VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

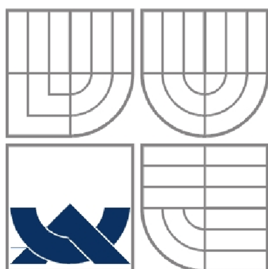
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTERGRAPHICS AND MULTIMEDIA

VÝVOJ GRAFICKÝCH APLIKACÍ NA IPHONE A IPAD PLATFORMĚ

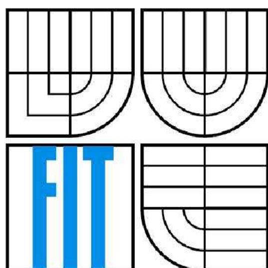
DIPLOMOVÁ PRÁCE
MASTER‘S THESIS

AUTOR PRÁCE BC. PETR FIALA
AUTHOR

BRNO 2012



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



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

VÝVOJ GRAFICKÝCH APLIKACÍ NA IPHONE A IPAD PLATFORMĚ

GRAPHICS APPLICATION DEVELOPMENT ON IPHONE AND IPAD PLATFORM

DIPLOMOVÁ PRÁCE
MASTER‘S THESIS

AUTOR PRÁCE — BC. PETR FIALA
AUTHOR

VEDOUCÍ PRÁCE — ING. PEČIVA JAN, PH.D.
SUPERVISOR

BRNO 2012

ZDE

VLOŽIT

ZADÁNÍ

Abstrakt

Práce se zabývá tvorbou grafických aplikací pro systém iOS, popisuje základy OpenGL ES 2.0, vývojové prostředí Xcode, Framework Cocoa Touch a jazyk Objective-c. Je zaměřena na popis tvorby OpenGL hry, z žánru tzv. "line drawing" her.

Abstract

The project deals with the creation of graphical applications for iOS system, describes the basics of OpenGL ES 2.0, development environment Xcode, Cocoa Touch Framework and Objective-C language. It focuses on the description of creation OpenGL game in the genre of "line drawing" games.

Klíčová slova

Mac OS X, Xcode, Cocoa, Touch, Opengl, Opengl ES, shader, App Store, per vertex lighting, schvalovací proces

Keywords

Mac OS X, Xcode, Cocoa, Touch, Opengl, Opengl ES, shader, App Store, per vertex lighting, approval proccess

Citace

Petr Fiala: Vývoj grafických aplikací na iPhone a iPad platformě, semestrální práce, Brno, FIT VUT v Brně, 2012

Vývoj grafických aplikací na iPhone a iPad platformě

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. J. Pečivy, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Bc. Petr Fiala
22.5.2012

Poděkování

Na tomto místě bych rád poděkoval Ing. J. Pečivovi, Ph.D., za cenné rady při realizaci mé diplomové práce.
Dále chci poděkovat Radkovi Vránovi za poskytnutí modelu mapy a letadel, společnosti Touch Art s.r.o. za poskytnutí grafických a zvukových podkladů a Romanu Maštalíři za obecné rady ohledně systému iOS.

© Bc. Petr Fiala, 2012
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Obsah ... 1

1 Úvod ... 3

2 Cíl projektu ... 3

3 Vývojové prostředí ... 4

3.1 Objective-C 2.0 ... 4

3.1.1 Nové datové typy, direktivy ... 4

3.1.2 Třídy ... 5

3.1.3 Metody ... 6

3.1.4 Protokoly ... 6

3.2 Cocoa, Cocoa Touch ... 6

3.2.1 Správa paměti ... 7

3.2.2 Hierarchie tříd, NSObject ... 8

3.2.3 Delegate, Data source ... 9

3.2.4 Důležité části Cocoa Touch ... 9

3.3 Xcode ... 10

3.3.1 Kompilátor ... 11

3.4 iOS simulátor ... 11

3.5 Instruments ... 12

3.6 Zařízení ... 12

4 OpenGL ... 15

4.1 OpenGL ES 2.0 ... 15

4.2 GLSL ES ... 16

4.2.1 Shader program ... 16

4.2.2 Předávání proměnných ... 16

4.2.3 Uniform, Attribute, Varying ... 17

4.2.4 Vertex shader ... 17

4.2.5 Fragment shader ... 18

4.2.6 Souřadnicové systémy ... 18

4.2.7 Projekce ... 19

4.2.8 Textury ... 19

5 Vlastní implementace ... 21

5.1 Osvětlení ... 21

5.1.1 Per vertex osvětlení 21
5.2 Shadow mapping 23
5.2.1 Generování shadowmapy 25
5.2.2 Aplikace shadowmapy 27
5.2.3 PCF filtr 28
5.2.4 Náročnost na výkon 31
5.3 Postprocessing 31
5.3.1 Efekt horkého vzduchu 32
5.4 Modely 35
5.4.1 Editor 35
5.4.2 Konverze do vlastního formátu 36
5.4.3 Renderování modelů 37
5.5 Struktura aplikace 37
5.6 Herní systém 38
5.6.1 Pomocné třídy 39
5.6.2 Systém eventů 39
5.6.3 Generování letadel 39
5.6.4 Střídání dne a noci 40
5.6.5 Skóre 40
5.7 Úrovně efektů na různých zařízeních 41
5.7.1 Rozeznání iPhone/iPad 41
6 Publikace na App Store 42
6.1 Schvalovací proces 43
7 Závěr 44

1 Úvod

V posledních letech se v obrovské míře rozmohl trh s mobilními aplikacemi a to jak App Store od společnosti Apple, tak Android Market od společnosti Google. Apple oznámil, že za jedno čtvrtletí v roce 2011 prodal 4,89 milionů počítačů Mac, 17,07 milionů telefonů iPhone, 12,12 milionů tabletů iPad a 6,62 milionů přehrávačů iPod [6]. Celkové tržby společnosti Apple rostou meziročně o desítky procent. App Store má podle veřejných zdrojů přibližně 6x vyšší výdělky, než konkurenční Android Market. Tento dlouhodobě přetrvávající trend je jedním z důvodů, proč se neustále více vývojářů soustředí právě na Apple.

V této práci se zaměřím jak na vývoj samotné aplikace pro iOS zařízení, tak i na postup při publikování aplikace na App Store, časté problémy, podmínky Apple a možnosti výdělků. Popíši základy grafické knihovny OpenGL ES 2.0, principy a funkce shaderového jazyka GLSL ES. Moje práce je zaměřená na technologie společnosti Apple, proto popíši vývojové prostředí, které mají vývojáři k dispozici, jeho výhody a nevýhody, Objective-C, Cocoa Touch.

Celý projekt je vyvíjen v prostředí Xcode 4.2 a je určen pro systém iOS verze 5.0 a vyšší.

2 Cíl projektu

Jako hlavní cíl jsem si stanovil vytvořit dobře vypadající aplikaci na mobilní zařízení, která bude využívat technologii shaderů. Jedná se o tzv. line drawing hru, jež bývá u dotykových zařízení velmi populární. V této konkrétní aplikaci hráč vidí mapu z ptačí perspektivy a prstem kreslí trajektorii objevujících se letadel. Každé letadlo má předem definovaný cíl letu. Na mapě se nachází letiště odkud letadla nejen vzlétají, ale i přistávají. Na každé ze 4 stran obrazovky je panel znázorňující prostor, do kterého se musí navést správné letadlo. Aplikace je inspirována již existující hrou Flight Control. Dílčí cíle práce jsou:

- implementace dynamického osvětlení
- dynamické stíny
- postprocessing, implementace efektu vlnění teplého vzduchu za tryskovými letadly
- reálně vypadající voda

Ve hře by mělo být implementováno postupné střídání dne a noci a s tím související pohyb slunce z jedné strany obrazovky na druhou, vrhající dynamický stín.

3 Vývojové prostředí

Pro vývojna zařízení s operačním systémem od společnosti Apple, musí být použito zařízení od téže společnosti (Mac) a jedna z novějších verzí operačního systému Mac OS X. To se dá považovat za největší překážku pro nové vývojáře, především z toho důvodu, že Apple zařízení patří mezi ty dražší. Značnému množství lidí nemusí vyhovovat samotný operační systém, z vlastní zkušenosti vím, že přechod na Mac OS X systém, je pro vývojáře zvyklé na systém Windows vcelku bolestný.

3.1 Objective-C 2.0

Objective-C je objektově orientovaná nadstavba jazyka C, založená na programovacím jazyku SmallTalk. Až do doby než ho Apple využil jako nativní jazyk svých API, byl tento jazyk téměř neznámý široké veřejnosti. Objektový přístup tohoto jazyka je pomalejší než například C++, nicméně rychlost je vykoupena výhodami, které Objective-C nabízí. Implementuje jednoduchou dědičnost a oproti C++ je skutečně objektovým jazykem. Narozdíl od C++ nezavádí žádné nové příkazy, STL a podobně. Je přidáno pouze několik direktiv, typů a jedna nová jazyková konstrukce, kterou je zasílání zpráv objektům. Díky tomu je zachována jednoduchost jazyka C.

Deklarace proměnných je možné psát kamkoli do kódu, komentáře jsou stejné jako v C++. Jsou zavedeny 3 nové identifikátory `self`, `super` a `_cmd` [7]. Identifikátor `self` představuje objekt ve kterém se daná metoda volá (sám sebe), `super` to samé s tím rozdílem, že se metody budou vyhledávat v rodičovská třídě a `_cmd` představuje zpracovávanou zprávu.

Přístup k proměnným objektu lze upravit pomocí direktiv `@public`, `@protected`a `@private` [8] s tím, že jejich význam je stejný jako v C++. Pokud není uvedena ani jedna z direktiv, použije se automaticky `@protected`.

U objektů lze využít tzv. "properties", což je způsob přístupu k proměnným v objektu, a které jsou pak přístupné přes tečkovou notaci. Jako návratové hodnoty a argumenty funkcí a metod lze používat jak základní datové typy a struktury stejně jako v C, tak i Objective-C objekty. Objective-C objekt se uvádí vždy jako ukazatel.

S výhodou lze využít Objective-C++, které zavádí kompatibilitu s C++, ale zároveň umožňuje Objective-C volání metod. Nicméně není možné v jednom zdrojovém/hlavičkovém souboru kombinovat deklarace/definice C++ a Objective-C tříd. Hlavičkový soubor má klasicky příponu ".h", zdrojový soubor je u Objective-C ".m" a u Objective-C++ ".mm".

3.1.1 Nové datové typy, direktivy

Prázdný ukazatel na objekt má nově identifikátor `nil` a prázdný ukazatel na třídu `Nil`. Pro boolean typ je nově použit identifikátor BOOL a může nabývat hodnot YES nebo NO. Typ SEL, skladuje ukazatel na metodu a jde získat pomocí `@selector()`. Jako obecný ukazatel na objekt se používá datový typ `id`, který je obdobou céčkového `(void *)`.

V Objective-C zavádí direktivu `#import`, která oproti `#include`, zavádí kontrolu cyklického vkládání hlavičkových souborů, je tedy možné uvádět křížové reference hlavičkových souborů bez nutnosti vkládat další ošetřují kód. Samotný `#include` lze samozřejmě použít také.

Většina nové funkčnosti je zavedena pomocí direktiv začínajících znakem '@', patří mezi ně: `@class`, `@protocol`, `@required`, `@optional`, `@end`, `@interface`, `@public`, `@package`, `@protected`, `@private`, `@property`, `@end`, `@implementation`, `@synthesize`, `@dynamic`, `@end`, `@throw`, `@try`, `@catch`, `@finally`, `@synchronized`, `@autoreleasepool`, `@selector`, `@encode`, `@compatibility_alias` a `@"string"` [8].

3.1.2 Třídy

Třídy v Objective-C se deklarujípomocí direktiv `@interface` a `@end` [7]. Za `@interface` se uvádí jméno třídy a zaním pak volitelně seznam protokolů. Mezi těmito direktivami se pak volitelně může vyskytovat jeden pár složených závorek, ve kterých mohou být uvedeny proměnné a direktivy pro upravení přístupu k těmto proměnným, následuje volitelný seznam properties a nakonec hlavičky metod. U metod není možné nijak měnit přístupová práva.Jediná možnost, jak udělat metody privátní, je ve zdrojovém souboru uvést znovu interface část, nyní již bez specifikace děděné třídy a protokolů, metody vní pak budou privátní. Deklarace vlastní třídy může vypadat například takto:

```
@interface MyClass:NSObject <MyProtocol1, MyProtocol2>
{
    int promenna1;    // tato promenna voni
    NSString*promenna2; /* tato jestevic */
}
@property (nonatomic, retain) promenna2;
- (int)doSomethingWithValue:(int)val andWithArray:(NSArray *)array;
- (int)doSomethingElse;
+ (int)doSomethingElse2;
@end
```

Definice třídy pak například takto:

```
@implementation
@syntesize promenna2;
- (int)doSomethingWithValue:(int)val andWithArray:(NSArray *)array
{
    return promenna1+val + [array count];
}
- (int)doSomethingElse
{
    return promenna1;
}
+ (int)doSomethingElse2
{
    return 5;
}
@end
```

3.1.3 Metody

V Objective-C rozeznáváme 2 typy metod. Jsou to metody instanční a metody tříd. Třída sama o sobě je vlastně objektem, tudíž je schopna přijímat zprávy, nicméně u toho nemůže využít proměnných dané třídy. Metody tříd, v terminologii C++ statické metody, obsahují nazačátku deklarace i definice znak '+'. Instanční metody mají plný přístup k proměnným dané třídy a jdou volat až přes instance třídy (na vytvořený objekt). Instanční metody mají nazačátku znak '-'.

Oba typy metod se dědí stejným způsobem a to tak, že se nejprve hledají metody ve třídě samotné a až následně ve třídách rodičovských, oproti tomu v C++ statické metody nejdou dědit vůbec.

3.1.4 Protokoly

Třídy v Objective-C mohou mít pouze jednoduchou dědičnost, tzn. třída má rodiče, rodič má svého vlastního rodiče atd., co ale dělat pro volaní metod, které nejsou nikde v dané hierarchii tříd? Tento problém elegantně řeší právě protokoly.

Protokoly se deklarují pomocí direktiv `@protocol` a `@end.` Neobsahují žádné proměnné, obsahují pouze deklarace instančních nebo třídních metod. Zadané metody jsou pak ovlivněny volitelnými direktivami `@optionala` `@required`. Pokud je metoda uvedena jako optional, není ji pak nutné v objektech, které protokol adoptují implementovat. Pokud není zadána ani jedna z direktiv, tak je defaultně použito `@required`.

Adoptované (použité) protokoly se uvádí v interface části tříd, ve špičatých závorkách, protokoly mají pouze deklarační část. Samotná definice metod se provádí až v metodách třídy, které ho adoptují [9].

3.2 Cocoa, Cocoa Touch

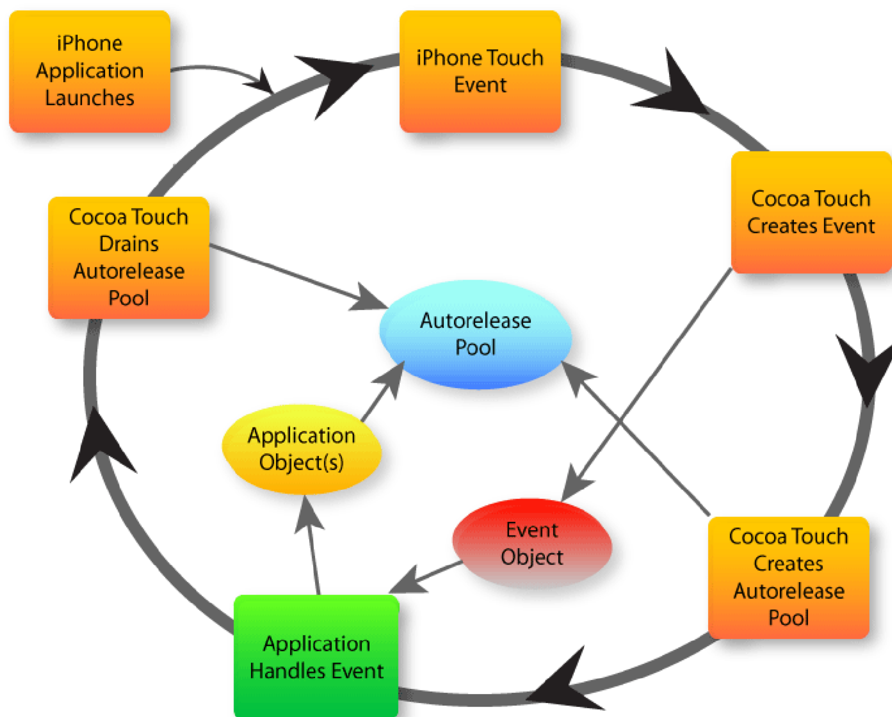
Cocoa je nativní API společnosti Apple, určené pro vývoj aplikací na platformu Mac OS X.

