Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

Tenký grafický klient pro mobilní zařízení

Plzeň 2013

Michal Žák

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 2. května 2013

Michal Žák

Abstract

Visualization of data is a common need in the industrial automation software. These data come from multiple sources and by visualizing we display them together with semantic information. Using cross-platform technologies, our goal is to provide simple and thin graphical client which can work on mobile devices and be controlled through its interface or defined protocol. The rendering paradigm that we use is scene graph representation.

Poděkování

Chtěl bych poděkovat především panu Ing. Petru Vaněčkovi, Ph.D. za cenné rady a připomínky při vedení práce a firmě ICONICS Europe B. V. – o. z. za zapůjčení hardwaru nezbytného k vývoji pro platformu iOS.

Obsah

1 Úvod **1**

2 Software GENESIS **2**

2.1 GraphWorX 2

2.2 Komponenta 3D View 4

2.2.1 Sada operací komponenty 3D View 5

3 Vizualizace scény **6**

3.1 Objekty a jejich transformace 6

3.2 Graf scény 7

3.3 Kamery 7

3.4 Výběr objektů 8

3.4.1 Softwarové řešení 8

3.4.2 Kódování objektů barvou 8

3.4.3 Picking využívající geometry shader 9

3.5 Existující frameworky grafu scény 9

3.5.1 OpenSG 9

3.5.2 Open Scene Graph 10

3.5.3 Další implementace 10

4 OpenGL ES a prostředí iOS **11**

4.1 Stručný přehled verzí OpenGL ES 11

4.2 Pipeline OpenGL ES 11

4.3 Přehled API OpenGL ES 13

4.3.1 Základní funkce a stavy OpenGL 13

4.3.2 Práce s shadery 13

4.3.3 Vytváření a plnění zdrojů 14

4.3.4 Kreslení 15

4.4 Vývoj pro iOS 15
4.5 Xcode 16
4.6 GLKit 16

5 Architektura grafického klienta **18**
5.1 Platformní nezávislost 18
5.2 Společný předek Generic 19
5.3 Systémové a řídicí třídy 20
5.4 Grafické zdroje 21
5.4.1 Technika kreslení 21
5.4.2 Trojúhelníkové meshe 22
5.4.3 Textury 23
5.5 Třídy obsluhující scénu 24

6 Rozhraní grafického klienta **26**
6.1 Meshe a textury 26
6.2 Uzly scény a kamery 26
6.3 Ostatní operace rozhraní 27

7 Ukázkový protokol pro ovládání klienta **29**
7.1 Příkazy protokolu 29
7.1.1 Správa meshů 29
7.1.2 Správa uzlů scény 30
7.1.3 Správa kamery 31
7.1.4 Dávka příkazů 32
7.2 Interpretující třída 32
7.3 Interaktivní zadávání příkazů ve Windows 33

8 Ukázkové scény **35**
8.1 Použité meshe 35
8.2 Benchmark 36

9 Závěr **37**

A Příloha: Ukázky scén **39**

B Příloha: Použité verze shaderů **44**

C Příloha: Obsah CD **45**

1 Úvod

V průmyslové sféře se setkáváme s potřebou zpracovávat a následně vizualizovat různá data. Tato data získáváme z mnoha zdrojů, například ze stavu výrobní linky nebo aktuálního nastavení klimatizace objektu. Sjednocené zobrazené údaje pak umožňují operátorovi snazší kontrolu situace.

Často je nutné data vizualizovat jako trojrozměrnou scénu. Za prvé mohou být původní data spjata s konkrétními souřadnicemi, za druhé tak můžeme usnadnit orientaci ve vizualizaci a za třetí se trojrozměrné zobrazení často jeví jako atraktivnější. Chceme-li zobrazovat prostorově, je vhodné sestavit grafického klienta, který nám naše záměry usnadní, neboť knihovní funkce OpenGL, OpenGL ES a DirectX poskytují až příliš nízkoúrovňová volání a je potřeba je vhodně obalit.

