci 3.3.3, **varying** proměnné jsou interpolované pro každý fragment, takže se používají i pro UV souřadnice vrcholů.



Obrázek 3.6: Pro nanesení textury musí být pro každý vrchol známé souřadnice UV. Pomocí nich se ve fragment shaderu zjistí barva pro každý bod trojúhelníka.

Kapitola 4

Návrh

4.1 Načtení a zpracování mapy

4.1.1 Načítání offline

Jak již bylo řečeno v kapitole 2.1, celá mapa je rozdělená do částí–dlaždic. Pro uložení a práci s mapou bez internetu je tedy nutné uložit více dlaždic dohromady. K tomuto účelu se nejčastěji používá obyčená relační databáze. Všechny dlaždice jsou uložené v jedné tabulce (TILES). Tabulka musí obsahovat minimálně 4 sloupce, a to: x, y, *zoom* a *data*. První tři obsahují odpovídající souřadnice ze systému OSM a poslední položka obsahuje samotná data (například v binární podobě jako BLOB).

Kromě tabulky TILES obsahuje databáze často ještě jednu tabulku s metadaty. V ní jsou uložena data, která popisují, co mapa obsahuje, aby bylo jednodušší ji správně zobrazit. Mezi položky může patřit například minimální a maximální úroveň přiblížení, formát dat (`jpg`, `png`, `mvt` atd.), licence a další.

Načtení dlaždice v režimu offline probíhá v těchto krocích:

1. Otevření databáze obsahující mapu a přečtení metadat která potřebujeme.
2. Z vyšších vrstev přicházejí požadavky na načtení dlaždice.
3. Vytvoří se požadavek do databáze, který vybírá řádek podle souřadnic hledané dlaždice.
4. Databáze vrátí odpovídající data, která se musí dále zpracovat podle formátu ve kterém jsou do databáze zapsaná. Může také vrátit chybu, pokud databáze požadovanou dlaždici vůbec neobsahuje.

4.1.2 Načítání online

Pokud je zdroj mapy uložený na vzdáleném serveru, je načítání jednoduché. Veřejné API na takovém serveru publikuje adresu přibližně v tomto formátu „`{zoom}/{x}/{y}.{formát}`“. Po provedení požadavku na tuto adresu budou vrácena data dlaždice na daných souřadnicích *zoom*, x a y. Stejná adresa může vracet data v různých formátech podle přípony, kterou v požadavku definujeme. Stejným způsobem jde proto přistupovat k serverům, ať už hostují mapy ve vektorové nebo rastrové podobě.

Protože je proces načítání dat časově náročná úloha, ať už se načítají lokálně nebo v horším případě z internetu, provádí se v sekundárním vlákně. Skutečnost je ale taková,

že ne vždy proběhne načítání stejně rychle, například vlivem routování v rámci internetu. Z tohoto důvodu, a protože jsou mobilní telefony dnes vysoce výkonné, je dobré těchto vláken vytvořit hned několik. Vznikne-li požadavek na načtení dlaždice, přidá se do fronty, ze které jej tato vlákna přebírají a paralelně je zpracovávají. Taková vlákna mohou existovat současně například 4 nebo 8. Protože ale není většinou na displeji vidět větší počet dlaždic a tudíž nevznikne požadavek načátat jich tolik současně, vytvářet jich více nepřinese žádné další zrychlení.

4.2 Správa dlaždic

Správa dlaždic úzce souvisí s předchozí sekcí, která popisuje načítání dat z databáze nebo internetu. Pro to je ale potřeba vědět, jaká data by měla být zobrazena uživateli. Načítání dlaždic, které budou zobrazeny, probíhá podle aktuálně viditelné oblasti, kterou lze vypočítat podle středu mapy, přiblížení a velikosti displeje. Pokud je znám levý horní roh a spodní pravý roh zobrazované oblasti (viewport), lze také určit x a y souřadnice krajních dlaždic viditelných na obrazovce, a jejich *zoom* na základě aktuálního přiblížení.

Okraje viditelné oblasti se určí jako:

$$v_{tl} = c - \frac{\frac{v_s}{s}}{2} \tag{4.1}$$

$$v_{br} = c + \frac{\frac{v_s}{s}}{2} \tag{4.2}$$

Souřadnice levého a pravého krajního tilu se určí jako (podobný výpočet je proveden i pro osu y):

$$x_0 = \left\lfloor \frac{v_{tl_x}}{t} \right\rfloor \tag{4.3}$$

$$x_1 = x_0 + \left\lceil \frac{v_{br_x}}{t} \right\rceil \tag{4.4}$$

Kde:

v_s je velikost zobrazovací plochy

s je aktuální zvětšení

c je aktuální střed viditelné oblasti

t je velikost dlaždice (šířka a výška jsou stejné)

v_{tl} je levý horní roh viditelné oblasti

v_{br} je pravý dolní roh viditelné oblasti

x_0 je x souřadnice levé krajní dlaždice

x_1 je x souřadnice pravé krajní dlaždice

Poté co jsou určeny krajní souřadnice, provede se iterace nad všemi kombinacemi x a y souřadnic uvnitř rozsahu. Pro každou z nich se nejdřív provede vyhledání v paměti cache. Pokud zadaná dlaždice v cache není, musí se samozřejmě načíst a následně také nahradí v cache dlaždici která nebyla nejdéle použita.

4.2.1 Cache

Načítání dlaždic se opakuje po každé manipulaci s mapou, která vede ke změně viditelné oblasti. Aby se ale stejné dlaždice nemusely načítat stále dokola, je potřeba nějakým způsobem je udržovat po určitou dobu v paměti. K tomuto účelu je součástí každé knihovny pro vykreslování map i paměť cache. Jednoduchý algoritmus, který funguje dobře i pro mapy, je Least Recently Used (LRU), při kterém se novými daty nahrazují vždy data, která byla naposledy použita před nejdelší dobou.

Součástí objektů, které se v cache paměti ukládají, musí být také klíč, na základě kterého se objekty hledají. Jak bylo řečeno v sekci 2.1, každou dlaždici lze identifikovat pomocí trojice souřadnic, které budou tvořit tento klíč. Pro nalezení dlaždice v cache se hledá shoda souřadnic x, y a $zoom$. V nejhorším případě proto pro každou dlaždici v cache dojde k porovnání tří čísel, což určitě není tak efektivní jako srovnání jednoho hash klíče. Protože je ale na mobilím zařízení vždy zobrazeno jen malé množství dlaždic (například 8), malá maximální velikost cache (20 položek se osvědčilo) je dostatečná.

4.3 Interakce s mapou

Stavem mapy jsou její střed a zvětšení. Z tohoto stavu a velikosti viditelné oblasti (dané velikostí displeje daného zařízení) lze určit, jaké dlaždice jsou zrovna viditelné. Tento stav se ve většině případů mění na základě interakce uživatele s mapou, jako je tažení nebo tzv. pinch-to-zoom.

Popis pomocí těchto dvou proměnných (střed a zvětšení) mnohdy ale není dostačující. Další interakce, která je u map velmi častá, je rotace mapy. Krom toho navíc přibližování i rotace mohou probíhat (a nejčastěji probíhají) podle jiného bodu než je střed mapy. Naštěstí existuje jednoduchý způsob jak všechny tyto transformace popsat a tím jsou transformační matice.

Při použití matic stav popisují dvě matice a jedna floating-point proměnná: matice posunutí, matice rotace a zvětšení. Nakonec musí existovat ještě jedna matice, do které se ostatní transformace akumulují vzájemným vynásobením. Poté se posunutí a rotace nastaví opět na matici identity. Jednotlivé interakce uživatele mají následující vliv na mapu:

Posunutí o (dx, dy) vynásobí aktuální matici posunutí s vektorem s negativními hodnotami dx a dy, a nulovou hodnotou v ose Z (mapa leží v rovině XY):

$$T' = T \begin{bmatrix} -dx \\ -dy \\ 0 \end{bmatrix} \tag{4.5}$$

Přiblížení je složitější než posunutí nebo rotace, protože musí ve skutečnosti na mapu aplikovat dvě transformace místo jedné. Pro přiblížení musí být známá změna přiblížení Δz a také bod p ke kterému přiblížení probíhá (přiblížit může uživatel k jinému bodu než je střed mapy). Na základě těchto hodnot a hodnot aktuálního stavu mapy lze spočítat o kolik se musí mapa posunout.