Cocoa Touch API určené pro vývoj aplikaci pro operační systém iOS, který je na zařízeních jako jsou iPhone, iPad, iPod Touch. Cocoa Touch je založeno na Cocoa a lze najít spousty podobností a stejných základních komponent. Obecně lze Cocoa Touch považovat za jakousi ořezanou verzi Cocoa, která je přizpůsobena mobilním zařízením typu iPhone. Oboje API jsou objektově orientované a jsou určené pro programovací jazyk Objective-C.

Jsou to rozsáhlé balíky knihoven zastřešující většinu funkčnosti, kterou by mohli vývojáři potřebovat.

Cyklus programu, tzv. "runloop", je smyčka opakující se stále dokola, až do doby ukončení nebo přesunutí aplikace na pozadí.Runloop začíná spuštěním programu, následuje zpracování dotyků a událostí vyvolaných samotným Cocoa Touch, jako například časovač. Z těchto událostí se vytvoří objekty, vytvoří se tzv. "autorelease pool" (bude vysvětleno později) a objekty událostí se do onoho autorelease pool automaticky vloží. Nyní proběhne zpracování událostí, tuto část programuje vývojář a zabere naprostou většinu času aplikace. Poté se automaticky vyprazdňuje autorelease pool a cyklus programu začíná od začátku.

Pro lepší představu uvedu grafické znázornění smyčky programu:



Obrázek 3.1: Cocoa Touch runloop [10]

3.2.1 Správa paměti

Pro API Cocoa je použit Garbage Collector a tzv. čítač referencí. Pro Cocoa Touch je k dispozici pouze čítač referencí. Počítání referencí (reference counting) je jednoduchá technika při které je každému objektu přiřazen čítač, jehož hodnota představuje počet vlastníků (uživatelů objektu). Při vytvoření objektu je tento čítač nastaven na hodnotu 1 a tvůrce objektu se obrazně řečeno stává jeho vlastníkem. Retain count se dá zvyšovat a snižovat pomocí metod `retain/release` [9].

Pokud chce nějaký objekt využívat jiný objekt, tak má možnost zvýšit jeho čítač referencí a tím se zajistí, že daný objekt nebude uvolněn z paměti.

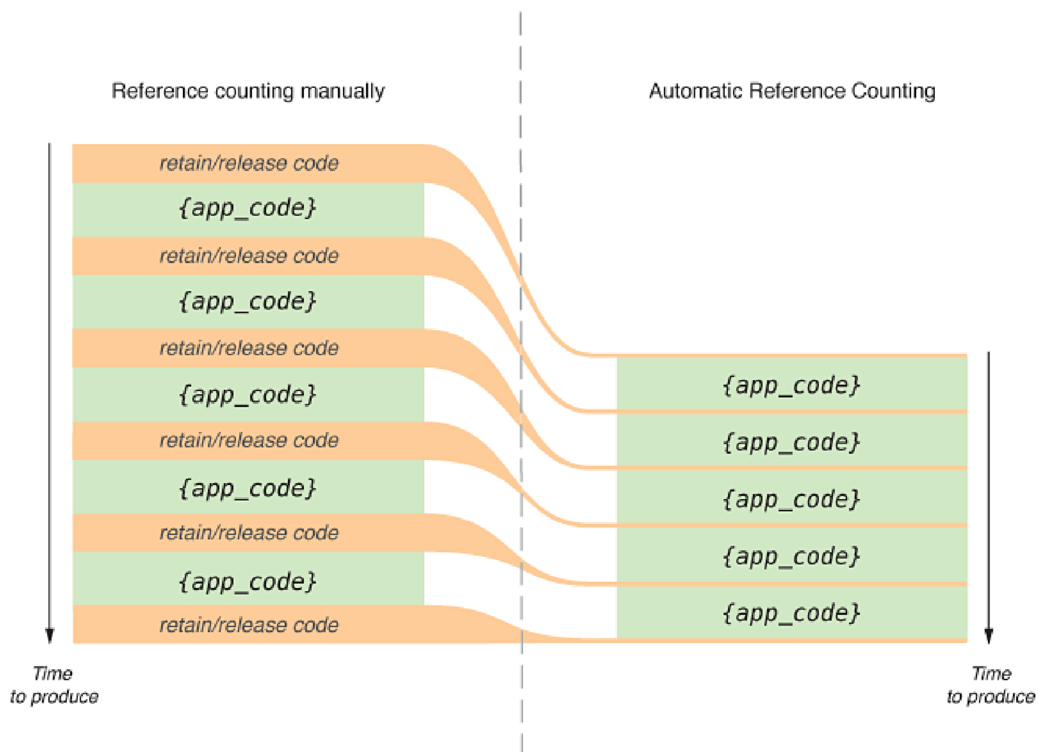
Pokud dosáhne retain count hodnoty 0, v objektu je automaticky zavolána metoda `dealloc` a objekt je uvolněn z paměti. Nyní se nesmí na danýukazatel na objekt volat žádné metody, protože by došlo k chybnému přístupu do paměti a následnému pádu programu. Nastávají situace, kdy je potřeba použít objekt pouze dočasně, nicméně není žádoucí, aby se uvolnil okamžitě, k tomu slouží metoda `autorelease` [10], která proměnnou uloží do tzv. "autorelease pool" a po doběhnutí aktuálního cyklu aplikace, se všem proměnným v autorelease pool sníží hodnota retain count o jedničku. Aby nedocházelo k memory leakům, tak každé volání metody `alloc` a `retain` by mělo mít protikladné volání `release` nebo `autorelease`.

Velké množství metod, vracejících nějaký objekt, nám tento objekt tzv. vypůjčí, tzn. zvýší jeho retain count o 1, ale zároveň na něj aplikuje `autorelease`. V případě, že objekt volající vytvářecí metodu nezvýší jeho retain count (nestane se jeho vlastníkem), tak je daný objekt k dispozici jen na aktuální cyklus aplikace a posléze se jeho retain count sníží o 1.

Od verze Xcode 4.2 je zaveden nový kompilátor využívající techniku ARC (automatic reference counting) [16], což je obrovské zjednodušení hlavně pro začínající vývojáře. Tato technika spočívá v tom, že automaticky během překladu přidává do kódu volání `retain/release/autorelease`. S tím souvisí úprava oproti staré verzi, `retain/release/autorelease` je zakázáno volat, z kódu úplně vymizí. Na první pohled se může zdát, že něco takového není možné implementovat, překladač nemá žádnou věštící kouli, aby věděl, kdy chce uživatel, který objekt přivlastnit nebo naopak odstranit, nicméně tento proces je plně deterministický a z vlastní zkušenosti mohu říci, že je velmi spolehlivý. Počítání referencí není především pro začínající vývojáře vůbec jednoduché a velmi často způsobuje mnohdy náhodné pády aplikací. Díky ARC se odstraní obrovské množství chyb způsobených špatným použitím retain counteru objektů, tvorba aplikace je snazší a rychlejší.

Výhodou je, že ARC je použito pouze během kompilace, výsledný kód je plně kompatibilní, protože stále obsahuje původní správu paměti pomocí čítání referencí.

Kvůli kompatibilitě a využití starších knihoven má vývojář možnost pro konkrétní soubory v projektu ARC vypnout a používat starší metodu, případně je možnost ARC vypnout v celém projektu úplně.



Obrázek 3.2: ARC [12]

3.2.2 Hierarchie tříd, NSObject

Veškeré třídy v Cocoa i Cocoa Touch dědí od základní třídy, kterou je `NSObject` [9]. Tato třída tvoří základní stavební kámen cele hierarchie tříd v Cocoa i Cocoa Touch API. K lepšímu pochopení třídy `NSObject` zde vypíši některé její často používané metody:

`+alloc`, volá se přímo na třídu, provede instanciaci třídy, vytvoří objekt

`-init`, volá se již na vytvořený objekt, provede prvotní inicializaci (například inicializaci proměnných)
`-copy`, vytvoří kopii objektu, retain count nového objektu nastaví na 1
`-dealloc`, volá se automaticky po dosažení retain count 0
`+class`, vrací třídu jako typ `Class`
`+conformsToProtocol:`, vrací BOOL hodnotu v závislosti na tom, zda třída adoptuje zadaný protokol
`+description`, vrací řetězec s jednoduchým popisem třídy

Třída NSObject adoptuje stejnojmenný protokol NSObject, obsahující metody jako jsou `retain, release, autorelease, retainCount, isKindOfClass:` nebo například `isEqual:`.

3.2.3 Delegate, Data source

Delegace je návrhový vzor, hojně využívaný v Cocoa i Cocoa Touch API [17]. Princip delegování není nijak složitý, než aby se objekt postaral o všechno sám, tak má v sobě uložený ukazatel na instanci třídy (na pomocný objekt), který adoptuje delegáta. Delegáta může adoptovat sám hlavní objekt, v tom případě pak tento objekt musí implementovat metody delegáta přímo v sobě.

Delegát je jednoduše protokol, obsahující povinné i nepovinné metody, které pak implementuje objekt využívající tohoto delegáta. Díky tomu je možné ovlivňovat chování objektu, reagovat na určité události a měnit vzhled.

Data source je velmi podobný delegátu, ale na rozdíl od něj je určen k poskytování dat objektu. Pěkná ukázka delegáta a data source je u třídy `UITableView` (komponenta pro tabulku), v delegátu jsou metody typu:
`-tableView:heightForRowAtIndexPath:`, kde vývojář volí výšku jednotlivých buněk
`-tableView:didSelectRowAtIndexPath:`, zavolá se automaticky po kliknutí na buňku nebo například `-tableView:viewForHeaderInSection:`, umožňuje změnit hlavičku určité sekce tabulky.

Data source pro třídu `UITableView`, pak zavádí metody jako jsou:
`-tableView:cellForRowAtIndexPath:`, volá se při potřebě zobrazit určitou buňku nebo
`-tableView:numberOfRowsInSection:` pro zjištění počtu buněk, pro každou sekci tabulky.
Samotná základní kostra aplikace, povinně implementuje delegát `UIApplicationDelegate`, s povinnou metodou `-application:didFinishLaunchingWithOptions:`, která se volá vždy po dokončení spouštění aplikace.

3.2.4 Důležité části Cocoa Touch

Každá aplikace musí obsahovat minimálně framework (skupinu knihoven) "Foundation", ten obsahuje všechny základní třídy, definice, datové typy, potřebné ke spuštění programu. Je v něm umístěn například i zmiňovaný `NSObject`.

Naprostá většina aplikací pak obsahuje Framework "UIKit", který obsahuje všechny grafické komponenty a některé třídy, struktury, definice spojené se vzhledem aplikace.

Pro OpenGL ES 1.1 a 2.0 slouží Framework "OpenGLES", obsahuje nejen funkce OpenGL ES, ale také EGL, jednoduše řečeno vrstvu spojující framebuffer s obrazovkou zařízení.

Nově je v iOS 5.0 zaveden framework "GLKit", který velmi usnadňuje zavedení OpenGL kontextu, obsahuje komponentu typu view, která provede vytvoření render, frame a depth bufferu a provede inicializaci, takže vývojář se již může starat čistě jen o OpenGL. GLKit také obsahuje struktury a funkce pro práci s vektory, maticemi, quaterniony a množství pomocných funkcí řešících běžné problémy v OpenGL.

3.3 Xcode

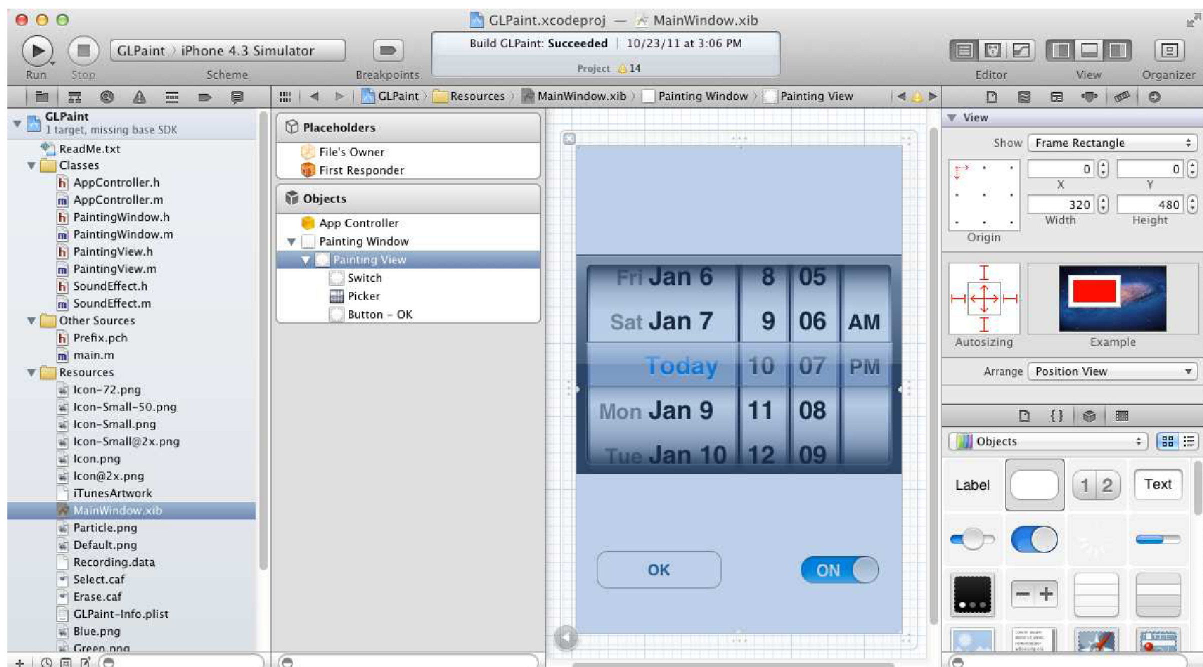
Xcode je vývojářský nástroj od společnosti Apple, je to sada nástrojů integrovaných do jednoho velkého balíku. Prostředí Xcode je úzce svázános frameworky Cocoa a Cocoa Touch.

Tento vývojový nástroj nepatří mezi nejrychlejší a je to poznat především na starších strojích, nicméně díky tomu co nabízí se z něj postupem času stal jeden z nejsilnějších vývojových nástrojů vůbec.

Xcode je velmi dobře nastavitelný, změna klávesových zkratek je samozřejmostí, lze změnit barvy celého prostředí, ale například také chování překladače, ve významu co se má stát pokud překlad selže, pokud je nalezen warning a mnohé další události lze spojit s množstvím akcí.

Součástí Xcode je velmi dobře zpracovaný systém dokumentace, v nastavení lze kdykoli stáhnout vybrané balíčky s dokumentací pro vývoj na iOS, na Mac OS X a další. Doplňování kódu během psaní je rychlé a velmi inteligentní.Jako hlavní výhodu považuji kompilátor, bude popsán dále.

V Xcode je od verze 4 integrován kvalitní editor uživatelského rozhraní, ve kterém lze vložené komponenty snadno a rychle propojovat přímo s kódem.



Obrázek 3.3: Xcode - Interface Builder

3.3.1 Kompilátor

LLVM (back-end) v kombinaci s Clang (front-end) tvoří kompilátor, který je rychlejší než GCC, potřebuje méně paměti a má volnější licenci, díky čemuž jej Apple může více integrovat do svého Xcode. Clang poskytuje velmi dobrou diagnostiku chyb ve zdrojovém kódu a poskytuje více informací již během překladu než GCC [11].

Z uživatelského hlediska je třeba vyzdvihnout umožnění kontroly syntaxe bez nutnosti manuální kompilace kódu.

3.4 iOS simulátor

iOS simulátor je velmi kvalitní nástroj pro testování iPhone, iPad a iPod Touch aplikací bez nutnosti nahrávat data na skutečné zařízení. Ve srovnání s konkurenčním simulátorem pro Android, je iOS simulátor velmi rychlý, pro standardní uživatelské rozhraní obvykle rychlejší než skutečné zařízení. Problém s testováním může nastat v případě OpenGL aplikace a simulování na starším stroji, je to poznat především v režimu pro iPad, kvůli relativně vysokému rozlišení iPadu.