Dalším cílem je snadná přenositelnost klienta, tedy jeho nezávislost na konkrétní platformě. V dnešní době totiž existují různá mobilní zařízení (chytré telefony a tablety), která na rozdíl od klasických počítačů disponují často jediným možným operačním systémem, kterému se musíme přizpůsobit. Výměnou za to však provozujeme aplikaci na stroji, který není umístěný napevno na jednom místě, dodávku elektřiny potřebuje pouze k dobíjení své vlastní baterie a často má i nižší pořizovací cenu. Tato práce popisuje implementaci klienta pro dvě platformy, Windows a iOS.

Pro ověření správné funkčnosti potřebujeme ukázkové scény. S jejich pomocí lze ověřit i výkon klienta, a tak nalézt případné nedostatky.

2 Software GENESIS

GENESIS je sadou aplikací firmy Iconics pro sledování a následnou vizualizaci průmyslových dat. Software staví na více vrstvách, jednotlivé produkty mezi sebou mohou komunikovat a předávat si měřená či syntetizovaná data. Uživatel si může vybrat mezi 32 a 64 bitovou verzí softwaru.

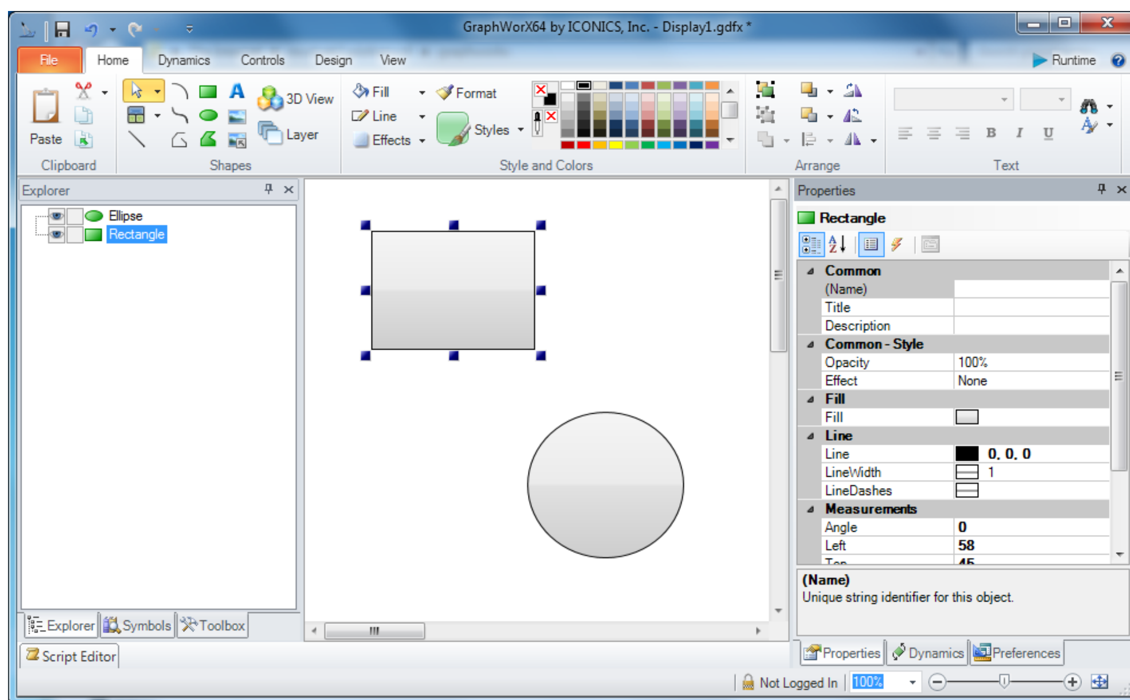
Sada GENESIS souvisí s odvětvími HMI a SCADA. Zkratka HMI vyjadřuje human-machine interface, tedy zařízení, kde uživatel monitoruje celý proces a může do něj zasáhnout. Do této oblasti patří právě například 3D vizualizace, kterými se tato práce zabývá. SCADA (supervisory control and data acquisition) pak značí centralizované systémy řízení rozsáhlých průmyslových komplexů. Oproti jednoduchému řízení se navíc do systému vnáší možnost zásahu, slouží jako nadstavba více průmyslovým počítačům.

Popišme nyní aplikaci GraphWorX ze sady GENESIS, která souvisí se samotnou vizualizací dat, tím pádem i s HMI. Informace o ostatních aplikacích nalezne laskavý čtenář na oficiálních stránkách produktu GENESIS [2].