Nové přiblížení určíme z aktuálního přiblížení následovně:

$$s' = s + \Delta z \tag{4.6}$$

Určení bodu, který je středem přiblížení ze středu mapy, a polohy bodu na displeji:

$$s_c = c - p \tag{4.7}$$

Kde c je aktuální střed mapy. Hodnota, o kterou se mapa posune odpovídá vektoru o jaký se posune bod po přiblížení:

$$t = \frac{s_c}{s'} - \frac{s_c}{s} \tag{4.8}$$

Rotace je zadaná pomocí bodu p, kolem kterého se má mapa otočit, a úhlu α. Pro korektní otočení kolem určitého bodu je nejdřív od matice rotace odečtená poloha tohoto bodu (tím se střed rotace stane identický s počátkem souřadného systému), poté se provede rotace, a posunutí zpět.

$$R' = \begin{bmatrix} p_x \\ p_y \\ 0 \end{bmatrix} R \begin{bmatrix} -p_x \\ -p_y \\ 0 \end{bmatrix} \tag{4.9}$$

Kde R je matice rotace na ose Z o úhel α.

Posunutí a přiblížení může probíhat i poté, co uživatel zvedne prst z displeje zařízení, například po dvojkliku běžně dochází k pomalému přiblížení mapy. Proto musí knihovna takovéto interakce zaznamenat a v rámci hlavní smyčky kontinuálně aktualizovat mapu.

4.4 Styl mapy

Jak již bylo definováno v úvodních požadavcích na vytvořenou knihovnu, vzhled zobrazené mapy musí být snadno upravitelný. K tomuto účelu slouží stylovací soubor ve formátu JSON, ve kterém programátor/designér definuje, jaké vrstvy OSM vybrat, jak je zobrazit, odkud načíst fonty a tak dále.

Kořen tohoto souboru je slovník který může obsahovat několik párů klíč–hodnota, minimálně však musí obsahovat klíč „layers“.

layers je pole vrstev které budou vidět na výsledné mapě. V každé vrstvě musí být definovaný název zdrojové vrstvy **source_layer** (ve zdrojových datech MVT). Dále musí vždy definovat typ vrstvy **type**, které je jeden z těchto: **line**, **polygon**, **icon** nebo **label**. Vrstva bude při kreslení ovlivňovat pouze prvky tohoto typu.

textures je pole textur, každá z nichž může obsahovat více obrázků. Ideální situace je obsažení všech potřebných ikon a obrázků v jedné textuře, čímž se lze vyhnout přepínání textury při vykreslování a tedy zpomalení vykreslování.

fonts obsahuje všechny použité fonty. Ke každému fontu patří dva soubory které jsou nutné k jeho správnému použití. První je textura, která obsahuje glyphy tohoto písma, a druhý je JSON soubor, ve kterém jsou jednotlivé glyphy popsané. Pro každý znak fontu musí obsahovat jeho pozici v odpovídající textuře, hodnotu posunutí na osách x a y.

4.4.1 Vrstvy

Každá mapa je složená z překrývajících se vrstev. V mé práci jsem se rozhodl rozdělit vrstvy na typy podle geometrie kterou obsahují. To znamená, že každá vrstva může obsahovat pouze čáry, polygony, text nebo symboly.

Vrstvy se navíc dělí podle požadavků na vykreslování: vykreslení předem do textury (platí pro čáry a polygony), nebo vykreslování v reálném čase (text, symboly). Důvodem tohoto dělení je, že čar a polygonů je na detailní mapě velké množství a pokud vznikne potřeba mapu překreslit (například když uživatel posune kameru), musí se překreslit všechny současně. Na mobilním zařízení to ale může znamenat až několik stovek milisekund, což je pro plynulý posun nepřípustné. Za předpokladu, že chceme dosáhnout rychlost vykreslování 30 snímků za sekundu, je čas pro jeden snímek pouhých 33 ms.

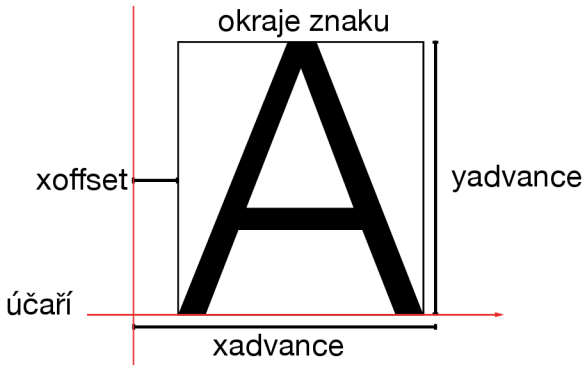
Z tohoto důvodu vykreslování probíhá ve dvou fázích. Všechna složitá geometrie, u které navíc nemusí vadit mírné rozostření při zvětšení (čáry a polygony) se vykresluje hned po načtení do textury. Druhá fáze vykreslování probíhá při každé změně viditelné oblasti mapy. V tu chvíli se nejdřív vykreslí pro každou dlaždici její předgenerovaná textura a poté všechen text a symboly. Nevýhodou je, že běžná geometrie nemůže nikdy překrýt text a symboly, ale nenarazil jsem na žádnou mapu kde by takový požadavek byl.

4.4.2 Textury

Kromě standardní geometrie se na mapě vyskytuje také mnoho různých ikon pro POI a jiné účely. Tyto ikony jsou obvykle před použitím shlukovány do jedné, maximálně několika málo textur. K použití je potřeba znát také pozice a rozměry obrázků, které textura obsahuje.

4.4.3 Fonty

Jak je popsáno v sekci 3.4.5, k vykreslování textu slouží předpřipravená textura obsahující znaky fontu. Pro načtení fontu je potřeba nejdřív načíst texturu s vyrenderovanými znaky. Kromě toho je nutné knihovně sdělit, na jakých pozicích se znaky nacházejí a také některé další informace ke správnému vysázení textu.



Obrázek 4.1: Ilustrace zobrazuje použití parametrů znaku při jeho vykreslování. $xoffset$ (případně i $yoffset$) pouze posune tento znak, ale neovlivní pozice následujících znaků. Nakonec se ke kurzoru přičte hodnota $xadvance$ (nebo $yadvance$ v případě vertikálního písma).

Odsazení ($xoffset, yoffset$) se přičítá ke kurzoru před vytvořením znaku, ale pozice kurzoru se tím nemění. Touto hodnotou se posouvají například nízká písmena na účaří textu.

Posunutí ($xadvance$) určuje, o kolik se kurzor má posunout po tomto znaku.

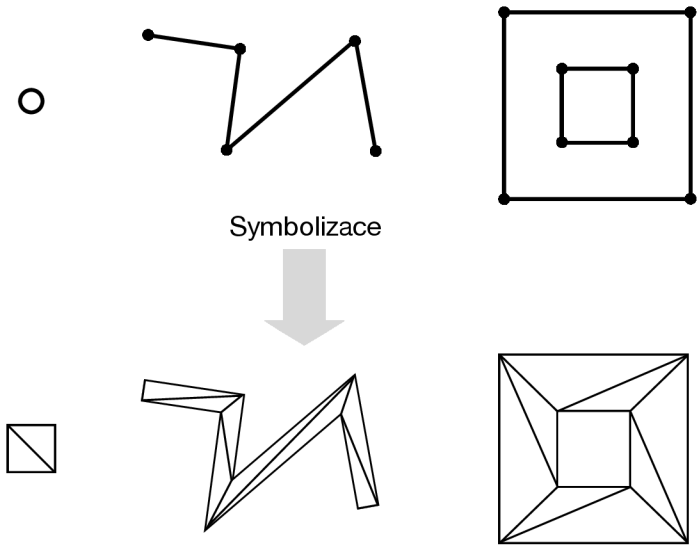
4.5 Zobrazení mapy

Každá dlaždice prochází zjednodušeně následujícím cyklem:

1. Načtení zdrojových dat pro souřadnice x, y, $zoom$.
2. Filtrace všech vrstev, které mají být na základě definovaného stylu zobrazené a zahození těch, které ne.
3. Symbolizace všech zbylých vrstev, tím vzniká dlaždice, která je připravený k vykreslení.
4. Vyrenderování části geometrie do statické textury (platí pro všechny vrstvy, které obsahuje čáry nebo polygony).
5. V rámci vykreslovací smyčky probíhá vykreslení textu a ikon v reálném čase.

4.5.1 Symbolizace a teselace

Výrazem symbolizace se často označuje proces převodu vstupních mapových dat na strukturu zobrazitelnou grafickým API. Jak bylo popsáno v sekci 2.1.1, mapa může obsahovat tyto tvary: body, řetězec čar, polygon, multipolygon (polygon obsahující díry). Jak ale vyplynulo ze sekce 3, OpenGL s takovými tvary pracovat neumí. Proto musí být mezi načtením a zobrazením mezikrok, který budu nazývat *symbolizace*.