Po přeložení aplikace v Xcode se zacílením na simulátor, se přesune aplikace simulátoru na popředí (případně se spustí) a testovaná aplikace se začne načítat.

Simulátor je samostatně běžící aplikace, která ke svému životu nepotřebuje Xcode, dá se dobře využít například pro testování vzhledu webových stránek na iOS zařízeních. Podporuje rotaci zařízení, simulaci memory warningu (událost upozorňující na docházející paměť), lze v něm nastavit výstup na televizi pro testování AirPlay a AirMirroring. Umožňuje debugování uživatelského rozhraní za pomocí možnosti zapnout režim zvýraznění překrytí jednotlivých vrstev UI aplikace nebo například zvýraznění špatně zarovnaných obrázků.

Simulátoru lze nastavit 3 úrovně zoomu 50%, 75% a 100%, lze zvolit verzi iOS, která se má simulovat (verze musí být nainstalována) a v neposlední řadě lze přepínat mezi iPad, iPhone a verzemi s retina obrazovkou (dvojnásobné rozlišení).

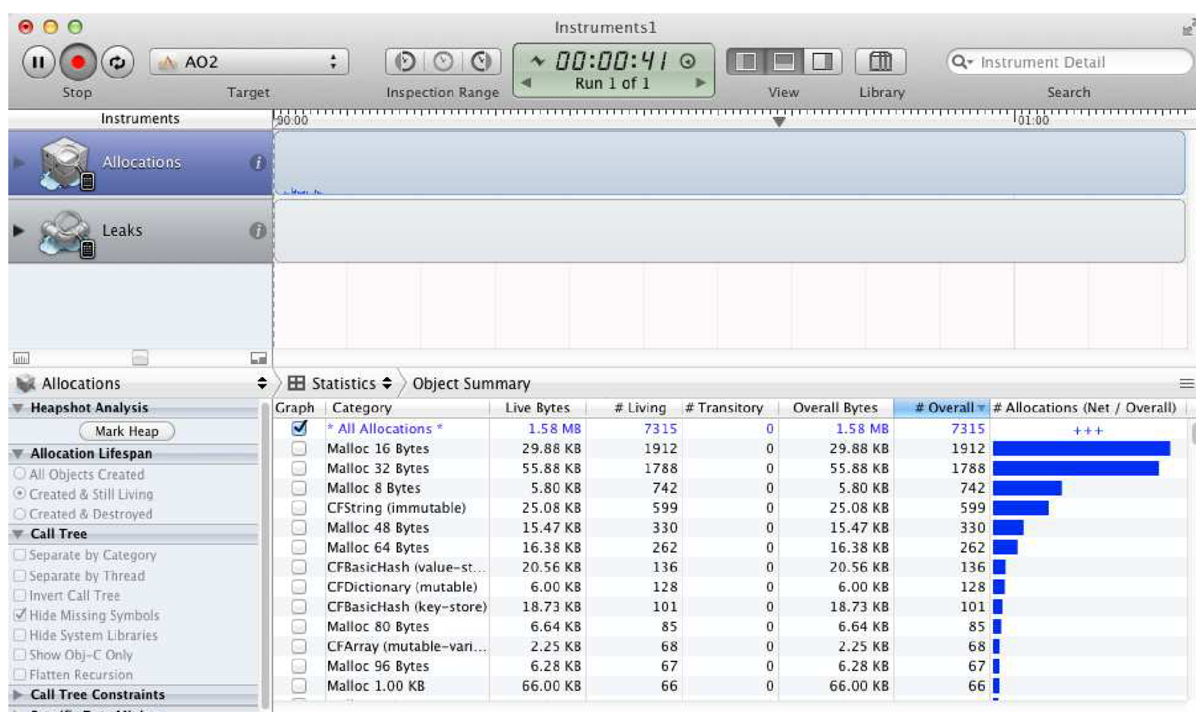
Obrázek 3.4: iOS Simulator

3.5 Instruments

Instruments je nástroj pro dynamické sledování a profilování aplikací. Nástroj účinně pomáhá v ladění aplikací, hledání memory leaků, zjištění bottleneck výkonu aplikace, sledování jednotlivých procesů, sledování alokací, zombie procesů nebo dokonce i sledování energetické spotřeby zařízení, což je dobře využitelné při psaní aplikací využívajících GPS a podobně.

Osobně ho využívám především k detailnímu zjištění využití paměti aplikace během jejího běhu. Instruments umožňují uložení naměřených dat k pozdější analýze.

Jde aplikovat jak na aplikace běžící na skutečném zařízení tak na simulátor. Je ho možné spustit a používat samostatně, avšak běžně se používá ve spojení s Xcode a odtud ho lze spustit pomocí režimu kompilace "build and profile".



Obrázek 3.5: Instruments

3.6 Zařízení

Operační systém iOS běží na několika zařízeních společnosti Apple. V současné chvíli jsou rozlišovány 3 základní typy zařízení: iPhone - obrázek 3.6 vpravo, iPad - obrázek 3.6 uprostřed, iPod Touch - obrázek 3.6 uprostřed (+ Apple TV) [18] s tím, že každé z nich se dělí na několik generací. V následujících tabulkách jsou uvedeny informace pro každý typ zařízení, zajímavé z pohledu vývojáře.

	iPhone	iPhone 3G	iPhone3GS	iPhone 4	iPhone 4S
Nejvyšší OS	iPhone OS 3.1.3	iOS 4.2.1	iOS 5.1		
Paměť	128MB (137MHz)		256MB (200MHz)	512MB (200MHz)	
OpenGL	ES 1.1		ES 2.0		
Grafický čip	PowerVR MBX Lite 3D (103MHz)		PowerVR SGX535 (150MHz, 200MHz)		PowerVR SGX543MP2
Rozlišení	320x480 (3:2)			640x960 (3:2)	

Tabulka 3.1: Generace zařízení iPhone

Zařízení iPhone a iPhone 3G (první dvě verze) již nejsou společností Apple podporované, nedají se koupit v oficiálních obchodech a nevydává se na ně nový iOS. Jak je vidět v tabulce 3.1 tak tyto dvě generace podporovaly pouze OpenGL ES 1.1 s fixní pipeline, tato dvě zařízení jsou již zastaralá a tvoří jen velmi malý, zanedbatelný díl všech iPhone zařízení, tudíž často nemá smysl aplikace psát univerzálně pro obě verze OpenGL.

	iPod Touch 1st gen.	iPod Touch 2nd gen.	iPod Touch 3rd gen.	iPod Touch 4th gen.
Nejvyšší OS	iPhone OS 3.1.3	iOS 4.2.1	iOS 5.1	
Paměť	128MB		256MB	
OpenGL	ES 1.1		ES 2.0	
Grafický čip	PowerVR MBX Lite 3D (103MHz)		PowerVR SGX535 (110MHz, 200MHz)	
Rozlišení	320x480 (3:2)			640x960 (3:2)

Tabulka 3.2: Generace zařízení iPod Touch

iPod Touch se od iPhone liší především v tom, že nemá GSM a GPS modul. Další na první pohled hůře postřehnutelný rozdíl je méně kvalitní display u zařízení iPod Touch.

	iPad	iPad 2	iPad (3rd gen.)
Nejvyšší OS	iOS 5.1		
Paměť	256MB	512MB	1GB
OpenGL	ES 2.0		
Grafický čip	PowerVR SGX535	PowerVR SGX543MP2	PowerVR SGX543MP4
Rozlišení	1024x768 (4:3)		2048x1536 (4:3)

Tabulka 3.3: Generace zařízení iPad

Zařízení iPad jsou kompatibilní s aplikacemi pro iPhone (iPod Touch), menší display iPhone se zobrazí uprostřed displaye iPadu a volitelně ho lze 2x roztáhnout.

Mobilní zařízení se systémy iOS mají celkem 4různá rozlišení. Kvůli kompatibilitě aplikací jsou z pohledu vývojáře základní rozlišení pouze 2. iPhone a iPod Touch má základní rozlišení 320x480 a

nové generace, s tzv. retina displayem mají přesně dvojnásobek 640x960. V případě iPadu je rozlišení nejnovější generace také přesně dvojnásobné.

Díky tomu je možné bez problémů spustit ty stejné aplikace na zařízeních s klasickým i retina rozlišením. Vevnitř iOS je to řešené důmyslně tak, že aplikace běžící na zařízení s retina displayem nevykreslují po pixelech, ale po půlpixelech. Jeden pixel v základním rozlišením je tedy ekvivalentní čtyřem pixelům na retině. Velikost a pozice všech grafických komponent, jako jsou např. tlačítka a podobné, se zadává vždy v základním rozlišení, avšak na retina display se vykreslí s dvojnásobným detailem. Co se týká obrázků, tak iOS na zařízení s retina displayem vyhledává obrázky s poupraveným jménem, které mají na konci "@2x" (například image@2x.png), tyto obrázky vývojář ukládá s dvojnásobným rozlišením, pokud takový obrázek není nalezen, použije se standardní obrázek. Díky tomuto je úprava aplikace pro vyšší rozlišení triviální, stačí do projektu přidat upravené obrázky. Oproti klasickému iOS GUI, je v případě OpenGL nutné nastavit ručně několik parametrů, aby bylo možné vykreslovat s dvojnásobným rozlišením.

Obrázek 3.6: Mobilní zařízení společnosti Apple (iPod Touch, iPad, iPhone) [13]

4 OpenGL

OpenGL je čistě grafická multiplatformní knihovna, která nám umožňuje zobrazit trojrozměrné prostředí, cokoli dalšího, jako například přehrávání hudby, načítání modelů a podobně si programátor musí vytvořit sám.

Vyvinula ho společnost Silicon Graphics Inc. V roce 1992 a je spravováno neziskovým technologickým konsorciem Khronos Group [4].

V jednoduchosti představuje OpenGL propojení mezi CPU a GPU, musí být podporováno grafickou kartou zařízení, na kterémmá být grafická aplikacespuštěna.OpenGL je vytvořeno tak, aby ho bylo možno použít v téměř libovolném programovacím jazyce jako jsou například C, C++, C#, Objective-C, Java, Javascript, Perl a mnohé další.

Od doby vytvoření prošlo OpenGL obrovskými změnami. Za největší změnu se dá považovat zavedení programovatelné pipeline ve verzi 2.0 v září roku 2004. Programovatelná pipeline umožnila nízko úrovňové ovlivnění výsledného vzhledu scény, umožňuje využít spoustu technik, které by bylo buď velmi komplikované nebo nemožné implementovat na fixní pipeline.

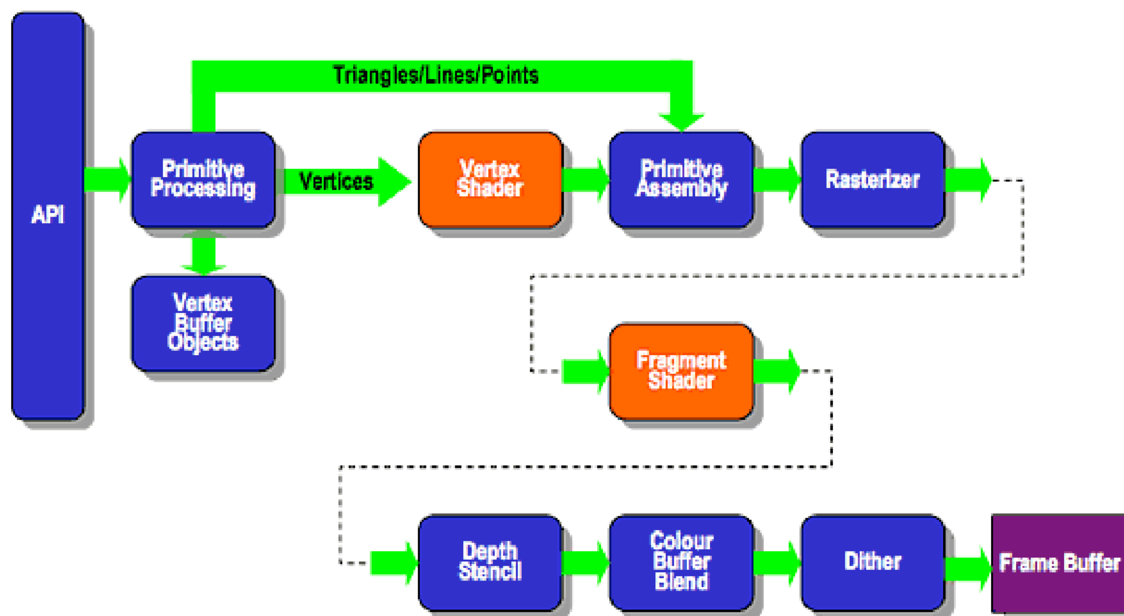
4.1 OpenGL ES 2.0

OpenGL ES (Embeded Systems) je podmnožinou klasické verze OpenGL, vytvořenou v roce 2007 a je využívaná především pro mobilní zařízení. OpenGL ES 1.x vychází z OpenGL 1.5 a Opengl ES 2.0 vychází z OpenGL 3.x. Je vytvořeno tak, aby bylo co nejrychlejší, jsou odstraněny funkce, které nejsou přímo nutné k fungování grafické knihovny. Veškeré immediate mode funkce, jako například `glBegin()`, `glEnd()`, `glVertex*()`, byly z OpenGL ES 2.0 kompletně vyjmuty. Verze ES podporuje pouze 3 typy primitiv: bod, úsečka a trojúhelník. Čtverce a polygony nejsou podporovány. Je odstraněna podpora display listů. Vykreslování primitiv je omezeno pouze na dvě funkce `glDrawElements()` a `glDrawArrays()` [4].

OpenGL ES 2.0 obsahuje plně programovatelnou pipeline a není kompatibilní s ES 1.x, fixní pipeline byla kompletně odstraněna, využít se dá jedině nasimulováním přes shadery. K dispozici je vertex a fragment shader, geometry shader není zaveden.

K vytvoření shaderů se používá zjednodušená verze OpenGL shader language – GLSL ES. Z verze OpenGL ES 2.0 byly odstraněny funkce `glVertexPointer()` a `glNormalPointer()`, místo nich se používá `glVertexAttribPointer()` [4].

Výpočty shaderů běží na GPU, z toho důvodu jsou v OpenGL zavedeny tzv. Buffer objekty. Ty jsou uloženy v grafické paměti a nemusí se v každém kreslícím cyklu znovu posílat na zpracování.



Obrázek 4.1: OpenGL ES 2.0 – programovatelná pipeline [1]

4.2 GLSL ES

GLSL ES je shaderový jazyk velmi podobný jazyku C, v němž se píší již zmiňované shadery. V OpenGL aplikaci používáme tzv. Shader program, který se skládá z jedné dvojice, vertex shaderu a fragment shaderu, od každého právě jeden [4]. Shader Program se obvykle kompiluje za běhu aplikace a běžně se překompilovává mezi jednotlivými vykreslovanými snímky, podle aktuální potřeby. Tato kompilace je velmi rychlá a téměř neovlivní výslednou rychlost aplikace.

4.2.1 Shader program

Vytvoření shader programu se skládá z několika kroků. Nejprve se zvlášť zkompilují vertex a fragment shader (`glCompileShader`), pak se vytvoří samotný shader program (`glCreateProgram`), nyní se již zkompilované shadery slinkují s shader programem (`glLinkProgram`), v tomto bodě je již shader program připravený k použití. Výsledný shader program může být zkontrolován funkcí `glValidateProgram`, nicméně to není nutné.

4.2.2 Předávání proměnných

Shadery běží jako samostatné programy na GPU, nijak nesdílí prostor proměnných s aplikací, je tudíž nutné nějakým způsobem do shader programu naše proměnné dostat. Používají se k tomu tzv. Uniform a Attribute proměnné [4].

Po vytvoření shader programu je třeba zjistit adresy (identifikátory) těchto proměnných za pomocí funkcí `glGetUniformLocation` a `glGetAttribLocation`. Tyto adresy jsou pak využity k propojení proměnných v aplikaci s proměnnými v shaderech.