2.1 GraphWorX

GraphWorX slouží k návrhu vizualizace. Uživatel postupně skládá grafické prvky a připojuje k nim žádané akce. Výsledek je poté možné si prohlédnout v runtime módu tak, jak ho spatří operátor. Při produkci zobrazuje software vytvořený displej s využitím technologie WPF (Windows Presentation Foundation), případně Microsoft Silverlight.

Kreslicí plátno uprostřed obrazu je obklopeno ovládacími a přehledovými panely. Horní ovládací panel nabízí v několika záložkách objekty, které můžeme umístit na plátno. Uživatel má k dispozici rovněž hierarchický přehled používaných objektů a jejich vlastností (viz obrázek 2.1). Umístění panelů není pevné, software umožňuje uspořádat pracovní prostředí jinak.



Obrázek 2.1: Prostředí aplikace GraphWorX.

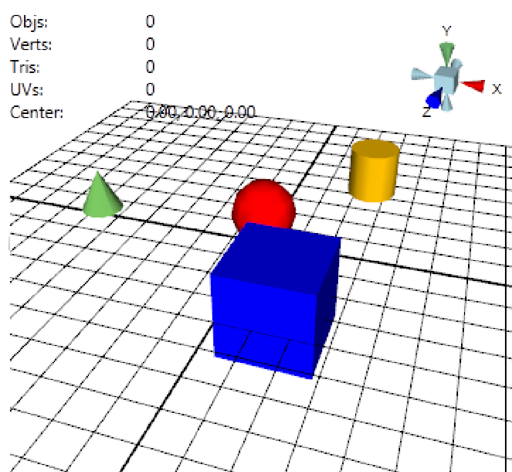
Umisťovanými objekty mohou být jednoduché vektorové tvary (obdélník, elipsa, mnohoúhelník), bitmapy (u kterých lze zvolit, zda mají být vloženy přímo do displeje, nebo se má uchovat pouze odkaz na ně) i různé předdefinované panely. Vložit můžeme také objekt zvaný *3D View*, který poskytuje zobrazování trojrozměrné scény. Pro úpravu vzhledu objektů poskytuje GraphWorX klasické možnosti jako změnu výplně, obrysu a umístění v ose Z (z důvodu správného překrývání 2D objektů se takto volí pořadí, v jakém se kreslí), ale i poněkud netradiční efekty známé z Microsoft Office: stín, rozmazání, záři.

Komponenty, které poskytují displeji určitou dynamičnost, jsou zastoupeny klasickými prvky GUI typu radio button, check box, edit box, ale i akcemi, které ovlivňují objekty umístěné na displeji. Akce a prvky GUI propojíme se zvoleným zdrojem dat *(data source)*, který může odkazovat do databáze, získávat data z přidružených aplikací GENESIS atd. Průběh dat lze rovněž simulovat, lokální simulátor pak poskytne například sinusoidový signál. V prvcích GUI se získané hodnoty mapují celkem přímočaře na zobrazovaný text nebo zaškrtnutí, akce se získanou hodnotou může zacházet už poněkud složitějším způsobem – aplikuje ji například na transformaci nebo barvu objektu, s nímž je akce spojena.

GraphWorX nabízí dále také komponenty související s ostatními aplikacemi sady GENESIS. Tímto způsobem můžeme sloučit více pohledů do jednoho displeje. Za zmínku stojí rovněž volba vzhledu displeje, což představuje velikost, barvu pozadí, případně šablonu vzhledu. Uživatel může upravit také pohled na scénu, aktivovat mřížku a zobrazit souhrnné statistiky.

2.2 Komponenta 3D View

Jelikož se tato práce zabývá trojrozměrnou vizualizací na mobilních zařízeních, podíváme se nyní podrobněji na komponentu 3D View (obrázek 2.2). Vzniklý grafický klient by měl být z velké části schopný operací, které tato komponenta poskytuje.



Obrázek 2.2: Komponenta 3D View obsahující několik objektů.

V design módu umožnuje 3D View (podobně jako plátno celého displeje v GraphWorX) umístění jednotlivých primitiv či meshů (například krychli, kouli, kužel a válec), každému lze přidělit identifikační název, podle něhož může být vyhledán. Objekty lze seskupit i rozdělit, přesouvat, rotovat a měnit jejich velikost. 3D View proto zobrazuje pomocnou mřížku známou z 3D softwaru pro modelovaní a animace.