Obrázek 4.2: Pro každý druh geometrie na vstupu do symbolizace vznikne odpovídající výstup. V bodech se vytvoří čtverce. Pro úsečky vznikne jejich reprezentace pomocí trojúhelníků a složitější polygony jsou převedeny na odpovídající reprezentaci pomocí trojúhelníků.

V tomto kroku musí pro každý prvek mapy proběhnout následující operace podle jeho typu:

Body mohou představovat mnoho různých druhů prvků. Nejčastěji se na jejich místě zobrazuje text, ikona, případně obojí. V případě ikony se na daných souřadnicích vytvoří čtverec s odpovídající texturou. Symbolizace textu vyžaduje vytvoření geometrie pro každý znak řetězce, může vyžadovat zalomení řádků apod.

Řetězec čar (anglicky linestring) se skládá z navazujících, nebo i oddělených úseček. Pro každou část se musí provést teselace na trojúhelníky se zadanými vlastnostmi.

Polygony a multipolygony se převedou vybraným algoritmem na trojúhelníky. Mnoho prvků v OSM mapách tvoří polygony s dírami, takže musí teselační algoritmus podporovat jejich odstranění. V případě, že je požadováno také obkreslení obrysu, mohou se pro okraje navíc vytvořit obyčejné čáry.

(a) Ikona zastávky (b) Silnice (c) Kontinenty

Obrázek 4.3: Příklady různých typů geometrie vykreslených touto knihovnou.

4.5.2 Renderovací smyčka

Architektura, kterou jsem zvolil a kterou popisuje tato práce, dělí vykreslování mapy do dvou fází:

1. Vykreslení složité geometrie dlaždice do textury jednou po jejím načtení. Do této fáze spadají všechny vrstvy, které obsahují polygony nebo čáry a probíhá na pozadí za pomocí sdíleného kontextu OpenGL, kapitola 3.3.1.

2. Druhá fáze je aktivní překreslení mapy, které probíhá v reálném čase. Nejdřív se vykreslí před-renderované textury, přes než se vykreslí všechny textové vrstvy a také ikony.

Cílem této architektury je hlavně rychlá reakce na uživatelovi akce. Protože vykreslení nejsložitějších částí probíhá na pozadí, není brzděno hlavní vlákno, na kterém probíhá nejen zpracování dotykových akcí uživatele, ale také překreslení obrazovky.

Tento přístup má jednu očividnou nevýhodu, a to částečná ztráta ostrosti, kterou jinak vektorové mapy přinášejí. Možností řešení je více, první možnost je dlaždice překreslit vždy po přiblížení mapy. Další možnost je vykreslit dlaždici do dostatečně velké textury tak, aby i při jejím maximálním zvětšení (to je zvětšení, při kterém jej ještě nepřekrývají dlaždice z další úrovně přiblížení) byla vykreslena 1 : 1. Při tomto přístupu je ale potřeba dát pozor na maximální velikost textury v konkrétní implementaci OpenGL, a také na paměťovou náročnost těchto textur.

4.5.3 Rozložení symbolů

Symboly jsou v této sekci myšleny popisky a ikony, které jsou vykreslovány v reálném čase. Ve fázi zpracování mapy jsou symboly vytvářeny nezávisle na ostatních a výsledkem je, že se velmi často mohou vzájemně překrývat. Částečně tomuto problému zabraňuje dobré nastavení stylu a úrovní přiblížení tak, aby se při nízkém přiblížení nezobrazovaly symboly pro místa, která ještě nejsou ani rozlišitelná.

To ale ve většině případů bohužel nestačí. Umístění symbolů je těžko předvídatelné, a stejně tak mapové materiály někdy obsahují prvky, které nelze očekávat (například dvě cesty velmi blízko u sebe, jako víceproudová silnice). Proto je součástí knihoven pro vykreslování map také třída, která dynamicky pro všechny viditelné symboly určuje, na základě různých pravidel, zda nemají být skryty. Detekci a vyřešení překrytí symbolů jsem už v rámci bakalářské práce neimplementoval.

Nejjednodušší detekci překrývání lze provést porovnáním hranic všech symbolů navzájem. Hranice symbolu lze reprezentovat jako obdélník a pro dva obdélníky lze snadno určit, zda se protínají. K urychlení detekce kolizí existují i složitější algoritmy, které dělí prostor do menších částí a tak se snaží snížit počet nutných porovnávání.

4.5.4 Několikavrstvé mapy

Některé mapy, se kterými se lidé také často setkávají, zobrazují hned několik mapových vrstev na sobě. Typicky je spodní vrstva klasická vektorová mapa a nad ní je vykreslená rastrová vrstva, která zobrazuje nějaká speciální data. Takto lze vytvořit například mapu srážek, větru apod. Může vzniknout také požadavek přidat rastrovou vrstvu mezi ostatní vrstvy které se vykreslují vektorově. To se využívá například k zobrazení elevace, protože stínování výšin a údolí nedokážeme zatím dost dobře vektorovými mapami popsat.

Zobrazit takovou mapu vyžaduje provést změny v dosud navrhované architektuře. Především by každá dlaždice mohla mít místo jednoho několik zdrojů dat. Ve stylu by navíc u každé vrstvy přibyla položka, která by odkazovala na jeden ze zdrojů. Vykreslování už by probíhalo stejným způsobem, protože by byla v předchozím kroku všechna data sloučena.

Kapitola 5

Implementace

5.1 Použité nástroje

Kód knihovny je rozdělený na sdílený kód, který tvoří jádro knihovny, a dodatečné projekty které ho zaobalují pro každou podporovanou platformu. Jádro knihovny je napsáno v C++, které je jednoduché zkompilovat na iOS i Android, a obě platformy umožňují s tímto kódem interagovat z nativního prostředí.

Na platformě iOS je použití C++ velmi jednoduché. Zdrojové kódy stačí přidat do projektu a poté k nim lze přistupovat z Objective-C, který je vlastně jen nadstavbou nad klasickým programovacím jazykem C. Umožňuje vytvářet instance tříd a volat jejich metody přirozeným způsobem. Nejpoužívanějším jazykem na platformách Apple se pomalu stává Swift, který je s každou verzí stále vyspělejší a také oblíbenější. Ten bohužel aktuálně nepodporuje použití C++ kódu, ale může snadno pracovat s kódem v Objective-C (což bylo nutné z důvodu zpětné kompatibility, mnoho iOS aplikací dnes obsahuje kód v obou jazycích), takže i ve Swiftu bude knihovna použitelná.

Situace na Androidu je bohužel složitější. Pro práci s kódem v C/C++ existuje sada nástrojů nazvaná **NDK**, která umožňuje zkompilovat kód v C++. Pro spolupráci mezi Javou a C++ se používá **Java Native Interface** (dále JNI). Výsledek je takový, že volání nativní funkce z Javy vyžaduje vytvoření dalšího hlavičkového souboru, který bude sloužit jako most mezi oběma funkcemi. Opačné volání, tedy volání Java funkce z nativního kódu, je ještě komplikovanější. JNI umožňuje získat ukazatel na funkci, k tomu je ale potřeba v textovém řetězci přesně zapsat její signaturu. Další komplikací je převod návratových hodnot a argumentů, které musejí být pomocí patřičných funkcí JNI upraveny.

K úspěšnému vývoji software jsou nutné také ladící a profilovací nástroje. V **Xcode** k tomuto účelu slouží **Instruments**, které umožňují profilování prakticky každého aspektu aplikace (alokace, time profiling, OpenGL ES profiling a další). Výchozí debugger v Xcode, LLDB, umožňuje spolehlivě ladit i kód v C++, což je velká výhoda. I v tomto ohledu je Android a konkrétně **Android Studio** (v době psaní byla poslední verze 2.3.1) za konkurencí pozadu. Ve verzi 2 tohoto IDE přibyla podpora hybridního ladění, tedy možnost plynule přecházet mezi kódem v Javě a C++. Bohužel z mé zkušenosti se v naprosté většině případů LLDB vůbec nepřipojí k běžící aplikaci, a tak nativní kód nejde vůbec ladit. V budoucnu se snad tato skutečnost změní ale zatím se zdá, že NDK obecně se při vývoji nevěnuje moc pozornosti.

Užitečný nástroj také do budoucna, který vyvíjí také Google, je GAPID [4]. Umožňuje zobrazit si jednotlivá API volání poslaná na grafickou kartu, rozebrat scénu a její části. Protože výstupem OpenGL aplikace je obraz, i když se povede program bez chyb zkompilovat,

nemusí být výstup takový, jaký programátor očekává. Chybu často nepomůže odhalit ani standardní ladění kódu, proto se hodí nástroje jako GAPID.