Nyní, když známe adresy, jednoduše voláme funkce `glUniform{1234}{if}`, `glUniform{1234}{if}v`, `glUniformMatrix{234}{fv}` pro uniform proměnné a `glVertexAttrib{1234}f`, `glVertexAttrib{1234}fv`, `glVertexAttribPointer`, pro attribute proměnné.

4.2.3 Uniform, Attribute, Varying

Proměnné typu uniform jsou globální pro celý běh shaderů, nemění se v rámci jednoho volání renderovacích funkcí `glDrawElements()` nebo `glDrawArrays()` a jdou použít jak ve vertex tak ve fragment shaderu. Využívá se obvykle pro předání transformační matice vykreslovaného objektu, pozice a vlastnosti světel či materiálů nebo třeba pro předání aktuálního času, u časově závislých efektů.

Attribute proměnné jsou přístupné pouze ve vertex shaderu a mohou mít stejný význam jako proměnné uniform nebo při využití `glVertexAttribPointer` bude mít daná proměnná zvlášť hodnotu pro každý vrchol. Využívá se k předání pozic, normál, texturovacích souřadnic nebo třeba i barvy vrcholů.

Varying proměnné se nepředávají z aplikace, ale jsou využity pouze v rámci jednoho běhu shaderu. Jsou vždy uvedeny jak ve vertex tak ve fragment shaderu. Tyto proměnné se ve vertex shaderu naplní zvlášť pro každý vrchol a při rasterizaci, je při každém volání fragment shaderu v těchto proměnných interpolovaná hodnota. Pro představu, ve vertex shaderu se zadají 3 texturovací souřadnice pro texturu trojúhelníka. Když se pak trojúhelník rasterizuje, fragment shader dostává interpolované hodnoty texturovacích souřadnic mezi jednotlivými vrcholy [4].

4.2.4 Vertex shader

Vertex shader se spouští zvlášť pro každý vrchol objektů, který projde funkcemi `glDrawElements()` nebo `glDrawArrays()`. Vertex shaderem je možné ovlivnit pozici daného vrcholu, ale i texturovací souřadnice, normálu a barvu. Pomocí vertex shaderu není možné přidávat nebo odebírat vrcholy.

Vertex shader nemá žádný přístup k vedlejším vertexům, tudíž nezná topologii modelu, všechny výpočty se dělají v závislosti na aktuálním jednom vertexu.

Hlavní funkcí vertex shaderu je převést prostorové souřadnice vrcholu na dvojrozměrné souřadnice na obrazovce. Běžně se využívá k výpočtu osvětlení a různým efektům využívajícím změnu pozice vrcholů.

Vertex shader obsahuje 2 výstupní proměnné: `gl_Position`, který představuje transformovanou pozici fragmentu a `gl_PointSize` určující velikost bodu, v případě že se bude rasterizovat bod.

4.2.5 Fragment shader

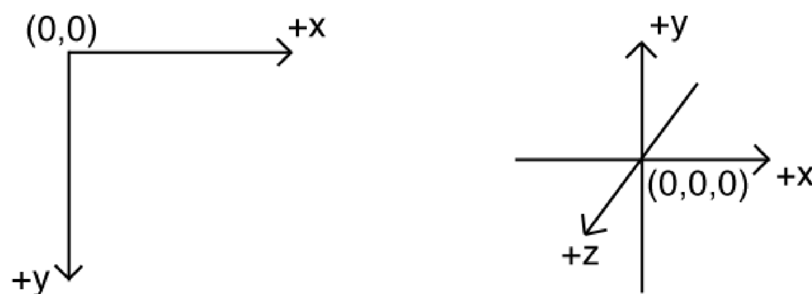
Fragment shader, někdy také pixel shader slouží k výsledné rasterizaci primitiv. Volá se pro každý fragment, který projde testem na hloubku. V případě že je zapnutý blending, tak se fragment shader provede vždy. V běžné grafické aplikaci je volání fragment shaderu několikanásobně vyšší něž vertex shaderu, proto je zvláště nutné tyto shadery dobře optimalizovat.

Fragment shader obsahuje 3 vstupní proměnné: `gl_FragCoord` představující pozici v okně (ve framebufferu), `gl_FrontFacing` říkající jestli normála vykreslované primitivy směřuje k obrazovce nebo od ní a `gl_PointCoord` pozici fragmentu v rámci bodu (při rasterizaci bodu).

Dále obsahuje 2 výstupní proměnné: `gl_FragColor` určující výslednou barvu fragmentu a `gl_FragData[n]` pro renderování do více cílů zároveň, přičemž `n` udává index bufferu.

4.2.6 Souřadnicové systémy

OpenGL používá Kartézský souřadnicový systém s bodem (0,0,0) uprostřed a osou Y směřující nahoru (obrázek 3.2 vpravo), oproti 2D souřadnicovému systému pro GUI v iOS, který má bod (0,0) vlevo nahoře (obrázek 3.2 vlevo).



Obrázek 4.2: Souřadnicové systémy

V 3D grafice je obecně rozlišováno mezi několika souřadnicovými "podsystémy" v rámci Kartézského souřadnicového systému [19]. Jsou to object space, world space, camera space a screen space. Při vykreslování scény se postupně převádí z jednoho systému do dalšího.

Object space je lokální souřadnicový systém modelu. Například u modelů letadel v mé práci je bod (0,0,0) ve středu modelu, směrem po ose –x je levé křídlo letadla atd.

World space (také Model space) je souřadnicový systém, ve kterém se modely napozicují, zrotují a zvětší, aby byly ve světě na správném místě.

Camera space (také Eye space nebo View space) je systém, ve kterém se napozicuje, zrotuje a zvětší celý svět tak, jako by ho viděla imaginární kamera (pohled hráče). Je důležité si uvědomit, že v OpenGL nic jako kamera neexistuje, pokud je pohnuto imaginární kamerou dopředu je to stejné, jako by se posunul celý svět dozadu.

Screen space (také Clip space nebo Projection space) je pak výsledné převedení všech 3d souřadnic na 2d souřadnice, aby bylo možné svět vykreslit na obrazovku.

4.2.7 Projekce

Jsou rozlišovány 2 základní projekce, perspektivní a ortogonální (pravoúhlou). Obecně se perspektivní projekce se používá na vykreslování 3D a pravoúhlá na 2D, ale není to žádné pravidlo, obě dvě projekce lze použít na 2D i 3D.

Pokud mají být vykresleny dva různé body v perspektivní projekci a ty samé body jsou posunuty dál do scény, na obrazovce se zobrazí blíže sobě, toto je základní princip perspektivní projekce a stejným způsobem člověk vidí i reálný svět. Oproti tomu v pravoúhlé projekci nezáleží na hloubce scény, objekty se zobrazí vždy stejně velké.

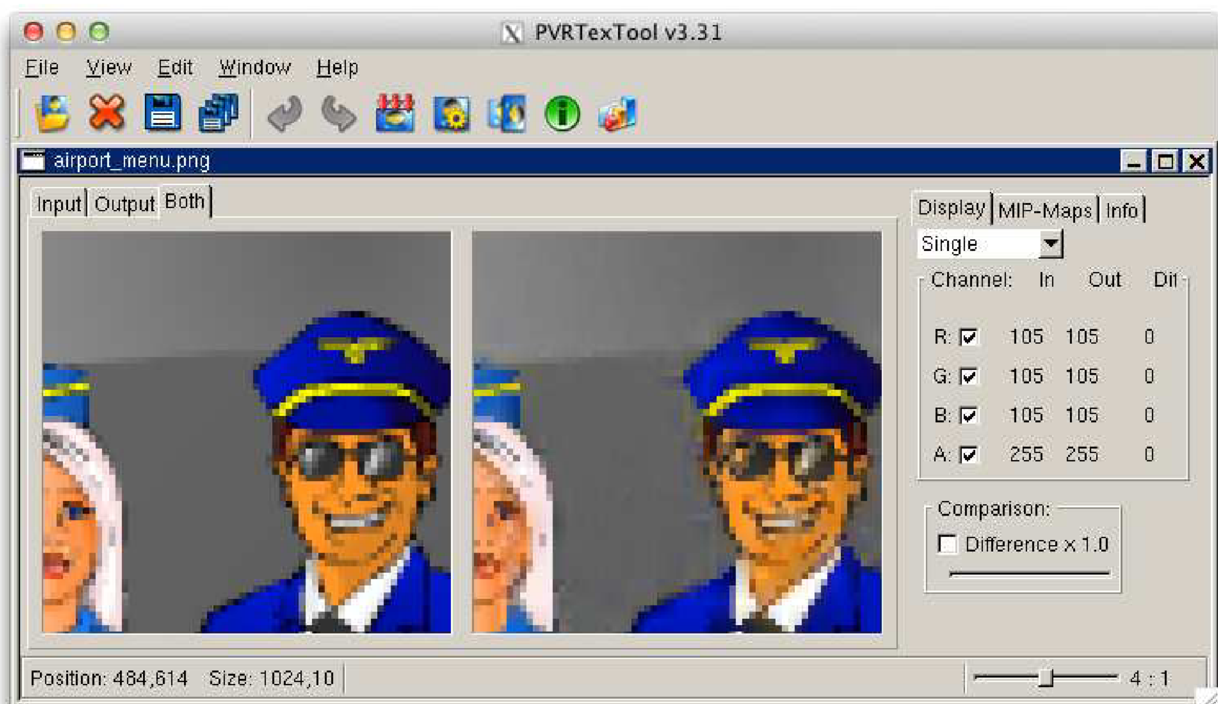
V OpenGL ES 2.0 se mezi těmito projekcemi nepřepíná, ale jednoduše se vygenerují příslušné transformační matice za pomocí knihovny GLKit, slouží k tomu dvě funkce: `GLKMatrix4MakePerspective` a `GLKMatrix4MakeOrtho`.

4.2.8 Textury

Textury jsou jednoduše řečeno obrázky uložené v paměti grafické karty, které se používají při rasterizaci polygonů (čar, bodů). Především kvůli mobilním zařízení, kde je omezené množství grafické paměti, vznikaly jiné formáty uložení a způsoby vykreslování textur.

Klasický přístup je, že textura zabírá v paměti šířka*výška*4, kde číslo 4 představuje počet bajtů na pixel, kde každé složce a alfa kanálu náleží 1B. Lze také použít 2 bajty na pixel s různým rozložením bitů pro jednotlivé barevné složky a alfa kanál, se znatelným zhoršením kvality.

Na mobilních zařízeních od Apple lze s výhodou použít formát textur PVRTC, který nejen podstatně sníží paměťovou náročnost textur, ale také znatelně zvýší rychlost vykreslování (i několikanásobně) a několikanásobně sníží rychlost načtení textury. Formát obsahuje ztrátovou kompresi a textury zůstanou ve speciálním formátu i v paměti grafické karty.



Obrázek 4.3: PVRTexTool - srovnání kvality (nalevo originál, napravo PVRTC)

PVRTC textury mohou být ve dvou formátech: 2bpp a 4bpp. Druhou verzi (4 bity na pixel), jsem bohatě využil ve své práci. Pokud mají být obrázky převedeny do tohoto formátu, musí být dodrženy následující podmínky: velikosti stran textury musí být mocniny dvou přičemž minimální délka strany je 8 pixelů a textura musí být čtvercová. PVRTC formát umožňuje i uložení mipmap.

Pro převod obrázků do formátu PVRTC slouží například program PVRTexTool, který umožňuje zvolit nejen bitovou hloubku, ale také 4 stupně kvality komprese. Zajímavé je, že stupeň kvality nijak neovlivní výslednou velikost či rychlost načítání a vykreslování, ovlivní to pouze dobu generování textury. Onen algoritmus na převod obrázku do PVRTC formátu je velmi náročný na výkon a pro texturu o velikosti 1024x1024 trvá na procesoru Core 2 Duo 2.4GHz, s nastavením 4bpp, 4. stupeň kvality (nejlepší), bez použití mipmap přes 50 vteřin.

PVRTC formátu dělají problémy táhle přechody barev a to především u nastavení nižší kvality, jsou při nich vidět barevné "schody", ostré barevné přechody (okraje tabulek a podobně) doprovází nehezké barevné artefakty, největší artefakty vznikají přitáhlém barevném přechodu kombinovaném s průhledností. Kvůli těmto nedokonalostem je dobré obrázky jemně přizpůsobit. Přes všechny tyto problémy se téměř v každém případě vyplatí PVRTC formát využívat a také se tak děje ve většině větších her.

5 Vlastní implementace

5.1 Osvětlení

Je rozlišovánomezi několika základními způsoby implementace osvětlení, klasicky je osvětlení děleno podle osvětlovacích modelů a podle typů stínování. Z běžně používaných osvětlovacích empirických modelů je potřeba zmínit Lambertův a Phongův osvětlovací model [15]. Lambertův počítá pouze s difuzním odrazem světla, s tím že se počítá s ideální difuzí (odraz se šíří konstantně do všech směrů). Intenzita difuze závisí na úhlu dopadu světla na povrch polygonu. Phongův osvětlovací model má oproti Lambertově navíc odrazovou složku, kde ostrost odrazu je definována parametrem.

Ze stínování zmíním konstantní, Gouraudovo a Phongovo. Konstantní (flat shading) je nejjednodušší a nejrychlejší, použije se pouze jedna (face) normála, každý polygon objektu má konstantní barvu, ta je vypočtena z osvětlovacího modelu. V Gouraudově stínování je barva vypočtena pro všechny tři vrcholy trojúhelníka, pro každý vrchol musí být známá normála, ta se dá vypočíst například průměrem face normál okolních polygonů. Při rasterizaci je barva jednotlivých bodů určena lineární interpolací barev vrcholů daného trojúhelníka. Toto stínování je dostatečně realistické a relativně rychlé. Phongovo stínování je z vyjmenovaných nejrealističtější, ale také nejnáročnější. Normály je také potřeba znát pro každý ze tří bodů trojúhelníka, barva se ale počítá z osvětlovacího modelu zvlášť pro každý bod rasterizované plochy.

OpenGL ES 1.x obsahovalo ve své fixní pipeline pouze osvětlení per vertex. Moje práce je psána pro OpenGL ES 2.0, tudíž mám možnost použít i per fragment osvětlení, nicméně na zařízeních iPad byla i základní implementace osvětlení per fragment (Phongovo stínování) nepřijatelně pomalá, z toho důvodu jsem se rozhodl použít pro všechna zařízení per vertex osvětlení (Gouraudovo stínování).

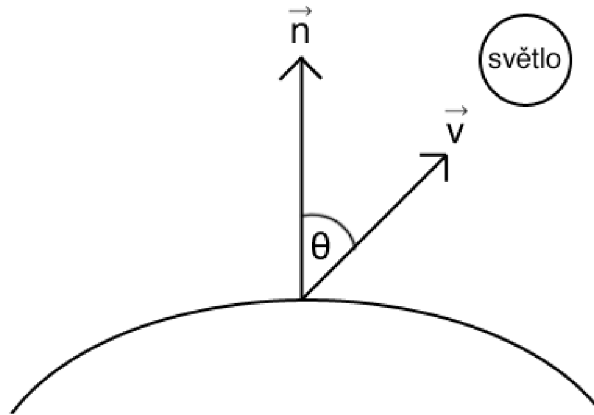
Obrázek 5.1: Per pixel + per vertex osvětlení [3]

5.1.1 Per vertex osvětlení

V mé implementaci osvětlení je využito Lambertova modelu a Gouraudova stínování [3]. Pro maximální rychlost a jednoduchost je použit pouze jeden zdroj světla, představující slunce a to je realizováno pomocí směrového světla (direction lighting). Vektor světla je stejný pro všechny fragmenty.

Lambertův osvětlovací model využívá Lambertova zákona, který říká, že intenzita osvětlení na difúzním povrhu je úměrná cosinu úhlu mezi normálou povrchu a vektorem z povrchu ke zdroji

světla. Takže jediné co je výpočtu potřeba jsou dva vektory (normála povrchu a opačný vektor světla).



Obrázek 5.2: Lambertův osvětlovací model

Rovnice pro Lambertův osvětlovací model:

$$I = \cos\theta \tag{5.1}$$

V rovnici 5.1 představuje intenzitu zdroje světla, koeficient difúzního odrazu materiálu a θ úhel, který svírá normála povrchu a vektor od vrcholu povrchu, ke světlu.

$$v_1 \cdot v_2 = |v_1||v_2| \cos\theta \tag{5.2}$$

$$I = I_p k_d (n \cdot v) \tag{5.3}$$

Pomocí úpravy v rovnici 5.2 je vytvořena rovnice 5.3 a zbavit se tak funkce cos.