Vzhled objektu ovlivňuje přiřazený materiál, který je popsán čtyřmi složkami osvětlovacího modelu (ambient, diffuse, specular a emissive) sečtenými po osvětlení ve výslednou barvu. Namísto jednolité barvy můžeme použít texturu, která se na model namapuje podle jeho texturovacích souřadnic. Tato možnost umožňuje více realistické vnímání scény.

Komponenta 3D View nabízí i nastavovaní kamer (pohledů na scénu), mezi nimiž pak lze přepínat. Tohoto chování se dá dobře využít u rozsáhlejších scén. Scény je vždy osvětlena jedním světlem, které se nachází na souřadnicích pozorovatele (aktivní kamery).

Akcí, které můžeme na objekty ve scéně připojit, nalezneme v GraphWorX více. První možností je aplikování transformace (změna pozice, rotace, velikosti podle hodnoty z data source), druhou změna vlastností materiálu, což odpovídá akci *Color* ve 2D prostředí. V nabídce se vyskytuje rovněž akce *Pick*, která nastává po výběru objektu uživatelem (konkrétně po kliknutí definovaným tlačítkem).

2.2.1 Sada operací komponenty 3D View

Sadu operací komponenty 3D View, které si tato práce klade za cíl naimplementovat do mobilního grafického klienta a poskytnout odpovídající přístupové aplikační rozhraní, tvoří:

- rendering trojúhelníkových meshů;
- konfigurace materiálů a jeho přiřazení meshům;
- aplikování transformací (translace, rotace, aj.);
- vytváření a správa kamer;
- osvětlení scény z pozice aktivní kamery;
- picking.

3 Vizualizace scény

Pod pojmem vizualizace rozumíme vytváření obrazu či animace za účelem předání informace. V našem případě dosáhneme výsledku za pomoci renderingu. Vektorová data jsou zaslána ke zpracování grafické kartě (avšak mohou být zpracována i softwarově na procesoru), kde jsou postupným sledem algoritmů přeměněna na rastrový obraz.

Grafická rozhraní poskytují operace na úrovni elementárních prvků, ať se jedná o manipulaci s buffery, texturami nebo shadery. Nad tyto operace potřebujeme postavit logiku, která bude spolehlivě spravovat scénu a elementární prvky ponese skryté uvnitř.

3.1 Objekty a jejich transformace

V základní podobě můžeme scénu chápat jako množinu rozmístěných, případně jinak transformovaných objektů, což mohou být např. trojúhelníkové meshe, křivky a plochy, ale i objekty se speciálním významem – kamery a světla. Tyto pomocné objekty nelze přímo zobrazit, ale lze je využít pro globální ovlivnění renderingu. Každé umístění definuje jeho transformační matice, u kamery definujeme navíc i matici projekční.

Pro naše účely uvažujme, že každý zobrazitelný objekt tvoří množina trojúhelníků a jeho umístění je definováno transformační maticí Q_W. Zvolíme jednu kameru scény jako aktivní, její projekční matici označíme Q_P, transformační matici definující její umístění pak Q_V. Všechny vrcholy (trojúhelníků) jsou poté transformovány podle vztahu 3.1.

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = Q_P\, Q_V\, Q_W \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \tag{3.1}$$

Tento přístup však neumožňuje nastavit implicitně závislost na jiném objektu. Pokud bychom chtěli vizualizovat scénu obsahující například projíždějící vlak, musíme v každém snímku nastavit transformaci lokomotivy i všech vagonů zvlášť.

3.2 Graf scény

Popisovaný způsob přídává do scény možnost relativních transformací vůči nadřazenému objektu v hierarchii.

Scénu reprezentujeme acyklickým, stromovým grafem s právě jedním kořenovým uzlem. Každá hrana grafu značí závislost, přičemž každý uzel smí záviset právě na jednom jiném, jediným nezávislým uzlem je kořen. Jak již bylo zmíněno, v grafu nesmí vznikat cykly, tedy žádný objekt nesmí rekurzivně záviset sám na sobě.