5.1.1 Platformě závislý kód

Některé operace, které jsou prováděny ze sdílené části knihovny, jsou platformě závislé. Patří sem zejména otevírání a čtení souborů, HTTP požadavky, ladící výstup do konzole a podobně. Pro tyto účely jsem vytvořil hlavičkový soubor `Platform.h`, který obsahuje deklarace těchto funkcí. Pro každou platformu existuje jiný zdrojový soubor, který tyto funkce definuje a obsahuje všechna volání závislá na platformě. Díky tomu může stejný kód na obou platformách pracovat se soubory transparentně a bez modifikací.

5.1.2 C++11 a C++14

Vývoj C++ jde velmi rychle dopředu a jeho použití se stává mnohem pohodlnějším než to bylo dřív. Aktuálními používanými verzemi jsou C++11 a C++14, které jsou z velké části podporovány ve většině moderních překladačů. Ve své implementaci jsem použil mnoho funkcí, které tyto moderní verze přináší. Ve velké míře jsem využil takzvané **smart pointers**, což je zaobalení obyčejného ukazatele, které umožňují automaticky spravovat příslušnou pamět a ve chvíli zrušení uvolnit daný objekt. Často využité jsou také menší novinky jako specifikátor `auto`, pro takto vytvořené proměnné bude automaticky určen datový typ z pravé strany výrazu.

Standard, který byl dokončen v roce 2017, nese označení C++17 a opět přináší užitečné prvky, jako datové typy `std::optional` nebo `std::any`. Protože ale zatím není běžně podporován a některé části standardu mohou v kompilátorech úplně chybět, raději jsem tuto verzi zatím nepoužil.

5.1.3 Matematické funkce

Protože vyžaduje má knihovna velké množství výpočtů, vybíral jsem kvalitní knihovnu se základními matematickými typy a operacemi. Nakonec jsem použil knihovnu OpenGL Mathematics (GLM) [20]. Tato knihovna obsahuje třídy jako `vec2`, `vec3` a `mat4`, které jsem hojně využil. Navíc má výhodu, že ji lze jednoduše využít s OpenGL, například při předávání matic z klientské aplikace shaderům.

5.2 Map

Třída `Map` je ústřední objekt celé knihovny. Tato třída spravuje připojený zdroj mapy, styl a aktuální viditelnou oblast a dostává informace o transformaci na základě uživatelových vstupů. Instance této třídy se vytváří v `MapView`, které představuje prvek v hierarchie UI a zobrazuje mapu. Tato třída obstarává základní životní cyklus ostatních důležitých komponent mapy, provádí inicializace OpenGL a aktualizaci a zobrazení mapy.

Aktualizace mapy probíhá standartně $60\times$ za sekundu (ale může dojít ke zpomalení pokud telefon nestíhá tak rychle vykreslovat). K aktualizaci mapy slouží funkce `update`. Na iOS je vytvořený objekt `CADisplayLink`, který umožňuje automatické volání metody synchronně s obnovovací frekvencí displeje telefonu (což je u iPhone právě $60Hz$). Na Androidu slouží k podobnému účelu třída `Choreographer`.

5.3 Načtení mapy

Jedním z kroků inicializace knihovny je definování zdroje map. Zdroje dat mohou být různého druhu a proto jsem vytvořil abstraktní třídu `DataSource`. Ta je velmi jednoduchá a její podtřídy musejí definovat jen funkcí `loadTile`.

Návratová hodnota `loadTile` říká, zda byla dlaždice nalezen nebo ne. Jako parametr je předán ukazatel na instanci třídy `Tile`. V případě úspěchu bude tento objekt naplněný daty dané dlaždice.

Jednotlivé podtřídy `DataSource` definují tuto metodu a v jejím těle načtou data z patřičného zdroje. V případě online zdroje map se provede HTTP požadavek na zadaný server se zadaným schématem adresy. Více času jsem věnoval načítání z offline zdroje, které je mírně složitější a je rozebráno v sekci 5.3.2.

5.3.1 Fronta zpracování

Jak bylo řečeno v návrhu, načítání a zpracování dlaždic může probíhat paralelně. K tomuto účelu jsem vytvořil frontu, do které jsou umísťovány žádosti o načtení dlaždice. Předem definovaný počet vláken `std::thread` z této fronty vybírá úlohy a provádí je v následujících krocích:

1. Inicializace instance třídy `Tile`.
2. Vyžádání dat od patřičného zdroje `DataSource`.
3. Přidání dlaždice do paměti cache.

Tyto kroky musejí být samozřejmě synchronizované tak, aby nedocházelo ke konfliktům mezi vlákny. K tomuto účelu má C++11 koncept mutexů a podmínkových proměnných. Všechna vlákna se na začátku své smyčky pokusí získat zámek. Vlákno, které ho získá, navíc čeká na signalizaci, že je ve frontě požadavek ke zpracování. Ve chvíli, kdy je požadavek přidán, může vlákno začít svoji činnost.

5.3.2 Načítání z MBTiles

Format *.mbtiles* je pouze převlečená SQL databáze, která musí dodržet předem definované schéma. Teoretický postup byl popsaný v sekci 4.1.1 a jeho implementace obsahuje třídu `MBTilesSource`. Využívá knihovnu SQLite 3 [13], kterou lze zkompilovat na obou platformách a má minimální nároky. Postup konkrétního dotazu s knihovnou SQLite je takovýto:

1. Připravení *statementu* voláním funkce `sqlite3_prepare_v2` s textem sql dotazu

   ```
   SELECT tile_data FROM tiles
   WHERE zoom_level=? and tile_column=? and tile_row=?
   ```

   K provedení této funkce musí být k dispozici otevřené spojení s databází, které je vytvořeno při inicializace datového zdroje.

2. Pomocí funkce `sqlite3_bind_int` se nastaví konkrétní hodnoty pro sloupce *zoom_level*, *tile_column*, *tile_row*, které závisí na souřadnicích tilu, který se má načíst.

3. Dotaz se provede a pokud je v databázi nalezen odpovídající řádek, je z databáze přečten blob a jeho délka v *bytech*. Bude následovat dekomprese a další zpracování dat.

Tabulka 5.1: Ukázka tabulky **tiles** z SQL databáze s mapou.

zoom_level	tile_column	tile_row	tile_data
1	0	0	bytes
1	0	1	bytes
1	1	0	bytes

Pokud nalezen není, funkce vrátí hodnotu `false`.

Pokud se povede z SQL databáze načíst příslušná data, další zpracování závisí na tom, v jakém formátu byla uložena. V případě OpenMapTiles se jedná, jak bylo zmíněno již v návrhu, o formát Mapbox Vector Tile. Aby databáze zabírala co nejméně místa, jsou před zapsáním buffery komprimované, takže po načtení se musí provést dekomprese knihovnou **zlib**. Tato knihovna je součástí obou operačních systémů a proto stačí vložit soubor `zlib.h` a přidat ji mezi linkované knihovny.

5.3.3 Parsování protocol bufferu

Na úrovni jednotlivých dlaždic jsou data ve formátu Mapbox Vector Tile (MVT). Existuje řada dalších formátů popsaných v sekci 2.3, ale implementoval jsem pouze práci s tímto formátem. Protocol buffery, což je formát ve kterém MVT specifikuje způsob zakódování mapových dlaždic, je binární formát. Proto je parsování na první pohled složitější než u textových formátů, ale také rychlejší.

Přestože existuje knihovna **Protobuf** také pro jazyk C++, její využití jsem po několika testech zavrhnul. Jednou z jejích hlavních funkcí je automatické generování tříd podle schématu, které programátor definuje ve zvláštním souboru `.proto`. Výsledné třídy obsahují serializační a deserializační metody, odpovídající proměnné a mnoho dalšího. Bohužel je ale jejich použití za běhu příliš pomalé, pokud má deserializace probíhat na desítkách objektů co nejrychleji.

Z tohoto důvodu jsem využil minimální implementaci protocol bufferů **protozero** [12]. Parsování probíhá manuálně, opět podle definovaného schématu, a žádné před-generované třídy nejsou použity.

Mapbox Vector Tile

Každá dlaždice zakódovaná jako MVT obsahuje na nejvyšší úrovni pole vrstev. Tyto vrstvy mohou obsahovat například kontinenty, dopravu, budovy, využití půdy a další. Každá vrtva obsahuje jednotlivé prvky mapy (features), například pro vrstvu dopravy jsou to jednotlivé silnice.