V praxi je toto implementovánoza pomocí vertex shaderu například následujícím způsobem. Normála počítaného vrcholu je převedena do eye-space:

```
modelViewNormal = vec3(u_mvMatrix * vec4(a_normal, 0.0));
```

Protože používám pouze direction lighting, tak mohu s výhodou předpočítat vektor světla a dopředu jej převést také do eye-space. Provedu skalární součin připravených vektorů a výsledek omezím na hodnotu v intervalu <0.0;1.0>:

```
NdotL = max(dot(modelViewNormal, u_lightDir), 0.0);
```

Nyní vypočítám difúzní a ambientní složku světla:

```
diffuse = vec4(u_matColorDiffuse * u_lightColorDiffuse, 1.0);
ambient = vec4(u_matColorAmbient * u_lightColorAmbient, 1.0);
```

Nakonec přes varying proměnou odešlu barvu světla daného vrcholu do fragment shaderu, kde se bude provádět interpolace těchto hodnot:

```
v_light_color = NdotL * diffuse + ambient;
```

Ve fragment shaderu pak pro samostatné osvětlení stačí pouze vynásobit barvu z textury, barvou osvětlení:

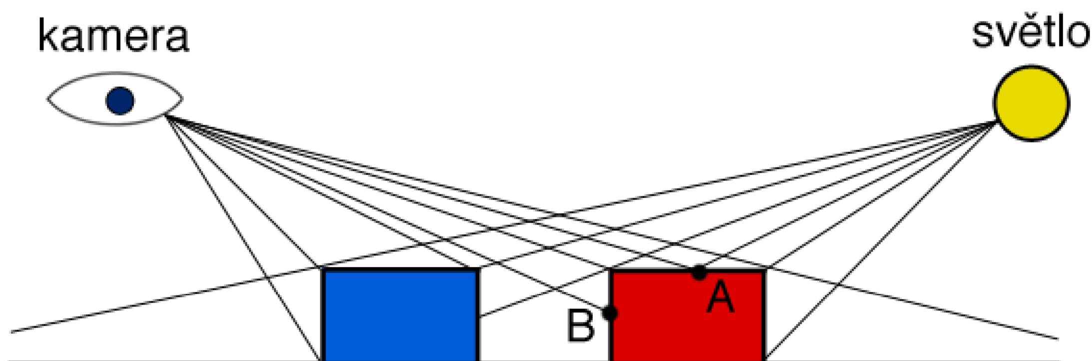
```
colorSample = texture2D(s_baseMap0, v_texCoord);
gl_FragColor = vec4(v_light_color * colorSample);
```

5.2 Shadow mapping

V mé práci jsem se rozhodl implementovat základní verzi shadow mappingu, což je v dnešní době velmi často používaný algoritmus pro generování dynamických stínů v reálném čase. Algoritmus je relativně nenáročný na výkon zařízení, rychlost jeho základní implementace závisí především na počtu světel, rozlišení shadowmapy, viewportu a rychlosti fragment shaderu zařízení. Shadow mapping generuje ostré stíny. V případě, že mají být stíny jemné, musí být provedena úprava algoritmu aplikací nějakého filtr, například PCF filtru.

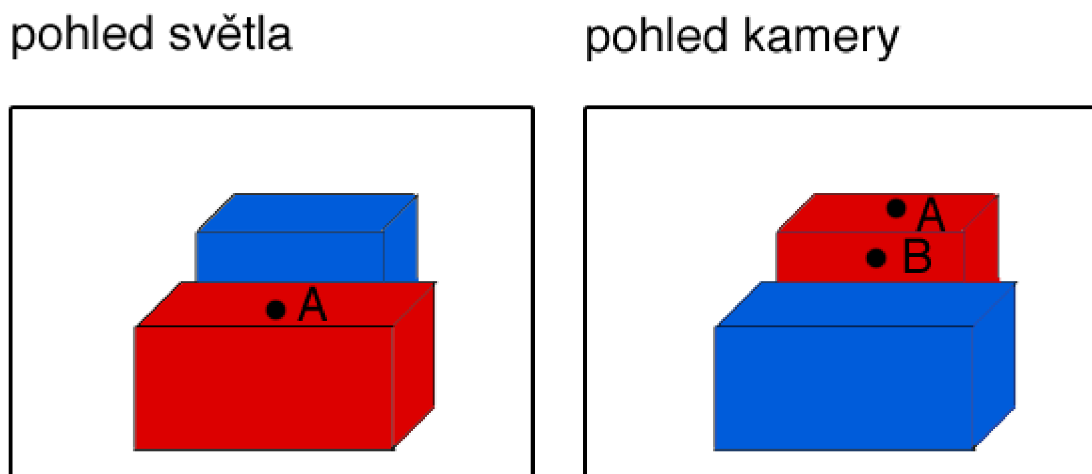
Algoritmus pracuje ve dvou průchodech. V prvním průchodu se vykreslí scéna z pohledu světla, obvykle se scéna "kreslí" do hloubkové mapy, do tzv. shadowmapy (stínové mapy). Pro základní funkci algoritmu není třeba znát barvy, stačí pouze hloubka. Je důležité zmínit, že stín je možné "vykreslit" pouze pokud je daná část objektu vykreslena do shadowmapy, to znamená že pokud určitý objekt nebude z pohledu světla vidět, pak se ve výsledku tento objekt vykreslí bez stínu i v případě, že by logicky měl být pokryt stínem.

V druhém průchodu se se scéna vykreslí klasicky z pohledu kamery, nicméně každý vykreslovaný bod je navíc převeden pomocí transformační matice světla, díky tomu je možné porovnat hloubku aktuálně vykreslovaného bodu s hloubkou uloženou v shadowmapě. Pokud je hloubka vykreslovaného bodu nižší tak je bod osvětlený, naopak pokud je hloubka vyšší, znamená to že bod leží ve stínu.



Obrázek 5.3: Shadow mapping - náčrt scény

Na obrázku 5.3 je znázorněnakamera a světlo, které jsou následně použity pro dva průchody algoritmu. Dále zde jsou dva body A a B, přičemž bod A je osvětlený a bod B je ve stínu. Pro názornost jsou na obrázcích 5.3 a 5.4 krychle vyobrazeny odlišnou barvou (modrá a červená).



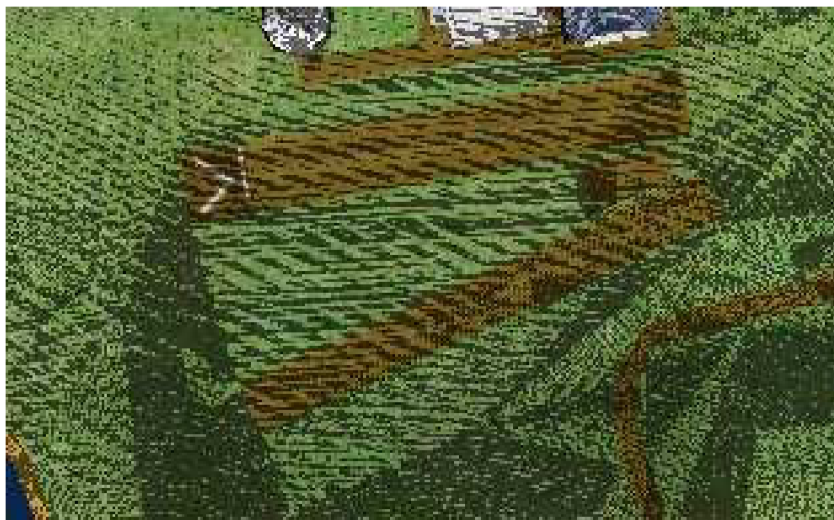
Obrázek 5.4: Shadow mapping - pohledy (nalevo pohled světla, napravo pohled kamery)

V prvním průchodu algoritmu se v případě scény jako je na obrázku 5.3 vykreslí do textury (shadowmapy) obrázek podobný obrázku 5.4 vlevo, bez vyznačených bodů A a B. Obecně by v základní implementaci algoritmu byla shadowmapa pouze v odstínech šedi, obrázek 5.4 je pro názornost barevný. Z Pohledu světla je bod B překryt stěnou, na které leží bod A, proto není vyobrazen.

Při druhém průchodu algoritmu se bude vykreslovat scéna podobným způsobem jako je znázorněno na obrázku 5.4 vpravo. Při výpočtu výsledné barvy bodu B je nutné zjistit, zda tento bod leží ve stínu nebo je osvětlen. Pokud je na bod B aplikována transformační matice, která byla použita pro světlo a z vypočítané pozice je zjištěna hloubka v shadowmapě, tak se zjistí, že bod B má větší hloubku než je vyčtená hloubka z shadowmapy, tudíž bude bod B ve stínu a jeho výslednou barvu je možno vykreslit například tmavší.

Tento algoritmus se setkává s nepříjemným problémem zvaným z-fighting. Jedná se o to, že pokud je hloubka v shadowmapě stejná jako je hloubka testovaného fragmentu scény nebo je hloubka příliš podobná, tak může kvůli přesnosti hodnot být jednou vyhodnocena jako vyšší a jednou jako nižší. Tento fakt způsobuje vizuálně velmi nepěkný artefakt, který vypadá jako problikávání a střídání stínu a světla tam, kde by měl být jen stín nebo jen světlo. Těmito artefakty trpí hlavně osvětlené plochy, protože u ploch, které jsou zastíněné, je mezi zdrojem světla a plochou kam dopadá stín nějaký objekt, tudíž pokud není tato plocha a objekt opravdu blízko k sobě, k artefaktům nedochází.

Z-fightingu se dá předejít například použitím OpenGL funkce `glPolygonOffset`, která automaticky posouvá všechny polygony do hloubky o určitou hodnotu, tudíž se volá před generováním shadowmapy nebo další možností je úprava ukládaných hloubek přímo v shaderu starajícím se o generování v shadowmapy. V mé implementaci upravuji hloubku přímo v shaderu.



Obrázek 5.4: Shadow mapping - z-fighting

Hloubkový buffer v OpenGL nemá lineární přesnost, čím blíže je vykreslovaný objekt ke kameře, tím je testování hloubky přesnější. Důležitý je fakt, že ořezávací roviny zNear a zFar by měli být co nejblíže sobě, aby se co nejvíce zpřesnil test na hloubku. V shadow mappinguobecně platí, čím vyšší rozlišení shadowmapy, tím kvalitnější výsledný stín. Pozice kamery by měla být nastavena tak, aby docházelo k co největšímu využití plochy shadowmapy, v případě směrového světla lze jednoduše posunout kameru po vektoru světla.

5.2.1 Generování shadowmapy

Vytvoření shadowmapy je jak jsme si pověděli první průchod algoritmem. Ve starších verzích systému iOS (<4.0) OpenGL ES postrádalo rozšíření `GL_OES_depth_texture` a tím pádemnebylo možné připojit k framebufferu texturu jako hloubkový buffer. Přestože tato skutečnost značně ztěžuje celou implementaci, protože se zapisování hloubky musí udělat jiným způsobem, rozhodl jsem se pro tuto těžší cestu a kompatibilitu se staršími systémy.Problém se dá elegantně vyřešit uložením hloubky do běžné textury(color bufferu) [14]. Jeden pixel v RGBA8 textuře spotřebuje pro svoji barvu 32bitů, takže místo RGBA je možné chytrým způsobem uložit 32bitový float. Pomocí následující metody lze uložit float hodnoty 0 až 1.

Ve vertex shaderu pouze vypočítáme pozice vrcholů vykreslovaného objektu:

```
gl_Position = u_mvpMatrix * vec4(a_vertex, 1.0);
```

Veškeré výpočty a celé uložení pak probíhá ve fragment shaderu. Nyní se naskýtá možnost hloubku uměle upravit tak, aby docházelo co nejméně k artefaktům při testování hloubky v druhém průchodu algoritmu:

```
highp float normalizedDistance  = v_position.z / v_position.w;
```

```
normalizedDistance += 0.0002;
```

Poté je třeba hloubku normalizovat do intervalu <0.0;1.0>:

```
normalizedDistance = (normalizedDistance + 1.0) / 2.0;
```

Hloubkaje nyní postupně vynásobena hodnotami 2^{24}, 2^{16}, 2^{8} a 1. Z výsledných čísel je vzata pouze desetinná část:

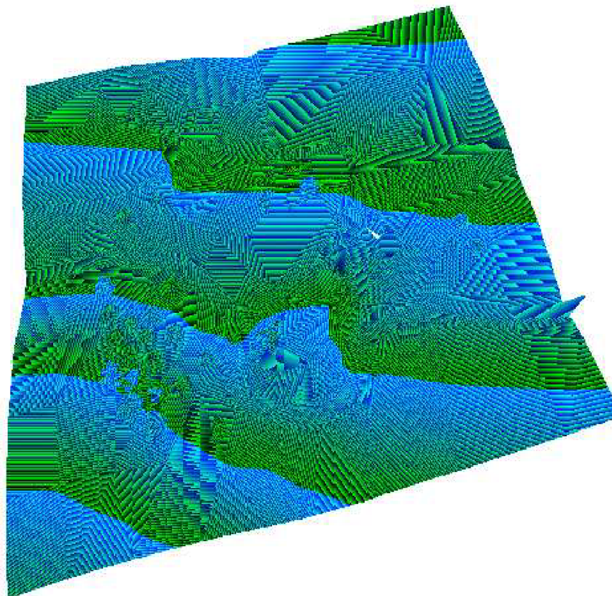
```
const highp vec4 packFactors = vec4(256.0*256.0*256.0,256.0*256.0,
                                  256.0, 1.0)
highp vec4 packedValue = vec4(fract(packFactors*
                                  normalizedDistance));
```

Následující kód slouží ke zvýšení přesnosti čísel ve výsledném vektoru:

```
const highp vec4 bitMask = vec4(0.0, 1.0/256.0, 1.0/256.0,
                              1.0/256.0);
packedValue -= packedValue.xxyz * bitMask;
```

Zbývá už jen předat výslednou "barvu" výstupní proměnné:

```
gl_FragColor = packedValue;
```



Obrázek 5.5: Shadowmapa (implementace přes color buffer)

Díky tomu, že shadowmapa je uložená do klasické RGBA textury, tak netvoří pouze odstíny šedi, ale pokud ji vykreslena, vypadá jako na obrázku 5.5.

Protože shadowmapa sémanticky obsahuje pouze hloubku, není potřeba při renderování brát v úvahu textury, barvy a osvětlení. Dbá se na to, aby vygenerování shadowmapy bylo co nejrychlejší.

5.2.2 Aplikace shadowmapy

Shadowmapu máme vygenerovanou, nyní je na čase ji v druhém průchodu aplikovat během ostrého vykreslování scény. V mé práci je shadow mapping použit ve dvou shader programech, pro osvětlení a pro osvětlení+voda. K shader proměnným ve vertex shaderu přibudou pouze dvě proměnné, transformační matice světla, která byla použita při generování shadowmapy v prvním průchodu a varying proměnná přes kterou se do fragment shaderu bude posílat aktuální vrchol s aplikovanou maticí pro světlo. Ve fragment shaderu přibude zmiňovaná varying proměnná, sampler s shadowmapou a uniform proměnná určující intenzitu stínu.

Výpočet stínů je součástí shaderu pro dynamické osvětlení, následující kód shaderů bude obsahovat pouze části týkající se přímo aplikace stínů.