Hlavní výhodou tohoto uspořádání je skládání transformací – matice Q_W je nyní definována jako součin transformačních matic uzlů cesty od kořenu k danému objektu. Můžeme tak vytvářet skupiny objektů ovlivnitelné transformací jednoho uzlu. Nad grafem scény může být postavena další pomocná datová struktura pro urychlení vykreslování, například strom octree.

3.3 Kamery

Kamery představují pohled na scénu pomocí své projekční matice, která definuje parametry promítání, dokonce i jeho způsob (ortogonální nebo perspektivní). Abychom přemístili pozorovatele do jiné pozice, můžeme rozšířit definici kamery o matici pohledu, avšak to se zdá být nevhodné ve chvíli, kdy lze využít grafu scény. Přiřadíme-li kameře nadřazený uzel scény, můžeme použít jeho transformaci pro výpočet matice pohledu podle vztahu 3.2, kde Q_V značí výslednou matici pohledu a Q_{W_i} matici transformace i-tého uzlu ve směru od kořenu. Jedná se o afinní transformace, což umožňuje urychlení vypočtu inverzní matice oproti obecnému algoritmu.

$$Q_V = \left(Q_{W_N}\, Q_{W_{N-1}}\, \dots\, Q_{W_1}\, Q_{W_0}\right)^{-1} \tag{3.2}$$

3.4 Výběr objektů

Ve vizualizaci můžeme od uživatele vyžadovat zpětnou vazbu, konkrétně výběr některého z objektů ve scéně pomocí myši či dotykového displeje (operace označovaná jako picking). Uveďme několik přístupů, jak toho docílit.

3.4.1 Softwarové řešení

Nejprve hledáme matice ${Q_P}^{-1}$ (inverzní k projektivní) a ${Q_V}^{-1}$ (inverzní k matici pohledu). Následně jimi transformujeme souřadnice kurzoru (x, y) a získáváme jeho pozici X v prostoru scény. Vektor **s** vzniklý jako rozdíl bodu X a pozice kamery C určuje směr polopřímky, která vybírá s ní kolidující objekty. Celý výpočet je popsán ve vztahu 3.3. Proměnné *width* a *height* představují rozměry plátna, do kterého rendering probíhá.

$$X = {Q_V}^{-1} {Q_P}^{-1} \begin{pmatrix} \frac{2x}{width} - 1 \\ 1 - \frac{2y}{height} \\ 0 \\ 1 \end{pmatrix} \qquad (3.3)$$

$$s = X - C$$

Nyní stačí nalézt objekty kolidující se vzniklou polopřímkou, z nichž vybereme nejméně vzdálený od bodu C. Algoritmus lze urychlit hierarchickým uspořádáním geometrie ve scéně nebo např. použitím bounding boxů. Výhodou jsou nulové požadavky na grafické rozhraní a v případě malého počtu objektů i rychlá odezva. Se stoupajícím počtem objektů nebo trojúhelníků pocítíme značné zpomalení aplikace při výběru objektů.

3.4.2 Kódování objektů barvou

Implementačně jednodušší a hardwarově akcelerovaný výběr lze provést tak, že každému objektu (případně i trojúhelníku) přidělíme číselný identifikátor, který

převedeme na odpovídající unikátní barvu. Následně provedeme rendering, kreslíme všechny objekty jednolitě jejich barvou. Po dokončení přečteme barvu pixelu na souřadnicích kurzoru, převedeme ji zpět na identifikátor (pokud se nejednalo o barvu pozadí) a vyhledáme objekt v příslušné datové struktuře.

Pro vysoké počty objektů (trojúhelníků) na scéně se tato metoda obecně jeví jako rychlejší. Navíc není potřeba udržovat geometrii v paměti přístupné procesoru, může být nadále uchována v grafické paměti. Zpomalujícími prvky mohou však být vysoké rozlišení (lze řešit podvzorkováním) a záverečné čtení barvy z color bufferu.

3.4.3 Picking využívající geometry shader

Detekci kolize paprsku s geometrií lze přesunout i na GPU, kde ji řeší geometry shader [9]. Oproti pickingu založenému na kódování barvou potřebujeme výstup z pixel shaderu o velikosti pouhého jednoho pixelu, který uchovává identifikátor objektu. Tento jediný pixel nám pak poslouží k vyhledání vybraného objektu. OpenGL ES 2.0 bohužel nedokáže pracovat s geometry shadery, a tak zde nejsme schopni toto řešení implementovat.