Dále popisuje každou vrstvu její jméno (*name*), velikost (*extent*) a klíče a hodnoty (*keys* a *values*). Každý prvek může mít vlastní metadata, která upřesňují jeho typ a další parametry, které jsou uložené jako pole celých čísel o sudé délce jménem *tags*. Každá dvě čísla z tohoto pole představují indexy odpovídajícího klíče a hodnoty, uložených ve vrstvě. Díky tomuto způsobu uložení se nemusejí klíče (a stejně tak i hodnoty), které jsou typu textový řetězec, opakovat na více místech a v každé vrstvě jsou uložené právě jednou, přestože na ně může odkazovat více prvků.

Zpracování MVT souboru probíhá ve dvou vnořených cyklech přes všechny vrstvy a poté přes všechny prvky vrstvy. U každého prvku probíhá kontrola na základě jeho tagů, a filtrů nastavených v souboru stylu mapy, zda má být do dlaždice přidán, nebo zahozen.

Každý prvek obsahuje pole *geometry*, které obsahuje zakódované polohy bodů, řetězce čar, nebo polygony. Z tohoto zakódovaného pole se vytvoří vektor bodů (`glm::vec2`), který je předán k symbolizaci.

5.4 Interakce uživatele

Vstup uživatele pomocí dotykového displeje je hlavní způsob jeho interakce s mapou a proto musí fungovat pohodlně a nabízet všechny očekávané funkce. Mezi mapou a operačním systémem jsem vytvořil mezivrstvu v podobě třídy `GestureDelegate`. Všechny zprávy o uživatelském vstupu jsou delegovány právě na instanci této třídy, která podle toho nastavuje polohu, přiblížení a rotaci mapy.

K tomuto účelu má několik veřejných metod, které jsou volány z kódu závislého na platformě:

```
void handlePanGesture(float dx, float dy);

void handleFlingGesture(float dx, float dy);

void handlePinchGesture(float x, float y, float scale);

void handlePinchGestureEnded();

void handleRotationGesture(float x, float y, float angle);

void handleDoubleTapGesture(float x, float y);
```

Rozpoznávání gest na iOS

Rozeznávání gest v iOS aplikacích probíhá pomocí tříd `UIGestureRecognizer`. Specializace této třídy jsou přidány na `MapView`, které při zavolání kteréhokoliv callbacku deleguje gesto přímo na `GestureDelegate` mapy.

Rozpoznávání gest na Androidu

Na Androidu je situace podobná jako na iOS. Pro detekci gest slouží `GestureDetector`, kterému je při inicializaci nutné předat také třídu implementující rozhraní `OnGestureListener`, na kterou deleguje `Detector` probíhající gesta. Bohužel není k dispozici rozpoznávání všech gest, například rotaci pomocí dvou prstů si musí programátoři implementovat sami.

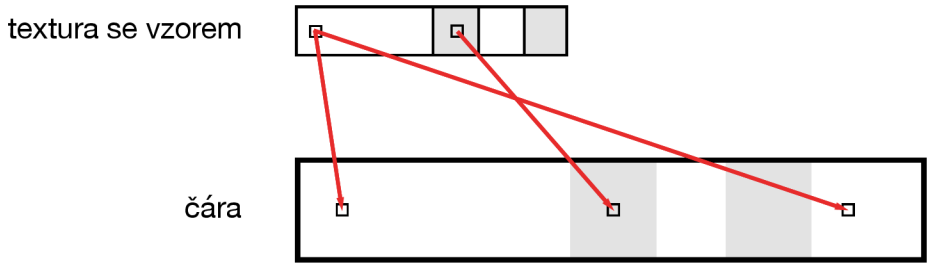
5.5 Symbolizace

5.5.1 Teselace čar

Pro vykreslení čar spolehlivým způsobem pomocí OpenGL je potřeba převést je na trojúhelníky, podobně jako při teselaci polygonů. Pro každý bod řetězce jsou vytvořené dva vrcholy. Oba mají normálu vypočítanou z úsečky, na které leží, a navíc je jednomu přiřazen směr posunutí 1 a druhému -1. Normála a směr posunutí jsou ve vertex shaderu vynásobené s šířkou čáry a výsledným vektorem je posunutý příslušný vrchol trojúhelníka. Interpolací směru normály (tedy mezi hodnotami -1 až 1), vznikne vzdálenost každého

bodu ležícího na úsečce od jejího středu. Na základě této vzdálenosti se na okraji úsečky provede vyhlazení pomocí GLSL funkce `smoothstep`. Díky tomu není pro vykreslení úseček potřeba žádný další antialiasing.

Knihovna umožňuje vykreslovat také přerušovanou čáru a libovolně definovat vzor pomocí pole hodnot, například $dashPattern = [2, 1]$. Tato hodnota říká, že bude čára vyplněná po dvě jednotky, poté jedna jednotka prázdná a tak stále dokola. Pole vzoru se při inicializaci stylu mapy převede na 1D texturu, která má velikost rovnu sumě všech hodnot v tomto poli. Mapování vzoru probíhá opět ve fragment shaderu následovně:



Obrázek 5.1: Pro zobrazení úsečky se vzorem se vytvoří $1D$ textura. Hodnota 255 (bílá) reprezentuje úsečku, černá hodnota naopak mezeru. Pomocí atributu vzdálenosti od počátku řetězce úseček, se pro každý bod zjistí hodnota z této textury. Pokud se bod nachází v mezeře, je jeho průhlednost nastavená na 0.

5.5.2 Teselace polygonů

Polygony v OSM mohou být jednoduché ale i složené, které navíc obsahují jednu nebo více děr. Pro převod těchto polygonů na trojúhelníky jsem využil knihovnu **Earcut** [2], vytvořenou firmou Mapbox. Vstupem do této knihovny je vektor vektorů. Vektory na druhé úrovni se skládají z bodů, a každý z nich popisuje jeden polygon. První z nich tvoří vždy vnější okraje, ostatní mohou být díry. Pravidlem také je, že úsečky tvořící polygony děr jsou orientované opačným směrem, než ty vnější.

5.5.3 Symbolizace ikon

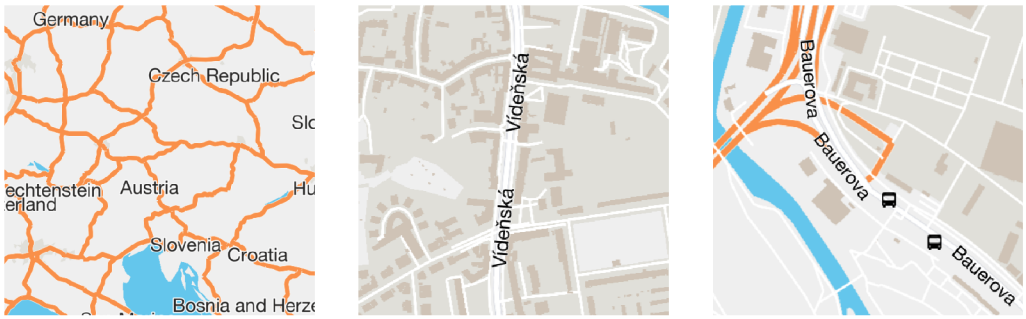
Přidání ikon probíhá velmi jednoduše. Zatím je podporované pouze pro bodové prvky mapy, které reprezentují například body zájmu (POI). Na pozici každého bodu se vytvoří čtverec, pomocí dvou trojúhelníků vykreslených OpenGL. Vytváření nového meshe v každém bodě není ideální řešení, ale OpenGL ES 2.0 bohužel nepodporuje instanciované vykreslování, takže to je jediná možná cesta.

5.5.4 Symbolizace textu

Pro přidávání textu do mapy musí být k dispozici `Font`. Text může být přidán pro jakýkoliv typ geometrie; zatím jsem implementoval pouze čáry a body. Později se chystám doplnit také umístění textu nad polygony.

Body

Pro body probíhá usazení textu jednoduše. Nejdřív se podle fontu a očekávané velikosti písma vypočítá šířka řádku pro patřičný textový řetězec. Toho lze docílit jednoduše tak,



Obrázek 5.2: Výsledek symbolizace textu v bodech a na čárách.