Ve vertex shaderu proběhne výpočet pozice bodu z pohledu světla:

```
v_lightPOVPosition = u_lightPOVPVMMatrix * vec4(a_vertex, 1.0);
```

Většina výpočtů bude probíhat ve fragment shaderu, to je také důvod proč náročnost shadowmappingu stoupá se zvyšujícím se rozlišením. Následující proměnná fragment shaderu bude určovat míru stínu na fragmentu. Hodnota 0.0 znamená maximální stín (tma) a hodnota 1.0 znamená žádný stín (plné osvětlení):

```
lowp float shadowFactor = 1.0;
```

Je provedena normalizace bodu z pozice světla do intervalu <0.0;1.0>:

```
highp vec4 lightZ = v_lightPOVPosition / v_lightPOVPosition.w;
lightZ = (lightZ + 1.0) / 2.0;
```

Vlastní funkcí `getShadowFactor` je zjištěno, zda je fragment ve stínu nebo ne:

```
shadowFactor = getShadowFactor(lightZ);
```

Nakonec je aplikován vypočtený stín na barvu fragmentu:

```
gl_FragColor = vec4(v_light_color.rgb * colorSample.rgb *
                 shadowFactor, colorSample.a * u_alpha);
```

Ve funkci `getShadowFactor` je vyčtena příslušná hodnota z shadowmapy a voláním vlastní funkce `unpack` je provedeno vyčtení float hodnoty (představující hloubku), která byla do shadowmapy zabalena v prvním průchodu:

```
lowp float getShadowFactor(highp vec4 lightZ)
{
    highp vec4 packedZValue = texture2D(s_shadowMap,lightZ.st);
    highp float unpackedZValue = unpack(packedZValue);
```

Rozbalená hodnotaje následně použita pro porovnání hloubky, v případě že je hloubka ze shadowmapy větší, znamená to že je vykreslovaný fragment osvětlený. V opačném případě se vrací hodnota nižší než 1.0. Já jsem zvolil hodnotu začínající na 0.4, díky tomu nebude nikdy stín úplně černý, ale fragment pouze oněco ztmavne:

```
    if (unpackedZValue> lightZ.z)
        return 1.0;
    else
    {
        mediump float shadowIntenzity = 0.4 + 0.6* (1.0 -
                                                u_shadowIntenzity);
        return shadowIntenzity;
    }
}
```

Ve vlastní shader funkci `unpack` se rozbalí hodnoty barev opačným způsobem, jako se hloubka zabalila v prvním průchodu. Jednotlivé komponenty barvy se vynásobí čísly $1/2^{24}$, $1/2^{16}$, $1/2^{8}$, 1 a následně se sečtou:

```
highp float unpack(highp vec4 packedZValue)
{
    highp vec4 unpackFactors = vec4(1.0/(256.0*256.0*256.0),
                        1.0 / (256.0*256.0), 1.0 / 256.0, 1.0);
    return dot(packedZValue, unpackFactors);
}
```

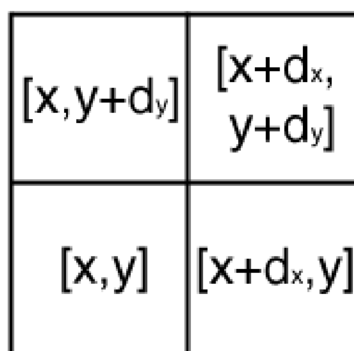
Obrázek 5.6: Shadow mapping (vlevo vypnuto, vpravo zapnuto)

5.2.3 PCF filtr

Shadow mapping je metoda dynamických stínů, která tvoří ostré stíny. Pokud by přímo na shadowmapu bylo aplikováno klasické vyhlazování, došlo by akorát ke zprůměrování hloubek uložených v shadowmapě, což by způsobilo jen obrovské množství artefaktů. Pro zjemnění okrajů

stínů se běžně aplikuje tzv. PCF filtr (percentage closer filter). Tato metoda spočívá v porovnávání hloubky vykreslovaného fragmentu s hloubkami několika nejbližších pixelů k příslušné pozici v shadowmapě. Porovnání hloubky se provádí pro každou vyčtenou hodnotu z shadowmapy zvlášť, počet úspěšných testů se pak sčítá a výsledek se vydělí počtem testů. Vypočtené číslo pak určuje intenzitu stínu.

Čím větší matice pixelů je pro filtr použita tím hladší jsou ve výsledku stíny, ale zároveň tím větší dopad na výkon tato metoda má. Pokud v OpenGL probíhá rasterizace polygonu a texturovací souřadnice nejsou ve fragmentshaderu uměle upravovány, tak GPU provádí přednačítání bodů textury a díky tomu je rasterizace rychlejší. V případě PCF, ale vyčítáme z textury (z shadowmapy) pixely i z okolních souřadnic, tudíž nemůže efektivně probíhat přednačítání a celkově se proces rasterizace o trochu zpomalí.



Obrázek 5.7: PCF - matice

Do mojí implementace PCF filtru jsem použil pouze matici o velikosti 2x2, z důvodu relativně vysoké výkonové náročnosti. Na obrázku 5.7 je vidět použitá PCF matice, okolní pixely se berou zprava a zvrchu. Hodnoty d_x a d_y jsou hodnoty posunutí texturovacích souřadnic a jsou vypočítány zvlášt pro zařízení iPhone a iPad jednoduchými vzorci:

$$d_x = \frac{1}{resolutionX * shadowMapRation} \tag{5.4}$$

$$d_y = \frac{1}{resolution * shadowMapRation} \tag{5.5}$$

Hodnoty $resolutionX$ a $resolution$z rovnic 5.4 a 5.5 představují rozlišení, ve kterém se renderuje scéna a $shadowMapRation$ je poměr velikosti shadowmapy oproti rozlišení používanému při renderování.

Aplikace PCF filtru spočívá pouze v úpravě již známé shader funkce `getShadowFactor`:

```
highp vec4 packedZValue = texture2D(s_shadowMap, lightZ.st);
highp float unpackedZValue = unpack(packedZValue);
```

```
mediump int suc = 0;
```

Nově se vyčítají a porovnávají 3 okolní hodnoty (hloubky) shadowmapy s hloubkou vykreslovaného fragmentu. Podle použitého zařízení se shader přeloží buď s větví pro iPad nebo pro iPhone, tyto větve se liší hodnotami posunutí vypočtenými podle vzorců 5.4 a 5.5:

```
#ifdef DEVICE_IPAD
    highp vec4 packedZValue2 = texture2D(s_shadowMap, lightZ.st
                                   + vec2(0.00048828125, 0.0));
    highp vec4 packedZValue3 = texture2D(s_shadowMap, lightZ.st
                                + vec2(0.0, 0.00065104166667));
    highp vec4 packedZValue4 = texture2D(s_shadowMap, lightZ.st
                      + vec2(0.00048828125, 0.00065104166667));
#else
    highp vec4 packedZValue2 = texture2D(s_shadowMap, lightZ.st
                                + vec2(0.00083333333333, 0.0));
    highp vec4 packedZValue3 = texture2D(s_shadowMap, lightZ.st
                                      + vec2(0.0, 0.00125));
    highp vec4 packedZValue4 = texture2D(s_shadowMap, lightZ.st
                            + vec2(0.00083333333333, 0.00125));
#endif

highp float unpackedZValue2 = unpack(packedZValue2);
highp float unpackedZValue3 = unpack(packedZValue3);
highp float unpackedZValue4 = unpack(packedZValue4);

if (unpackedZValue > lightZ.z)
    suc++;
if (unpackedZValue2 > lightZ.z)
    suc++;
if (unpackedZValue3 > lightZ.z)
    suc++;
if (unpackedZValue4 > lightZ.z)
    suc++;
```

Počet úspěšných testů hloubky je vydělen číslem 4.0, tím je mezivýsledek převeden do intervalu <0.0;1.0>, k tomu je fixně připočtena hodnota 0.4 a hodnota intenzity stínu. Výsledek je oříznut opět do intervalu <0.0;1.0>:

```
lowp float percentage = float(suc) / 4.0;
percentage += 0.4 + 0.6 * (1.0 - u_shadowIntenzity);
return clamp(percentage, 0.0, 1.0);
```

Obrázek 5.8: PCF - porovnání (nalevo vypnuto, napravo zapnuto)

5.2.4 Náročnost na výkon

V následující tabulce 5.1 jsou uvedeny rozdíly v jednotkách fps (frames per second) na jednotlivých zařízeních se třemi možnostmi nastavení: vypnuté stíny, zapnuté stíny, zapnuté stíny + PCF. Pro co nejpřesnější výsledky testování probíhalo hned po spuštění hry bez zobrazených letadel. Horní hranice fps je 60, což je obnovovací frekvence obrazovek zařízení.

	stíny vypnuto	stíny zapnuto	stíny zapnuto + PCF
iPhone 3GS	60+	28	19
iPhone 4	60+	39	22
iPad 1	43	8 - 9	4 - 5
iPad 2	60+	48	28
iPad (3rd gen)	60+	60+	51

Tabulka 5.1: Výkonová náročnost implementace dynamických stínů a PCF na jednotlivých zařízeních

Po zobrazení několika letadel, křivek a efektů teplého vzduchu se logicky fps ještě o nějakou část sníží. Z tabulky 5.1 lze vyčíst, že použití dynamických stínů na starším iPadu 1 je vyloučené. PCF filtrování stínů má vyhovující fps pouze na novějších zařízeních iPhone 4S a iPad (3rd gen).

5.3 Postprocessing

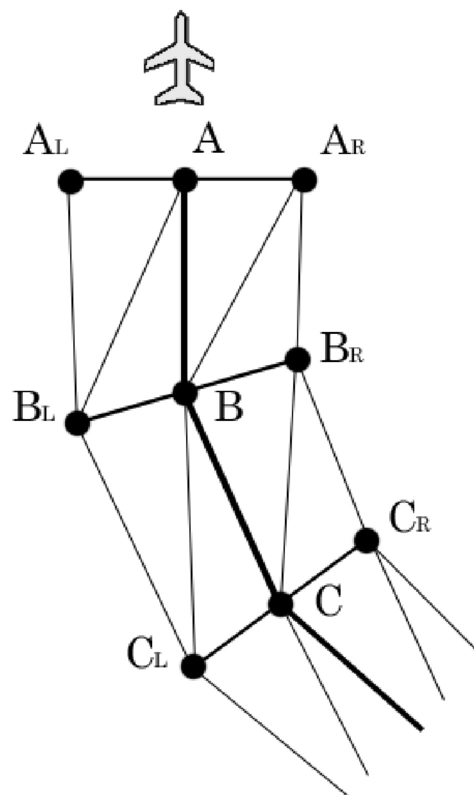
Jedním z cílů této práce byla implementace postprocessingu, konkrétně efektu vlnění horkého vzduchu za tryskovými letadly.Postprocessing je úprava již vyrenderované scény, jedná se tedy vlastně o úpravu 2D obrázku.

Implementace nějakého základního postprocessing efektu v OpenGL ES 2.0 je poměrně jednoduchá. Scénu stačí vyrenderovat do textury, nejrychlejší způsob jak to udělat je přes FBO (frame buffer object), který je propojen z texturou. Tuto texturu následně vykreslit do obdélníka pokrývajícího celou obrazovku a na tento obdélník aplikovat shader s postprocessing efektem. Téměř celý kód postprocessingu je umístěn ve fragment shaderu, tudíž se výpočet provádí pro každý pixel. Náročné efekty tedy mohou spolknout nezanedbatelnou část celkového fps.

5.3.1 Efekt horkého vzduchu

V prvních pokusech o implementaci tohoto efektu jsem se pokoušel o fullscreen postprocessing s předáváním pole informací o trajektorii na které se efekt měl aplikovat, ale setkal jsem se s neúspěchem, tento postup byl velmi pomalý s neuspokojivým vizuálním výstupem.

V Dalších pokusech jsem se rozhodl aplikovat postprocessing shader pouze na určitou část obrazovky, konkrétně přesně na plochu, na které měl být efekt zobrazen, tento postup byl nakonec tím správným. Pro použití tohoto postupu je nutné každý snímek vypočítat seznam polygonů, které dohromady tvoří plochu (heat plochu), na kterou se bude efekt aplikovat.



Obrázek 5.9: Heat plocha - náčrt

Sada úseček tvořená body A, B, C z obrázku 5.9 představuje trajektorii, kterou letadlo uletělo, stím že bod A je poslední aktuální bod letadla. Při každém vykreslovacím cyklu se ukládá aktuální pozice letadla v případě, že je dodržena určitá minimální vzdálenost od minulého bodu, tím se postupně tvoří historie trajektorie letadla. Následně se prochází celá tato historie pozic a z ní se

vytvoří trajektorie nová, ve které mají všechny pozice konstantní vzdálenost. Body A, B, C jsou již přepočítané s konstantními rozestupy.

Boční body (s indexy L-left a R-right) se vypočítají třemi následujícími způsoby, pro první bod se spočte směrový vektor, který je tvořen prvním bodem a bodem následujícím (druhým), k tomuto vektoru je vytvořen vektor kolmý, ten je roztažen na určitou velikost. Tím je získán jeden z bočních bodů, pokud je kolmý vektor otočen, vznikne druhý boční bod. Boční body pro poslední bod trajektorie se spočítají obdobně stím rozdílem, že směrový vektor je tvořen posledním a předposledním bodem trajektorie. Všechny ostatní boční body budou mít směrový vektor vypočtený z předchozího a z následujícího bodu trajektorie, výpočet kolmého vektoru a samotných bočních bodů je pak stejný jako v předchozích dvou možnostech.

Po získání všech potřebných bodů se začne skládat výsledné pole polygonů. V současném stavu by přechod mezi oblastí, na kterou bude postprocessing aplikován a oblastí bez efektu byl velmi ostrý a vizuálně nepěkný. Proto je třeba zavést tzv. attenuation (postupný útlum), každému z bodů je přidána hodnota v intervalu <0.0;1.0> určující intenzitu aplikovaného efektu při postprocessingu. Všechny boční body mají vždy attenuation 0.0, první a posledníbod má attenuation také nulovou. Zbývá vypočítat hodnoty pro další hlavní body trajektorie (body B,C na obrázku 5.9). Každý bod trajektorie má u sebe uloženou dobu vytvoření, díky této hodnotě lze stanovit attenuation podle času, v mé práci jsem stanovil dobu náběhu do plně intenzity na 0.7 sec., dobu začátku ztrácení intenzity na 3.5 sec. dobu snižování na 1.4 sec. Znamená to, že nově vloženému bodu bude trvat 0.7 vteřin než se jeho hodnota attenuation dostane na 1.0 a ten samý bod začne v čase 3.5 sec. od vytvoření ztrácet na intenzitě efektu. Tím, že jsou časy stanoveny fixně, závisí výsledná délka trajektorie, na kterou se efekt bude aplikovat pouze na rychlosti letadla.

Texturovací souřadnice jednotlivých bodů se vypočtou velice snadno, protože pozice každého z vrcholů odpovídá přímo pozici na obrazovce:

$$texCoord_x = \frac{vertex_x}{resolutionX} \tag{5.6}$$

$$texCoord_y = \frac{vertex}{resolutionY} \tag{5.7}$$

Pokud by tento seznam polygonů tvořící plochu pro aplikaci heat efektubyl vykreslen jako wireframe (síťový model), bylo by vidět jak se snímek po snímku mění vzájemné pozice všech vrcholů a vypočtené pozice vrcholů by plynule obtékali původní trajektorii.

Aplikace heat efektu probíhá následujícím způsobem: scéna, na kterou se efekt bude aplikovat se pomocí FBO vyrenderuje do textury, tato textura se vykreslí do hlavního framebufferu (na obrazovku) jako klasický obrázek bez zvláštních efektů, následně se pro každé tryskové letadlo zvlášť vykreslí vypočtené heat plochy za použití postprocessing shaderu a textury s vyrenderovanou scénou. Tato textura se nebude nijak měnit, protože efekt se vykresluje také přímo do hlavního framebufferu (na obrazovku). Tento postup má jeden problém, který se vizuální projeví jen málo, pokud jsou přes sebe vykresleny dvě heat plochy s efektem vznikne ostrý přechod intenzity efektu. Tím, že je efekt aplikován na malou plochu a efekt vlnění vzduchu obsahuje pouze malé vlnky, tak se tato nedokonalost ztratí.