3.5 Existující frameworky grafu scény

Uspořádání scény do grafu je často používané řešení a existuje mnoho frameworků, například OpenSG a Open Scene Graph, které poskytují často mnohem robustnější implementaci, jež je pro naše účely v podstatě nadbytečná. I tak při své velikosti a užití mohou sloužit jako dobrý vzor pro vývoj vlastního řešení.

3.5.1 OpenSG

OpenSG implementuje graf scény pro mnoho platforem (Windows, Mac OS X, Linux a Solaris) a staví na OpenGL. O framework se stará seskupení OpenSG Forum a je šířen pod licencí LGPL (jedná se tedy o open source produkt). V grafu scény jsou geometrická data umísťována pouze do listů (nazvané interior node), průchozí uzly interpretují různé akce (seskupení objektů, transformace a další) [8].

3.5.2 Open Scene Graph

Ačkoliv název může čtenáře zmást, že se jedná o již výše popisovaný produkt OpenSG, opak je pravdou. Počátky vývoje Open Scene Graph a OpenGL se datují do přibližně stejné doby, což vedlo ke zvolení podobných názvů. Jedná se o framework šířený pod vlastní licencí *OpenSceneGraph Public License*, která je založena na LPGL. Vývojář má tedy k dispozici zdrojové kódy, přičemž se využívá jazyka C++.

3.5.3 Další implementace

Firma VSG vyvíjí komerční objektové API Open Inventor. Kromě původní multiplatformní implementace v C++ jsou dnes k dispozici varianty také pro platformy .NET a Java. VSG rovněž poskytuje několik rozšíření, například vizualizaci objemových dat nebo rendering pomocí raytracingu.

Pro platformu Java existují například knihovny Java3D, Jreality a Aviatrix3D, nicméně již několik let nevycházejí jejich nové verze. Z oblasti webových technologií (HTML5 a WebGL) jmenujme OSG.JS, aktivně vyvíjenou javascriptovou implementaci scene graphu, a X3D/X3DOM, které reprezentují scénu pomocí jazyka XML.

4 OpenGL ES a prostředí iOS

Pro akcelerované vykreslování potřebujeme využívat takové rozhraní, které zajistí komunikaci s grafickou kartou nebo čipem. Zatímco u klasických desktopových počítačů se setkáváme s OpenGL (a v operačním systému Windows i s Direct3D, což je součást frameworku DirectX), mobilní platforma implementuje OpenGL ES, které bylo navrženo s ohledem na nižší výkon zmíněného hardwaru jako podmnožina klasického OpenGL.

4.1 Stručný přehled verzí OpenGL ES

OpenGL ES 1.0 a 1.1 vychází ze specifikace OpenGL verze 1.3, resp. verze 1.5 [4]. Rendering probíhá pouze za pomoci fixed function pipeline, tedy pevně nadefinovanou cestou, programovatelnost není součástí normy. Ne všechna zařízení pracující s OpenGL ES podporují operace s čísly v pohyblivé řádové čárce, a tak jsou aplikaci zpřístupněna i volání předávající hodnoty ve formátu pevné řádové čárky.