že se pro každý znak v řetězci nalezne odpovídající **glyph**, a vzájemně se sečtou všechny hodnoty *xadvance* (posunutí po ose x po vykreslení znaku). Výsledek je navíc třeba vynásobit hodnotou *fontScale*, protože velikost nahraného písma a zobrazovaného textu může být odlišná:

```
float fontScale = layer.fontSize / font.size;
float lineWidth = 0.0f;
for (int i = 0; i < str.length(); i++)
{
  lineWidth += font.xadvance(str.get(i)) * fontScale;
}
```

Kde:

`layer` je vykreslovaná vrstva

`str` je příslušný textový řetězec

Po vypočítání šířky už záleží na tom, jak má být text umístěn na obou osách, vzhledem k danému bodu: vlevo, vpravo, na střed. Zatím jsem implementoval pouze umístění na střed, ale plánuji doplnit i ostatní, které se budou definovat opět ve stylovacím souboru. Pro vygenerování geometrie znaků, což jsou vždy dva trojúhelníky, se nejdřív vytvoří kurzor na pozici horního levého rohu prvního znaku:

```
vec2 cursor = { point.x - lineWidth,
                point.y - layer.fontSize / 2 };
```

Kde:

`point` je bod, ve kterém má být text umístěný

Pro každý znak se vygeneruje `Mesh`, obsahující dva trojúhelníky, který dohromady tvoří obdélník odpovídající velikosti. Poté se kurzor vždy posune a takto se vygenerují postupně všechny znaky zadaného řetězce.

Čáry

Pro čáry, respektive řetězce úseček, může proběhnout umístění popisku více než jednou. Pokud mají začátek a konec řetězce velmi malou vzdálenost, nebo pokud je počet segmentů nízký, tento krok se přeskočí a rovnou se vybere segment nejdelší, na kterém bude popisek s největší pravděpodobností dobře rozeznatelný. Tento postup se často uplatní například na

silnice, které se skládají pouze z jedné úsečky z bodu A do bodu B, kterých je například ve městech hodně.

Pokud umístění popisku neproběhne podle předchozího algoritmu, musí se provést vybrání ideálních míst, kde budou popisky zobrazeny. Popisky se umísťují na pozice ležící na čáře v předem definovaných vzdálenostech od sebe. Vygenerování znaků už probíhá podobně jako by to byl pouze bod. Narozdíl od bodů může být ale text natočený podle úhlu čáry. K tomuto účelu má každý symbol vlastní transformační matici, která se aplikuje před jeho vykreslením.

5.6 Tile

Třída `Tile` zapouzdřuje každou dlaždici, ze kterých je tvořená výsledná mapa. Pro jedinečnou identifikaci má každý tile strukturu `TileID`, která obsahuje proměnné pro jeho souřadnice x, y a *zoom*. Tato třída má také přetížený operátor porovnání, které se provádí právě podle `TileID`, a využívá se například při hledání v cache paměti.

`Tile` obsahuje texturu, do které je vykreslena statická geometrie, což jsou všechny vrstvy typu `LineLayer` a `PolygonLayer`. Ve vykreslovací smyčce vykreslí `Renderer` tuto texturu na obrazovku. Po vykreslení textury se vykreslí také vrstvy typu `IconLayer` a `LabelLayer`. Tyto vrstvy se zobrazují v reálném čase, protože se jejich velikost mění podle přiblížení.

5.7 Styl mapy

Při parsování stylovacího JSON souboru je vytvořena instance třídy `Style`, která obsahuje vektor vrstev mapy `StyleLayer`. Navíc přechovává `Style` také fonty a další textury, které jsou potřebné k vykreslení mapy. Umožňuje získat jednotlivé vrstvy na základě jejich indexu, a také najít odpovídající vrstvu pro prvek mapy.

Nalezení vrstvy, která koresponduje s určitým prvkem mapy, se provádí na základě několika kritérií. Nejdřív se musí porovnat druh geometrie, který je přečtený ze zdrojového JSON souboru, a uložený v každém objektu `StyleLayer`. Pokud se typ vrstvy stylu shoduje s typem prvku mapy, porovnají se také názvy zdrojové vrstvy a název vrstvy, do které patří daný prvek. V případě, že i ty se shodují, provede se ještě porovnání podle filtrů, které mohou být definované v JSON souboru. V této implementaci jsou podporované pouze filtry na základě porovnání textových řetězců. Každý filtr obsahuje klíč a hodnotu, a pokud je stejný klíč i hodnota nalezena mezi tagy porovnávaného prvku, pak filtr splňuje.

5.7.1 Vykreslování

Styl mapy je využíván hlavně při vykreslování mapy. Všechny meshe jsou v třídě `Tile` uložené do *bucketů* (každý bucket je vektor instancí třídy `Mesh`), a každý bucket odpovídá jedné vrstvě mapy. Při vykreslování dlaždice se provede iterace nad všemi buckety. Pro každý z nich se aktivuje `StyleLayer` funkcí

```
void apply(const RenderInfo& renderInfo)
```

která aktivuje odpovídající **shader program** a také nastaví všechny uniformní proměnné. Poté co je vrstva aktivní, provede se volání funkce `render` všech mesh objektů v bucketu.

5.8 Renderer

Třída `Renderer` provádí veškeré vykreslování potřebné pro zobrazení mapy. Provádí mimo jiné inicializaci OpenGL prostředí, jako je nastavení míchání barev, barvy pozadí a podobně. První funkce, která se volá z hlavní smyčky, vykresluje dlaždice na obrazovku. Argumenty jsou instance `RenderInfo`, vytvořená ve třídě `Map`, a vektor dlaždic které budou zobrazeny. Dlaždice jsou dopředu seřazené tak, aby se nakonec vykreslily ty s největší hodnotou sořadnice *zoom* (jsou nejdetailnější).

```
void renderTiles(const std::vector<std::shared_ptr<Tile>> &tiles,
                 RenderInfo *renderInfo)
```

V průběhu vykreslování dlaždic se volá také metoda pro vykreslení symbolů, což jsou ikony a popisky, která má následující deklaraci

```
void renderSymbols(std::shared_ptr<Tile> tile,
                   RenderInfo *renderInfo)
```

RenderInfo

V průběhu celého kreslení mapy je potřeba předávat určité stavové proměnné do dalších podsystémů. K tomuto účelu slouží třída `RenderInfo`. Její instance je vytvořena ve třídě `Map` na začátku každého vykreslovacího cyklu, a poté předána rendereru. Jedná se o strukturu, která neobsahuje žádnou logiku, ale předává se pomocí ní aktuální matice transformace, projekční matice, zvětšení, a také styl mapy.

Rozložení atributů vrcholů

Vertex atributy jsou položky, které jsou předávané vertex shaderu pro každý vrchol geometrie. Každý atribut představuje nějakou vlastnost, nejběžnější jsou pozice, barva, texturovací souřadnice atd. Každý shader má jako vstup jiné atributy, například *lineshader* má atributy: pozice, normála, směr posunutí vrcholu, vzdálenost od počátku řetězce čar. Před použitím shader programu musí být atributy aktivovány a pro každý z nich musí být nastaven ukazatel.

Pro tyto účely jsem vytvořil třídu `VertexAttributeLayout` a její statické instance pro každý druh shaderu, který je v knihovně používán. Konstruktor třídy má jako jediný argument pole struktur `VertexAttribute`, která každá obsahuje všechny vlastnosti daného atributu. Funkce `enable()` a `disable()` umožňují jednoduchým způsob všechny potřebné atribut aktivovat, respektive deaktivovat. Díky této třídě mohu na jednom místě jednoduše definovat atributy všech shaderů.

Mesh

`Mesh` je třída, která zapouzdřuje objekt vykreslitelný pomocí OpenGL. Každý mesh obsahuje vlastní `VertexBuffer` a případně i `IndexBuffer`. Při jeho prvním použití automaticky proběhne nahrání geometrie na grafický adaptér, a má také metodu, která tyto data opětovně uvolní, pokud už s nimi nebude knihovna pracovat.

Nejdůležitější je metoda `void render(RenderInfo* renderInfo)`, ve které proběhne vykreslení, které má následující kroky:

1. Pokud ještě není geometrie nahraná na GPU, nahraje se.

2. Funkcí `glBindBuffer` se nastaví oba buffery jako aktivní.

3. Aktivuje se vertex layout funkcí `enable()`.

4. Teď může proběhnout samotné vykreslení. Pro indexované trojúhelníky k tomu slouží OpenGL funkce `glDrawElements`.

5. Nakonec je potřeba deaktivovat vertex layout (při dalším kreslení by jinak mohly být nastavené nepoužité atributy) a pro jistotu také nastavit výchozí buffery (hodnotou 0 ve funkci `glBindBuffer`).