Ve vertex shaderu se oproti běžnému výpočtu `gl_Position` předávají varying proměnné s texturovacími souřadnicemi a již zmiňovaným attenuation.

```
v_texCoord = a_texCoord;
v_attenuation = a_attenuation;
```

Všechny náročné výpočty pak probíhají ve fragment shaderu hodnota attenuation je upravena tak, aby přechod z nulové intenzity do maximální nebyl lineární a ve výsledku vypadal co nejlépe pomocí funkce `sin`. Křivka sinus je posunuta o -90°, posunuta nad osu x a normalizována do intervalu <0.0;1.0>, tím vznikne pozvolný náběh z hodnoty nula a pozvolný doběh do hodnoty 1.0. Výsledná hodnota je vynásobena hodnotou uniform proměnné `u_strength` představující dodatečnou úpravu síly efektu:

```
const mediump float PI = 3.1415926535;
const mediump float PIdiv2 = 1.57079632675;
mediump float attenuation = ((sin(v_attenuation * PI - PIdiv2)
                                       + 1.0) / 2.0) * u_strength;
```

Hlavní část efektu je jednoduchá a využívá pouze funkce `sin`. Princip efektu spočívá v posouvání texturovacích souřadnic rasterizovaného polygonu, pomocí uniform proměnné `u_time` se celý efekt animuje:

```
mediump vec2 uv;
uv.x = v_texCoord.x + sin(v_texCoord.y * 4.0 * 2.0 * PI * 20.0
                               + u_time*8.0) * 0.005 * attenuation;
uv.y = v_texCoord.y + sin(v_texCoord.x * 4.0 * 2.0 * PI * 20.0
                               + u_time*5.0) * 0.005 * attenuation;
gl_FragColor = vec4(texture2D(s_baseMap0, uv).xyz, 1.0);
```

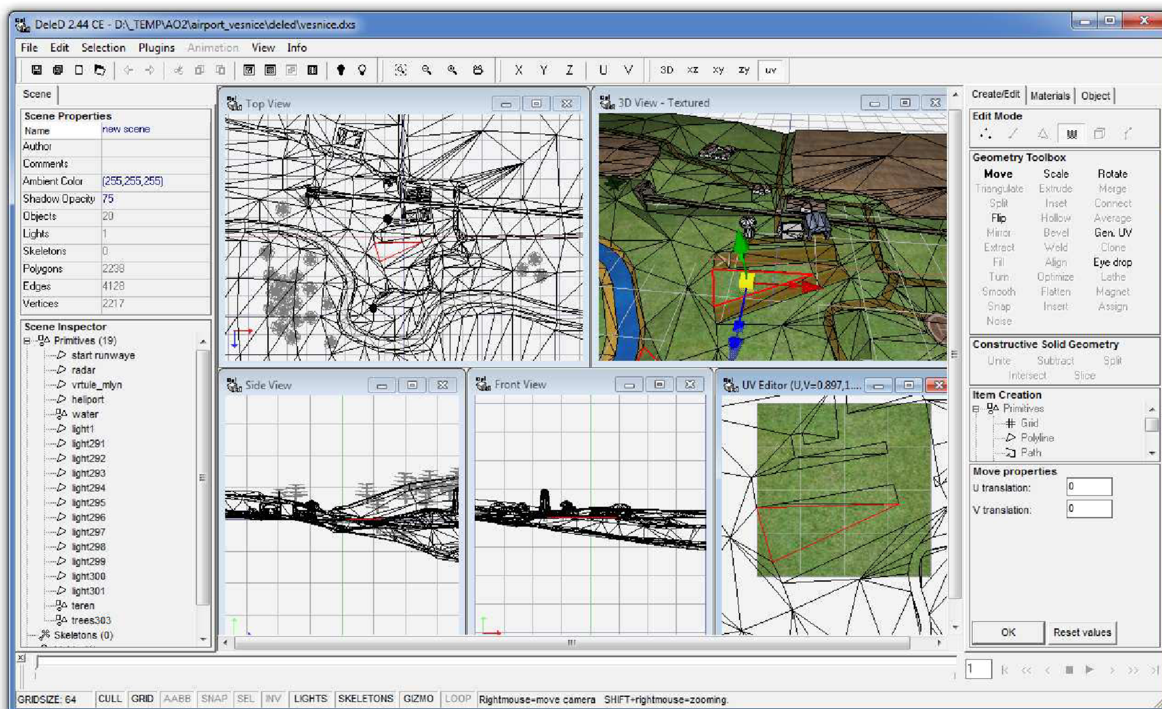
Obrázek 5.10: Postprocessing – heat efekt (vlevo vypnuto, vpravo zapnuto)

5.4 Modely

Modely jsou v paměti reprezentovány tak, aby je bylo možno co nejrychleji vykreslovat. Každý model má v sobě uložen seznam materiálů, které mohou být použity pro vykreslování a seznam objektů v rámci modelu. Materiál obsahuje kromě obecných informací především id textury, případně barvu. Objekty v modelu obsahují pole prokládaných vertexů (zasebou položená data: vertex, normal, texcoord), z tohoto pole vytvořené VBO (vertex buffer object) a pole se změnami materiálů. Každé tři vrcholy tvoří jeden polygon (trojúhelník), tyto trojice jsou seřazeny tak, aby docházelo co nejméně ke změnám materiálu a tím pádem aby se funkce `glDrawArrays` volala co nejméně.

5.4.1 Editor

Modely jsou vytvořeny v modelovacím nástroji DeleD CE. Tento software je zdarma a je vhodný především k modelování nízkopolygonových modelů. DeleD je jednoduchý editor, který stačí na jednodušší modely, obsahuje nástroje pro práci s geometrií, CSG, základní animační systém, UV mapping, lightmapping a lze použít i raytracing na tvorbu obrázků. DeleD je pouze pro systém Windows a v současné chvíli je vyvíjen pouze skupinou nadšenců.



Obrázek 5.11:DeleD CE

V pozdějších fázích projektu se ukázala exkluzivita pro systém Windows jako značná komplikace a celkově se volba tohoto nástroje ukázala jako nepříliš šťastná, kvůli přílišné jednoduchosti a nízké stabilitě.

5.4.2 Konverze do vlastního formátu

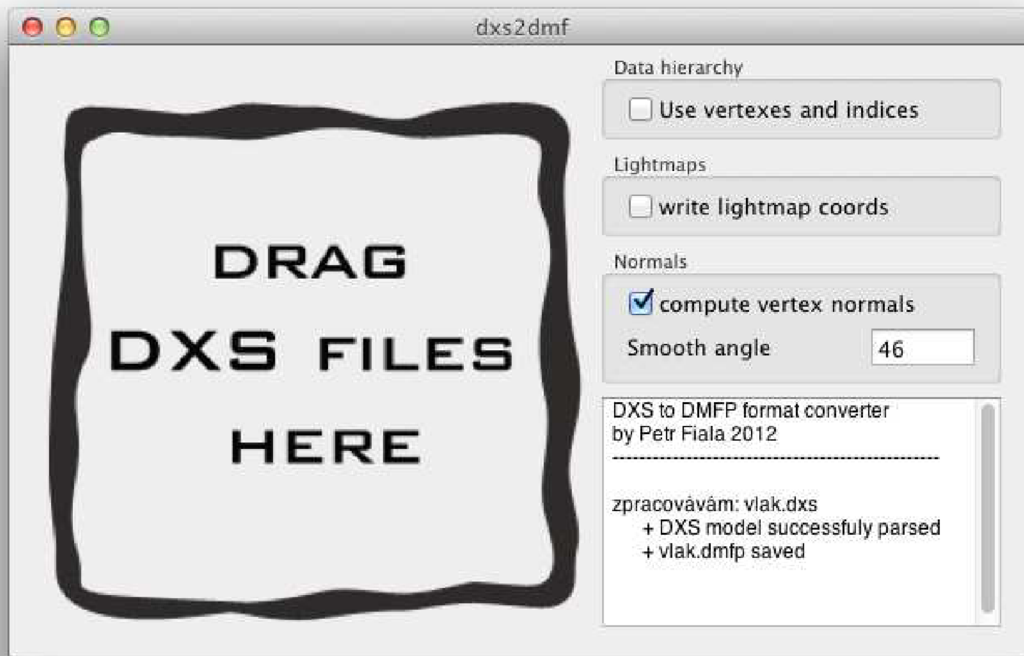
DeleD ukládá modely do formátu XML, který je sice dobře čitelný, ale není z důvodu rychlosti vhodný pro přímé použití v aplikaci. Z tohoto důvodu jsem vytvořil doplňující aplikaci, která soubory modelů z DeleD umí převést do vlastního optimalizovaného formátu. Aplikace umí vypočítat normály pro každý vrchol modelu, s tím že maximální úhel, který mohou polygony svírat u jednoho vrcholu jde nastavit před začátkem konverze. Při převodu se provádí optimalizace pořadí polygonů tak, aby se musely textury, případně barvy měnit co nejméně.

Proces konverze probíhá následujícím způsobem: polygony a vertexy načtou do paměti prakticky 1:1, pak se ke každému polygonu vypočítá face normála a ke každému vertexu se uloží náležitosti k polygonům (indexy polygonů, které vertex obsahují). Pole polygonů se postupně prochází a dopočítávají se vertex normály průměrováním face normál okolních polygonů, okolní normála je použita pouze pokud je dodržena podmínka maximálního úhlu mezi normálami. Pokud je úhel větší než hodnota zadaná do GUI před začátkem konverze, tak se vytvoří kopie daného vertexu a ten se pak vloží do výsledného pole jako vertex nový a z původního vertexu se odstraní náležitost k danému polygonu.

Můj formát modelů (dmfp) má následující strukturu:

- textová informace o formátu
- verze
- použití lightmap koordinátů (0-ne / 1-ano)
- hierarchie dat (1-velke prokládané pole / 2-pole vertexu a pole polygonu)
- počet materiálů
- seznam materiálů (id; jméno materiálu; kategorie; 0; počet vrstev; typ; jméno souboru textury; 1;)
- počet objektů
- seznam objektů
  - hlavička (jméno objektu; pořadové číslo, tag)
  - počet prokládaných vrcholů
  - seznam prokládaných vrcholů (vertexX; vertexY; vertexZ; normalaX; normalaY; normalaZ; texcoordU; texcoordV;)
  - materiály ke každé trojici vrcholů (id; id; id;...)
- počet tagů
- seznam tagů (X_0; Y_0; Z_0; X_1; Y_1; Z_1; X_2; Y_2; Z_2; text tagu)

Formát se velmi rychle načítá, protože se nemusí nic dopočítávat, pouze se vyplňují připravené struktury. Tagy objektů slouží k určení, který shader se má pro renderování použít (například voda, stromy, terén), seznam tagů na konci obsahuje pro každý záznam tři vrcholy tvořící trojúhelník slouží k pozicování objektů ve scéně, například pozice vrtule mlýna, pozice radaru, přistávací světla a další.



Obrázek 5.12: Pomocná aplikace – dxs2dmfp

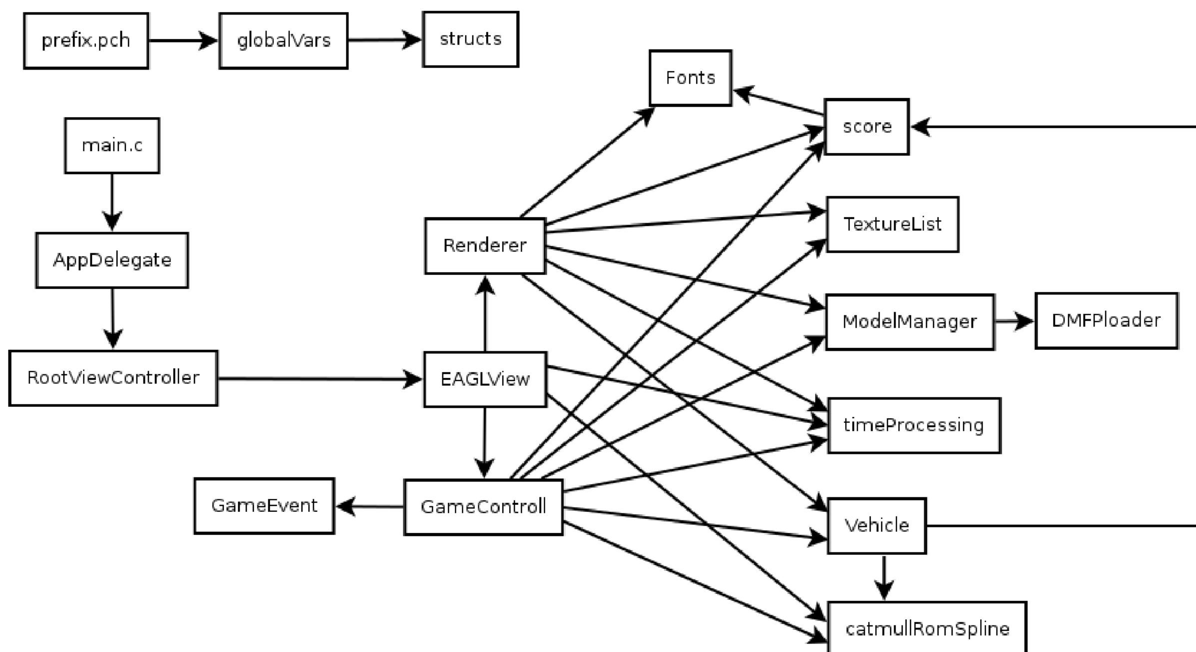
5.4.3 Renderování modelů

K vykreslování modelů slouží pouze jedna funkce (`drawModel: withShadows: alpha: shadowGenerator: parameters:`), která při nastavení parametru `shadowGenerator` umí generovat shadowmapu a při nastavení `withShadows` umí shadowmapu aplikovat na ostrou scénu. Ve funkci se defaultně nastavuje vypnutá průhlednost, ta se zapíná jen pokud je v `parameters` uložena hodnota `"forceBlend"` nebo vykreslovaný objekt má tag hodnotu `"blend"`.

Probíhá cyklus přes všechny objekty modelu a před začátkem vykreslování daného objektu se podle tagu objektu nastaví příslušný shader (voda, terén). Podle parametru `withShadows` se použije buď shader bez použití shadow mappingu nebo s ním. Jak jsme si již řekli, každý objekt modelu má v sobě uložené pole změn materiálů. Probíhá vnořený cyklus přes toto pole a před každým kreslením se nastaví správný materiál.

5.5 Struktura aplikace

Po spuštění se stejně jako v klasických c/c++ aplikacích volá funkce `main`. Soubor s touto funkcí je zpravidla předgenerovaný a není třeba jej nijak měnit. Následně uvedu zjednodušené schéma závislostí modulů aplikace:



Obrázek 5.13: Zjednodušený diagram závislostí modulů

V iOS projektech, které obsahují soubor prefix.pch, je tento soubor automaticky vkládán na začátek všech zdrojových souborů. V prefix.pch je vložen hlavičkový soubor s globálními proměnnými, díky tomu jej není nutné nikam ručně vkládat.

Po zavedení aplikace je v `AppDelegate` automaticky zavolána metoda `application:didFinishLaunchingWithOptions:` [2], ve které je vytvořeno hlavní okno aplikace a do něj je umístěn `RootViewController`. Příchozí události typu otočení zařízení jsou předávány hlavnímu oknu a to předává událost dál do svého prvního viewControlleru.

V `RootViewController` je provedena alokace `EAGLView` (OpenGL view). V `EAGLView` je provedena inicializace OpenGL a probíhá zde hlavní běhová smyčka, v které se pořád dokola opakuje kompilace shaderů (pokud je to potřeba), správa času - `timeProcessing`, herní výpočty - `GameControll` a renderování - `Renderer`.

5.6 Herní systém

Vnitřní systém hry obsahuje dva stavy, `MODE_MENU` a `MODE_GAME`. Výpočty probíhají podle toho v jakém stavu se aplikace aktuálně nachází. V třídě `EAGLView` jsou metody pro zachytávání veškerých dotyků obrazovky [2]. V závislosti na stavu hry se testují různé dotyky. Po kliknutí na "Hrát" v menu aplikace, se volá metoda `newRound`, která provede veškerou inicializaci hry, načtení modelů a textur, inicializace systému eventů, vytvoření křivek určených k přistávání a vzlétání, vyčištění seznamu aktivních letadel a obecně vynulování všech potřebných proměnných.

5.6.1 Pomocné třídy

Pro usnadnění práce jsou zavedeny třídy `Button` a `GUIitem`. Třída `Button` obsahuje metody pro vykreslení tlačítka, testování zda zadaný bod leží v tlačítku (test zmáčknutí) a změnu nastavení tlačítka. `GUIitem` slouží k postupnému mizení a zobrazení vykreslovaných prvků. Typický příklad je přepínání obrazovek "Nastavení" a "Skóre" v menu, které se postupně prolínají nebo prolínání menu, loading obrazovky a samotné hry.

Pro správu textur slouží třída `TextureList`, která zařídí, že jedna textura se nenačte vícekrát, vnitřně si vytváří slovník s dvojicemi jméno souboru – textura. Při zavolání načítací metody se nejprve zkontroluje zda textura není ve slovníku, pokud ano, vrátí se již dříve načtená textura.