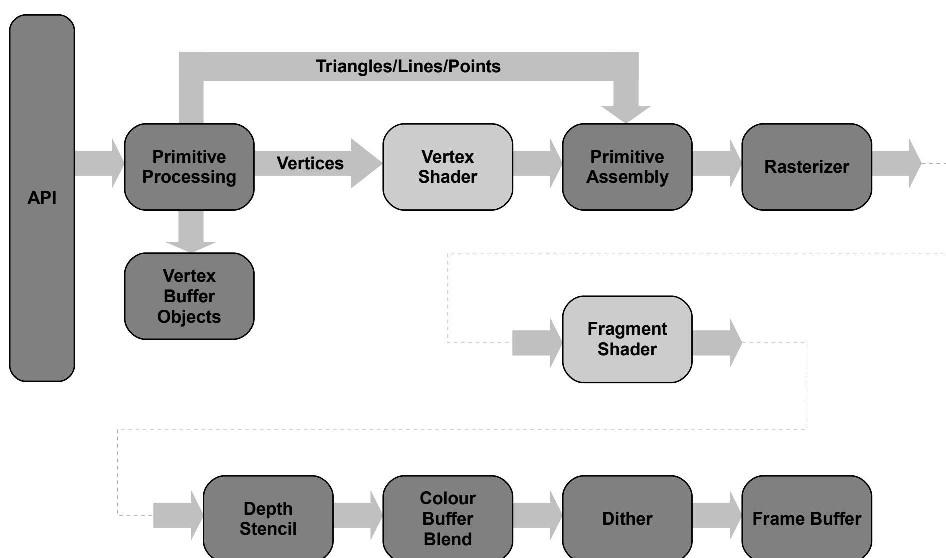
V roce 2010 vyšla specifikace OpenGL ES ve verzi 2.0 [5]. S předchozí verzí není kompatibilní, kreslení cestou fixed function pipeline již není dále podporováno. Tu nahradila programovatelná pipeline a je explicitně nutné naimplementovat operace na úrovni vrcholů (vertex shader) a fragmentů (fragment shader).

Tento rok (2013) byla zveřejněna oficiální specifikace OpenGL ES verze 3.0. Vzhledem k momentálně nízkému rozšíření zařízení podporujících nejnovější verzi se OpenGL ES 3.0 tato práce nezabývá.

4.2 Pipeline OpenGL ES

Přejdeme k hardwarové implementaci zpracování geometrie. Každý požadavek na rendering je zpracován vnitřní *pipeline*, která obsahuje neměnné a programovatelné prvky. Fixní části mohou být díky své jednoduchosti (nevykonávají jiný typ operace) optimalizovány již na úrovni hardwaru.

Na počátku zpracováváme předanou množinu primitiv daného typu: body, čáry, trojúhelníky. Typ je pro celou množinu neměnný, pokud bychom chtěli vykreslit např. trojúhelníky i body, nezbyde nám jiná možnost, než rozdělit operaci na dvě. Jednotlivé vrcholy primitiv následně prochází programovatelným vertex shaderem. Ten je psaný v jazyce GLSL *(OpenGL Shading Language)*, přičemž standardně slouží k transformaci vrcholů a k výpočtu dalších hodnot k nim se vztahujících. Vzhledem k tomu, že tato část je programovatelná, může posloužit i k méně obvyklým výpočtům. Výpočet je prováděn nezávisle na ostatních vrcholech, což umožňuje celý proces paralelizovat.



Obrázek 4.1: OpenGL ES 2.0 Pipeline. Zdroj: `khronos.org/opengles/2_X`

Po průchodu vertex shaderem se provede rasterizace primitiv a vzniklé fragmenty putují do fragment shaderu, opět programovatelné součásti pipeline. Fragment odpovídá v podstatě jednomu pixelu, opět platí, že prováděné operace nezávisí na ostatních (nelze k nim ani přistupovat). Fragment shader je možné využít například pro výpočet per-pixel osvětlení. Jeho výstup se ořezává podle hodnot depth a stencil bufferu a nastavení depth a stencil testů, poté dochází k míchání (blendingu) s již existujícími pixely frame bufferu, ditheringu a zápisu do frame bufferu. Na obrázku 4.1 je přiblížena celá pipeline OpenGL ES 2.0.

4.3 Přehled API OpenGL ES

Popisovat kompletně celé API dle specifikace je samozřejmě nad rámec této práce, zmiňme však alespoň některé funkce, které využijeme pro vizualizaci scény. Tyto funkce jsou prostředkem k univerzální komunikaci s grafickým hardwarem, jednotným rozhraním ovladačů mnoha grafických karet a čipů. Stav renderingu a správu jednotlivých zdrojů má na starosti kontext OpenGL.

Příkazy OpenGL dodržují přesný formát pojmenování. Každý je uvozen nejprve sekvencí `gl`, následuje název funkce a pak počet proměnných a zkratka datového typu. Datovým typem může být celé číslo (integer, `i`), číslo v pohyblivé řádové čárce (float, `f`), případně pole (`v`). Jako příklad uveďme příkaz `glUniform1fv`: voláme funkci s názvem *Uniform*, a to ve variantě, kdy vyžaduje jako argument jedno pole čísel v pohyblivé řádové čárce.