Shader program

Aby se s shadery lépe pracovalo, vytvořil jsem třídu, která je zapouzdřuje. Stará se o kompilaci programů, které jsou předány jako textové řetězce. Součástí kompilace je také navázání atributů na přednastavené konstatní lokace funkcí `glBindAttribLocation`. To se provádí proto, aby nebylo potřeba jejich lokace manuálně získávat od shader programu (novější verze GLSL umožňují definovat lokaci atributu přímo ve zdrojovém souboru shaderu).

Ostatní funkce v této třídě slouží pro nastavování **uniform** proměnných různých typů (*int*, *float*, *vec*2, *vec*3) a vypadají například následovně

```
void setUniform3f(GLint location, const glm::vec3& v)
```

Jak je vidět z příkladu, pro nastavení uniformní proměnné je potřeba znát její lokaci (což je obyčejný identifikátor v podobě celého čísla). I k tomuto účelu obsahuje třída `ShaderProgram` funkci, která navíc lokace kešuje. Nalezení lokace v OpenGL je poměrně zdlouhavá operace a je zbytečné ji opakovat při každém použití programu. Proto se hodnota nejdřív hledá v lokální mapě, a až pokud není nalezena (tedy při prvním použití), vyžádá se přímo voláním `glGetUniformLocation`.

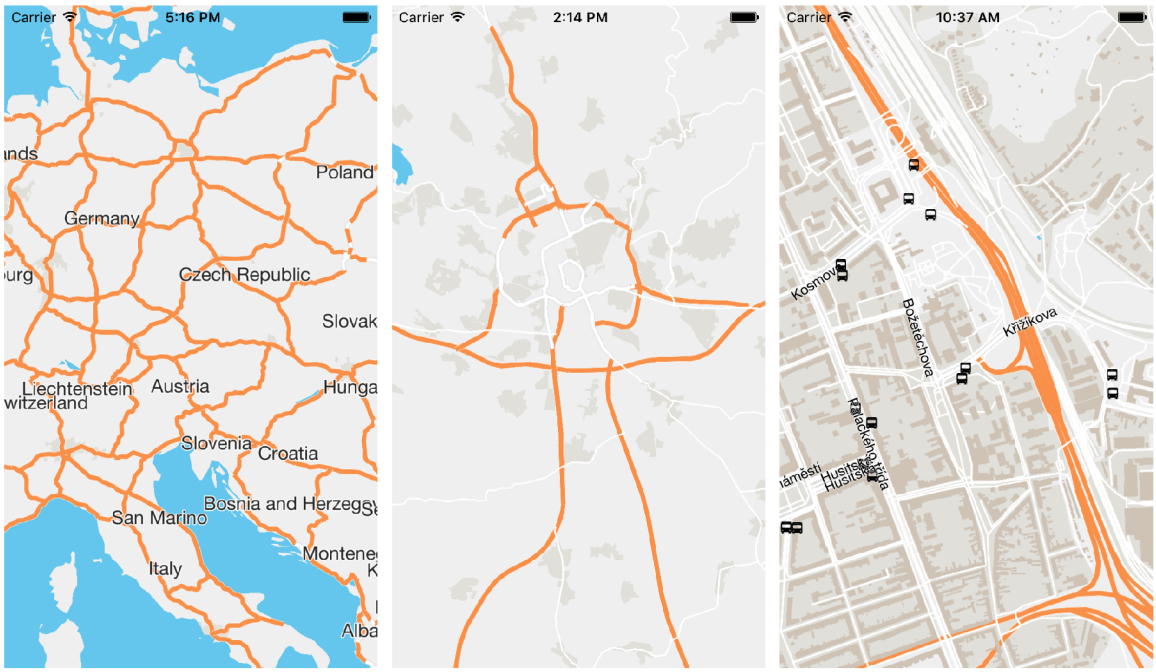
Kapitola 6

Vyhodnocení výsleků

Navržená knihovna byla implementovaná v jazyce C++. Výsledkem je kód, který běží na obou platformách: iOS a Android. Vytvořil jsem také vazby na jazyk specifický pro každou platformu, tedy Javu a Objective-C (použitelné také ve Swiftu).

6.1 Výsledná mapa

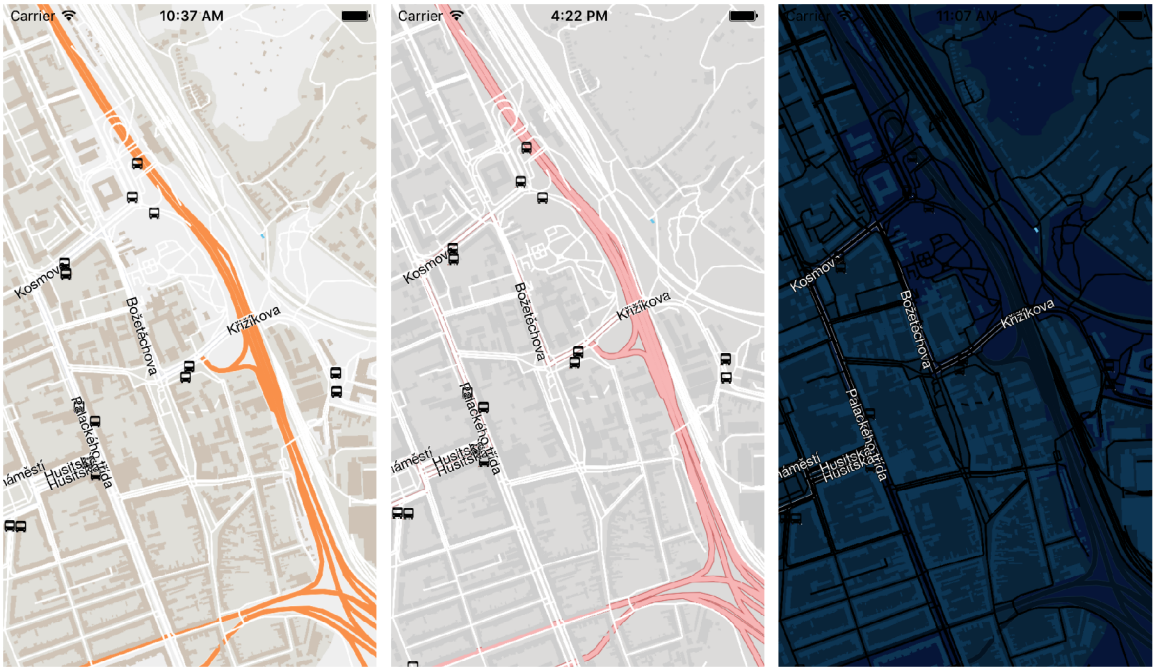
Pro účely testování a demonstrace jsem vytvořil ukázkovou aplikaci na obou platformách. Její součástí je offline balíček z OpenMapTiles s mapou Brna.



Obrázek 6.1: Snímky z ukázkové aplikace s mapou vykreslenou touto knihovnou. Každý z nich zobrazuje jinou úroveň přiblížení.

6.1.1 Definice stylu mapy

Součástí implementace je také podpora základních možností změny stylu mapy. Uživatel může definovat barvu všech prvků mapy, u cest také šířku a obrys. Může přidat vlastní font a také texturu s ikonami pro symboly. V budoucnu se chystám přidat také další možnosti (např. změnu stylu zakončení cest).



Obrázek 6.2: Stejné místo na mapě v různých stylech.

6.2 Rychlost knihovny

Všechny testy jsem prováděl na zařízení Samsung Galaxy S4. Aplikace byla zkompilovaná bez optimalizací (`-O0`). Pro měření rychlosti určitých úseků kódu jsem použil třídu `high_resolution_clock` ze jmenného prostoru `chrono`, který je součástí standardní knihovny C++. Všechny časy jsou naměřené v milisekundách a výsledek v tabulce je aritmetický průměr 10000 měření. Pro měření rychlosti operací nad konkrétní dlaždicí jsem zvolil úroveň přiblížení 14 se složitou mapou města.

Tabulka 6.1: Měření rychlosti knihovny

akce	**čas** [ms]
A) Načtení dlaždice	10.63
B) Zpracování dlaždice ve formátu MVT	427.57
C) Vykreslení statické geometrie dlaždice do textury	58.34
D) Vykreslení mapy do framebufferu	2.13

A) Čas který zabere připravení SQL dotazu, jeho provedení a následně dekomprese získaných dat (knihovnou zlib).

B) Po načtení dlaždice z internetu nebo databáze musejí být binární data zpracovaná. Naměřený čas je průměrné trvání metody `buildFromMVT`, která z binárních dat vytvoří instanci třídy `Tile`. Provádí se zde mimo jiné relativně náročná teselace geometrie.