O vykreslování textů se stará třída `Fonts`. Systém fontů je navržen tak, že pro každý font určité velikosti musí být vytvořena instance třídy `Fonts` a až při prvním vykreslení nějakého textu daným fontem se pro každé písmeno textu začnou generovat malé textury přibližně o velikosti fontu. Obrázky písmen jsou generovány za pomocí grafických funkcí knihovny CoreGraphics a následně jsou z nich v OpenGL vytvořeny textury. Každé písmeno má určitou šířku a podle ní se při vykreslování textu určují rozestupy písmen.

5.6.2 Systém eventů

Ve hře je zaveden systém eventů, ve kterém se do pole ukládají objekty typu `GameEvent`. Tyto objekty obsahují čas spuštění eventu a samozřejmě informace co se má stát. V každém běhu herní smyčky jsou otestovány všechny eventy z pole eventů a ty, u kterých čas spuštění překročil aktuální čas se provedou a odstraní z pole. Ve hře je rozlišeno několik typů eventů: generování letadla, zobrazení varování na místě příletu letadla, smazání letadla, změna intervalu generování letadel a příjezd vlaku.

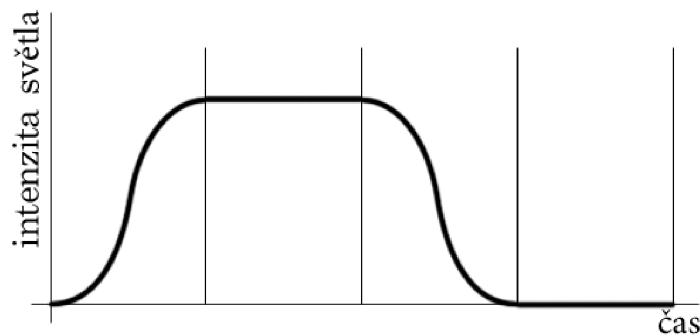
5.6.3 Generování letadel

Herní systém je nastaven tak, že po uplynutí určité doby (intervalu), která jde nastavovat pomocí eventů, se vytvoří dva nové eventy. Nejprve event upozornění na příchozí stroj (letadlo, helikoptéra) a následně samotný event vygenerování stroje. Letadla helikoptéry se tedy generují v případě, že přijde příslušný event typu "vygeneruj novy stroj". Náhodně se vybírá zda bude letadlo startovat z letiště nebo jestli přiletí z kraje obrazovky, stejně tak se náhodně určí kam má letadlo letět a jakého je typu. První letadlo je napevno určeno jako přilétající z kraje obrazovky, aby hráč nemusel dlouho čekat, než bude moci něco ovládat.

Letadla a helikoptéry mají společnou třídu Vehicle, ve které je uveden konkrétní typ stroje, jeho poloha, rychlost, aktuální směr, aktuální stav, model, trajektoriiurčenou k přistávání a vzlétání a další pomocné proměnné. Třída `Vehicle` také, obsahuje metody, pro výpočet pozice stroje, výpočet heat plochy (postprocessing) a vykreslování trajektorie příslušného stroje.

5.6.4 Střídání dne a noci

Implementace střídání dne a noci obnáší změnu pozice zdroje světla a změnu intenzity světla. Pozice zdroje světla má vliv na směr osvětlení všech objektů a na směr stínů, které vrhají objekty ve scéně. Intenzita světla v tomto případě znamená umělé tlumení světlosti vykreslovaných fragmentů podle stádia dne. Křivka denního cyklu je rozdělena na 4 stejně dlouhé části (obrázek 5.14). První část je postupný náběh intenzity světla, druhá je plná intenzita světla (den), třetí je pokles intenzity a čtvrtá část je noc.



Obrázek 5.14: Křivka intenzity světla

Ve stádiu plné noci jsou vypnuty stíny, protože ani v noci není světlost scény nulová a bylo by vidět jak se stín vrací stejnou cestou. Zdroj světla se pohybuje po ose x lineární rychlostí v závislosti na čase, po ose y je jeho pozice upravována pomocí funkce `cos` a pozice na ose z je nastavena na konstantní hodnotu.

5.6.5 Skóre

Systém skóre je realizován pouze lokálně pro konkrétní zařízení, na kterém je hra spuštěna. O celýsystém bodování se stará třída `score`. Data se ukládají přes `NSUserDefaults`, což je systémová třída umožňující rychlé ukládání a načítání dat v rámci dané aplikace.

Třída `score` obsahuje metody pro přidání určité částky herních peněz (i záporné), zobrazení klávesnice pro zadání jména hráče a uložení a načtení celého skóre.

Pro lepší vizuální efekt je přičítání peněz uděláno postupně v závislosti na čase. Na herní obrazovce vlevo nahoře je oblast vyhrazená pro zobrazení aktuálního score, pokud mají být hráči přičteny nové peníze, tak zleva přiletí text s konkrétní částkou.

5.7 Úrovně efektů na různých zařízeních

Hra je univerzální jak pro iPhone (iPod Touch), tak pro iPad. Jednotlivá zařízení se výkonově velmi znatelně liší, z toho důvodu je třeba vyvážit množství použitých efektů a složitost scény tak, aby hra byla hra dobře hratelná na všech podporovaných typech zařízení. Použití efektů jsem rozdělil následujícím způsobem:

	shadow mapping	**postprocessing**	**PCF**
iPhone 3GS	ano	ano	ne
iPhone 4	ano	ano	ne
iPhone 4S	ano	ano	ano
iPad 1	ne	ano	ne
iPad 2	ano	ano	ne
iPad (3rd gen)	ano	ano	ano

Tabulka 5.2: Použití efektů na jednotlivých zařízeních

Jak je z tabulky 5.2 vidět, efekt postprocessingu (teplý vzduch) je použit na všech typech zařízení a to z toho důvodu, že tryskových letadel, u kterých se tento efekt projevuje, je za běžných okolností zaráz aktivních jen několik málo. Na zařízení iPad 1 nebylo možné použít dynamické stíny z důvodu jeho nedostatečné rychlosti.

5.7.1 Rozeznání iPhone/iPad

K rozpoznání typu zařízení slouží vlastní metoda `machine`, která vrací typ zařízení jako jednoduchý řetězec. Nastavení použitých efektů se pak dělá dynamicky po spuštění aplikace, poté co je rozpoznáno o jaké zařízení jde. Z důvodu dvou různých základních rozlišení je nutné zavést dvoje pozicování a velikosti všech GUI prvků.

Pro iPhone i iPad je použita naprosto stejná herní mapa, aby na menším iPhone nevypadalo vše tak drobně, je přiblížena kamera. Všechny velikosti, rychlost a vlastnosti letadel jsou upraveny pro konkrétní rozlišení, aby byla hra co nejlépe hratelná.

6 Publikace na App Store

App Store jecentralizovaný systém umožňující uživatelům prohlížení a nákup aplikací z elektronického obchodu iTunes Store na všechna výše uvedená zařízení.

Na počítači vývojáře musí být vygenerován elektronický podpis, na základě kterého Apple prostřednictvím serveru určeného pro vývojáře (developer.apple.com) vygeneruje certifikát, kterým je možné podepisovat vyvíjené aplikace. Pro novou aplikaci se musí v sekci Provisioning Portal vytvořit "App ID", které bude sloužit jako unikátní identifikátor aplikace. Poté je třeba vytvořit Distribution Provision Profile a spárovat jej z vytvořeným App ID.

K umístění aplikace na App Store je zapotřebí na internetovém portálu iTunes Connect (obrázek 6.1), vytvořit profil aplikace spárovat jej s vytvořeným App ID, vyplnit všechny požadované informace, nahrát screenshoty, popis a označit aplikaci jako "Ready for upload", to umožní upload aplikace přímo z prostředí Xcode.

Obrázek 6.1: iTunes Connect

Poté co je vytvořen Provision Profile pro danou aplikaci, je třeba jej stáhnout do počítače, nahrát jej do Xcode a tam ho přiřadit aplikaci. V Xcode je třeba nastavit vývoj pro zařízení, nikoli pro simulátor a následně spustit kompilaci v režimu "archive". Po úspěšné kompilaci se zobrazí okno "Organizer", kde je možné vybrat ze všech aplikací přeložených pro archivaci. Po výběru té správné stačí kliknout na tlačítko Distribute, které se zeptá na přihlašovací údaje k vývojářskému účtu. Po úspěšné autentizaci se naskytne několik možných kroků, vytvoření balíku aplikace pro speciální použití (Enterpriseor Ad-Hoc Deployment), vytvoření balíku pro testovací účely (pro beta testery) nebo odeslání aplikace do App Store. Při volbě odeslání na App Store se aplikace pomocí použitého distribučního profilu spáruje s aplikací vytvořenou na iTunes Connect a pokud nenastanou neočekávané komplikace, aplikace se začne nahrávat na server.

Pokud je aplikace úspěšně uploadována neznamená to, že je vyhráno. Nyní je v iTunes Connect potřeba aplikaci nastavit jako připravenou pro schvalovací proces.

6.1 Schvalovací proces

Aplikace musí splňovat podmínky pro schválení stanovené společností Apple. Podmínky jsou sepsány v dokumentu App Store Review Guidelines [20], který je napsán jako výpis nepovoleného chování aplikace.

Následující položky jsou nejčastějšími a nejvážnějšími důvody zamítnutí, aplikace bude zamítnuta pokud aplikace: padá, obsahuje vážné chyby, chování se neshoduje s popisem aplikace, obsahuje použití neveřejného API, je to demo (beta, trial..) verze, iPhone aplikace musí být spustitelná na iPad, obsahuje umělé zvyšování prokliků u reklam a podobné podvodné chování, obsahuje násilné obrázky/videa skutečných zabitých, mučených, postřelených lidí nebo zvířat, pornografie, zesměšňuje náboženství.

Aplikace bez rozšířené funkcionality jsou zpravidla zamítány (aplikace která zobrazí pouze integrovaný webový prohlížeč, pouze obrázek a podobné), takže vyzkoušet si vytvořit a odeslat pokusnou nic neumějící aplikaci je zbytečné.

Schvalovací doba bývá okolo 10ti dnů, záleží to na aktuální zátěži schvalovacího centra. Po uploadu aplikace do iTunes Connect. je aplikace ve stavu "Waiting for review", poté co se dostane na řadu (jednotky až desítky dnů) bude ve stavu "in review". Při úspěšném schválení bude ve stavu "approved" a v opačném případě "rejected".

Každá aplikace nejprve projde automatickou kontrolou, kde se kontroluje použití privátního API (nezdokumentované třídy, metody, funkce...), pokud tímto testem projde, testuje ji člověk.

7 Závěr

Cílem práce bylo vytvořit dobře vypadající aplikaci na mobilní zařízení, která bude využívat technologii shaderů. Tento hlavní cíl se mi povedlo úspěšně splnit. Ze stanovených podcílů jsem podle mých představ implementoval dynamické osvětlení, dynamické stíny (metodou Shadow mapping) a postprocessing efekt pro horký vzduch za letadly. Implementace vody byla splněna pouze částečně, shader vody je z důvodu rychlosti aplikace velmi jednoduchý.

Před začátkem této práce jsem neměl žádné zkušenosti s programováním shaderů a obecně s pokročilejší 3D grafikou. Tato práce mi dala silný základ pro budoucí vývoj dalších grafických aplikací.

Práce je průřezem přes problematiku vývoje pro systém iOS, je psána tak, aby byla přínosná i pro čtenáře neznalého zařízení a systému od společnosti Apple.

V práci je obecně popsáno, jak by mohla vypadat základní struktura OpenGL aplikace (hry). Jsou zde ukázány moje vlastní základní programové struktury, které mohou být přínosem pro začínající vývojáře.

V budoucnu je možné projekt vylepšovat hned v několika směrech: zdokonalení metody dynamických stínů, vylepšení postprocessing efektu, zlepšení hratelnosti, přidání herního obsahu (mapy, modely) nebo vylepšení zmiňovaného shaderu vody.

Literatura

[1] KHRONOS GROUP. *OpenGL* [online]. 1997 [cit. 2012-05-18]. Dostupné z: http://www.opengl.org/

[2] MARK, Dave a Jeff LAMARCHE. *IPhone SDK*: *Průvodce vývojem aplikací pro iPhone a iPod touch*. Brno: ComputerPress, a.s., 2010. ISBN 978-80-251-2820-6.

[3] GLSL tutoriál, osvětlení. In: *Lighthouse3d* [online]. 2011 [cit. 2012-01-15]. Dostupné z: http://www.lighthouse3d.com/tutorials/glsl-tutorial/

[4] Allabout OpenGL ES 2.x. In: BOMFIM, Diney. *Db-interactively* [online]. 2011 [cit. 2012-05-18]. Dostupné z: http://db-in.com/blog/2011/01/

[5] Přehled funkcí OpenGL ES a GLSL ES. KHRONOS GROUP. *OpenGL* [online]. 2010 [cit. 2012-05-18]. Dostupné z: http://www.khronos.org/opengles/sdk/docs/reference_cards/OpenGL-ES-2_0-Reference-card.pdf

[6] Apple hlásí rekordní čtvrtletí: tržby přes 28 miliard dolarů. *Apple Magazín* [online]. 2011 [cit. 2012-05-18]. Dostupné z: http://applemagazin.cz/tag/rekord/

[7] Objective-C. ČADA, Ondra. *OCS* [online]. 1999 [cit. 2012-05-18]. Dostupné z: http://www.ocs.cz/text/ObjectiveC/index.html

[8] TheComplete List of Objective-C 2.0 @ CompilerDirectives. ITTERHEIM, Steffen. *Learn& Master Cocos2D Game Development* [online]. 1999 [cit. 2012-05-18]. Dostupné z: http://www.learn-cocos2d.com/2011/10/complete-list-objectivec-20-compiler-directives/

[9] Objective-C Beginner'sGuide. *Otierney* [online]. 2004 [cit. 2012-05-18]. Dostupné z: http://www.otierney.net/objective-c.html

[10] IOS Memory Management: Using Autorelease. *Anderson Software Group, Inc.*: *Bits, Bytes&Words* [online]. 2011 [cit. 2012-05-18]. Dostupné z: http://www.asgteach.com/blog/?p=73

[11] Xcode 4: the super megaawesomereview. PILKINGTON, Martin. *Pilky.me* [online]. 2011 [cit. 2012-05-18]. Dostupné z: http://pilky.me/view/15

[12] Mac OS X Developer Library. APPLE. *Apple* [online]. 2010 [cit. 2012-05-18]. Dostupné z: http://developer.apple.com/library/mac/

[13] Apple: PressInfo. APPLE. *Apple* [online]. 2012 [cit. 2012-05-18]. Dostupné z: http://www.apple.com/pr/products/ios/ios.html

[14] Zabalení floatu do RGBA. In: *GameDev* [online]. 2007 [cit. 2012-02-03]. Dostupné z: http://www.gamedev.net/topic/442138-packing-a-float-into-a-a8r8g8b8-texture-shader/

[15] KRŠEK, Přemysl a Michal ŠPANĚL. FIT VUT V BRNĚ. *Základy počítačové grafiky*: *Osvětlení a stínování 3D objektů* [online]. 2012 [cit. 2012-03-11]. Dostupné z: https://www.fit.vutbr.cz/study/courses/IZG/private/lecture/izg_slide_osvetleni_stinovani_print.pdf

[16] Understanding Automatic Reference Counting in Objective-C. *The Long WeekendWebsite* [online]. 2011 [cit. 2012-05-18]. Dostupné z: http://longweekendmobile.com/2011/09/07/objc-automatic-reference-counting-in-xcode-explained/

[17] IOS Developer Library: Delegation. APPLE. *Apple* [online]. 2012 [cit. 2012-05-18]. Dostupné z: http://developer.apple.com/library/ios/#documentation/General/Conceptual/DevPedia-CocoaCore/Delegation.html

[18] List of iOS devices. In: *Wikipedia*: *the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2012 [cit. 2012-05-18]. Dostupné z: http://en.wikipedia.org/wiki/List_of_iOS_devices

[19] Coordinate Systems in OpenGL. In: RATH, Egon. *Egon Rath's Notes* [online]. 2012 [cit. 2012-05-06]. Dostupné z: http://www.matrix44.net/cms/notes/opengl-3d-graphics/coordinate-systems-in-opengl

[20] App Store Review Guidelines. *Apple* [online]. 2012 [cit. 2012-05-19]. Dostupné z: https://developer.apple.com/appstore/resources/approval/guidelines.html

Seznam příloh

1. CD