C) Tato část vykreslování je prováděna na pozadí po zpracování dlaždice. Do textury jsou vykreslené čáry a polygony. Nakonec je zavolaná metoda `glFinish`, která zaručí, že jsou všechny operace ve frontě provedeny okamžitě.

D) Kompletní čas za který se provede metoda `render` objektu `Map`. Tato metoda zahrnuje výběr dlaždic viditelných na obrazovce a vykreslení každé z nich do framebufferu.

Jak je vidět z tabulky 6.1, přestože vykreslení jedné dlaždice zabere v průměru 58 ms, díky přesunutí těchto operací na druhé vlákno může být mapa vykreslená na obrazovku bez čekaní za přibližně 2 ms. Druhý poznatek z provedeného měření je, že z operací s dlaždicemi nejvíce času zabere jejich zpracování po načtení, takže by to mělo být prvním cílem dalšího ladění.

6.3 Nedostatky

Současná implementace je sice už dost komplexní, ale přesto se najdou určité nedostatky, které souvisí s vykreslováním. Některé funkce, které by byly ve finální knihovně potřeba, nebyly v rámci bakalářské práce implementované.

6.3.1 Překrytí symbolů

Jak jsem psal již v kapitole 4.5.3, ikony a popisky se velmi často navzájem překrývají. Zobrazovací knihovny proto musejí tyto kolize nalézt a vyřešit; posunout symboly tak, aby se nepřekrývaly, nebo některý ze symbolů skrýt. Tuto funkcionalitu jsem v bakalářské práci neimplementoval, ale plánuji ji jako pozdější vylepšení.

Obrázek 6.3: Dva výřezy ilustrující problém s překrývajícími se symboly.

6.3.2 Zakřivení popisků

Zobrazení popisků na cestách vyžaduje často také zakřivení tak, aby text následoval jednotlivé segmenty. Tím se samozřejmě komplikuje sázení znaků tak, aby se znaky navzájem nepřekrývaly.

Obrázek 6.4: Na těchto snímcích je zvýrazněné, jak by mělo vypadat zakřivení textu. Jak je vidět, zatím jsou znaky vysázeny vždy do rovného řádku.

6.3.3 Zakončení čar

V rámci materiálů OSM se může stát, že na první pohled navazující cesty jsou ve skutečnosti rozdělené na více částí. Při vykreslování takové cesty potom nastává problém, kdy na sebe jednotlivé části správně nenavazují.

Tento problém lze alespoň částečně zamaskovat mírným protažením cest na jejich začátku a konci. Prodloužením prvního a posledního segmentu zakryje mezeru mezi přerušenými cestami.

Kapitola 7

Závěr

Cílem této práce bylo prozkoumat a popsat způsob zobrazení vektorových map na mobilních telefonech, a to hlavně možnost zobrazení map z projektu OpenStreetMap, a následně takovou knihovnu implementovat. V rámci dokumentu byla popsána architektura knihovny, která byla také implementována v jazyce C++ na dvou mobilních platformách (Android a iOS).

Všechny základní funkce potřebné k zobrazení funkční mapy se povedlo implementovat. Mezi tyto části patří: načtení a správa tilů, definice stylu mapy v JSON souboru, vykreslení všech možných útvarů OSM. Knihovna v době dokončení práce funguje jak na systému Android, tak iOS. Vývoj jsem prováděl nejčastěji na zařízení Samsung Galaxy S4, na kterém knihovna většinu času dosahuje rychlosti 60 snímků za sekundu, s občasnými krátkými (v řádech desítek až stovek milisekund) záseky. Přesto není výsledek úplně stabilní, za což může především složitá správa paměti ve vícevláknovém prostředí, mým hlavním cílem v nejbližší době bude odladění knihovny.

Do budoucna potřebuje knihovna ještě mnoho práce aby bylo možné použít ji v produkčním kódu. Mým plánem je nejdřív se věnovat odladění existující části a odstranění všech chyb které ji v současné době zpomalují. Dále bude potřeba zdokumentovat schéma stylovacího souboru tak, aby mohl programátor nebo designér jednoduše vytvářet vlastní styly. Poté začnu přidávat nové funkce a také vylepšovat existující vykreslování, například přidáním vykreslování polygonů se vzorem.

Literatura

[1] Android egl API.
https://developer.android.com/ndk/guides/stable_apis.html#egl
[Online; navštíveno 10. 5. 2017].

[2] Earcut. https://github.com/mapbox/earcut.hpp [Online; navštíveno 10. 5. 2017].

[3] FreeType knihovna. https://www.freetype.org [Online; navštíveno 10. 5. 2017].

[4] GAPID: Graphics API Debugger. https://github.com/google/gapid
[Online; navštíveno 10. 5. 2017].

[5] GeoJSON. http://geojson.org [Online; navštíveno 10. 5. 2017].

[6] Google Maps library. https://developers.google.com/maps/documentation/
[Online; navštíveno 10. 5. 2017].

[7] Google Maps, Tile Coordinates. https://developers.google.com/maps/
documentation/javascript/maptypes#TileCoordinates
[Online; navštíveno 10. 5. 2017].

[8] Google Protocol buffers. https://developers.google.com/protocol-buffers/
[Online; navštíveno 10. 5. 2017].

[9] Hiero, nástroj pro generování (nejen) SDF fontu.
https://github.com/libgdx/libgdx/wiki/Distance-field-fonts
[Online; navštíveno 10. 5. 2017].

[10] History of OpenStreetmap.
http://wiki.openstreetmap.org/wiki/History_of_OpenStreetMap
[Online; navštíveno 10. 5. 2017].

[11] Khronos Group, Inc. https://www.khronos.org [Online; navštíveno 10. 5. 2017].

[12] Knihovna protozero. https://github.com/mapbox/protozero
[Online; navštíveno 10. 5. 2017].

[13] Knihovna SQLite. http://www.sqlite.org [Online; navštíveno 10. 5. 2017].

[14] Mapbox. https://www.mapbox.com [Online; navštíveno 10. 5. 2017].

[15] Mapbox-gl-native GitHub repozitář.
https://github.com/mapbox/mapbox-gl-native [Online; navštíveno 10. 5. 2017].

[16] MapKit framework. https://developer.apple.com/maps/ [Online; navštíveno 10. 5. 2017].

[17] Mapzen. https://mapzen.com [Online; navštíveno 10. 5. 2017].

[18] MVT Specifikace. https://www.mapbox.com/vector-tiles/specification/ [Online; navštíveno 10. 5. 2017].

[19] OpenGL ES zastoupení na Android zařízeních. https://developer.android.com/about/dashboards/index.html#OpenGL [Online; navštíveno 10. 5. 2017].

[20] OpenGL Mathematics. http://glm.g-truc.net/0.9.8/index.html [Online; navštíveno 10. 5. 2017].

[21] OpenGL Shading Language. https://cs.wikipedia.org/wiki/OpenGL_Shading_Language [Online; navštíveno 10. 5. 2017].

[22] OpenMapTiles. https://openmaptiles.org/about/ [Online; navštíveno 10. 5. 2017].

[23] OpenStreetMap website. https://www.openstreetmap.org [Online; navštíveno 10. 5. 2017].

[24] OpenStreetMap Zoom levels. http://wiki.openstreetmap.org/wiki/Zoom_levels [Online; navštíveno 10. 5. 2017].

[25] Sférická Mercator projekce. http://wiki.openstreetmap.org/wiki/EPSG:3857 [Online; navštíveno 10. 5. 2017].

[26] Vulkan API. https://www.khronos.org/vulkan/ [Online; navštíveno 10. 5. 2017].

[27] Brothaler, K.: *OpenGL ES 2 for Android: A Quick-Start Guide (Pragmatic Programmers)*. Pragmatic Bookshelf, 2013, ISBN 1937785343.

[28] Deslauriers, M.: Drawing Lines is Hard. https://mattdesl.svbtle.com/drawing-lines-is-hard [Online; navštíveno 10. 5. 2017].

[29] Konstantin Käfer, M.: Drawing Antialiased lines. https://www.mapbox.com/blog/drawing-antialiased-lines/ [Online; navštíveno 10. 5. 2017].

[30] of Valve, C. G.: Improved Alpha-Tested Magnification for Vector Textures and Special Effects. 2007, SIGGRAPH Course on Advanced Real-Time Rendering in 3D Graphics and Games. http://www.valvesoftware.com/publications/2007/SIGGRAPH2007_AlphaTestedMagnification.pdf.

[31] Verth, J. M. V.; Bishop, L. M.: *Essential Mathematics for Games and Interactive Applications*. A K Peters/CRC Press, třetí vydání, 2015, ISBN 1482250926.