4.3.1 Základní funkce a stavy OpenGL

Rendering se neprovádí přímo na obrazovku, ale do pomocného bufferu *(back bufferu)*, který je po dokončení kreslení propagován (stane se *front bufferem*). Tímto chováním předcházíme problikávání scény při renderingu. K vyplnění back bufferu jednotnou hodnotou barvy a hloubky slouží příkaz `glClear`. Barvu nastavíme pomocí funkce `glClearColor`, která si za své čtyři parametry bere barevné složky (r, g, b, a), hloubku funkcí `glClearDepth` vyžadující parametr v rozsahu od nuly do jedné.

V OpenGL existují booleovské stavy jako například `GL_DEPTH_TEST` (určuje, jestli má docházet ke kontrole hloubky a jejímu zápisu do depth bufferu), které lze zapínat a vypínat funkcemi `glEnable` a `glDisable`.

4.3.2 Práce s shadery

Pro práci s shadery musíme nejprve přistoupit k jejich kompilaci a linkingu. OpenGL nepracuje se soubory, ale s řetězci, tudíž je nutné nejprve shader načíst např. pomocí standardní knihovny jazyka C či C++. Nový shader vytvoříme voláním `glCreateShader`, přičemž jako parametr uvedeme typ shaderu (hlavičky OpenGL

definují konstanty `GL_VERTEX_SHADER` a `GL_FRAGMENT_SHADER`). Kód psaný v GLSL a uchovávaný v načteném řetězci předáme funkcí `glShaderSource`, načež spustíme kompilaci příkazem `glCompileShader`.

Vertex a fragment shader spojíme dohromady v jeden program, který nejprve vytvoříme voláním `glCreateProgram`, poté připojíme oba shadery použitím funkce `glAttachShader` a provedeme linking funkcí `glLinkProgram`. Pro kontrolu překladu shaderu i jejich linkingu můžeme získat logy celého procesu pomocí `glGetShaderInfoLog`, respektive `glGetProgramInfoLog`. Shadery uvolníme z paměti voláním `glDeleteShader`, program voláním `glDeleteProgram`.

Dostáváme se k samotnému použití (vykonání) programu složeného z přeložených shaderů. Aktivace programu je jednoduchá, stačí zavolat funkci `glUseProgram`, které předáme patřičný identifikátor. Předáme-li nulu, používaný program je odpojen a v klasické verzi OpenGL (nikoliv OpenGL ES) přebírá rendering fixní pipeline. V GLSL používáme globální konstanty *uniforms* například k předávání transformační matice renderingu. Ty předáme shaderům jednou z variant funkce `glUniform` (záleží na typu předávané proměnné a jejich počtu).

4.3.3 Vytváření a plnění zdrojů

Zdroji *(resources)* máme v našem případě na mysli vertex buffery, element array buffery, vertex arrays a textury. Vertex buffer, jak již název napovídá, slouží k uchovávání vrcholů, element array buffer slouží k uchování indexů pro nepřímé adresování vrcholů. Tímto způsobem jsme schopni během renderingu využít jeden vrchol vícekrát (odkazuje na něj více indexů). Vertex array object zapouzdřuje veškerý stav potřebný k rederingu, tj. formát vrcholů a připojené buffery. Textura uchovává bitmapu určenou k namapování na renderovaná primitiva.

Použití bufferů je v postatě jednoduché, vytvoření provedeme funkcí `glGenBuffers` vracející identifikátor vytvořeného objektu, připojení funkcí `glBindBuffer`, které naopak identifikátor předáváme. Data do bufferu umísťujeme voláním `glBufferData`, kdy specifikujeme velikost, ukazatel na data a způsob užití bufferu. Uvolnění provedeme funkcí `glDeleteBuffers`. Popsané funkce jsou shodné pro vertex i element array buffer.

Texturu vytvoříme voláním `glGenTextures`, poté ji musíme připojit pomocí `glBindTexture`, abychom s ní mohli nadále manipulovat. Jako parametry předáme cíl připojení a identifikátor vytvořené textury. Cílem je v našem případě vždy dvojrozměrná textura definovaná konstantou `GL_TEXTURE_2D`. Bitmapu do textury nakopírujeme funkcí `glTexImage2D`, která vyžaduje parametrů již více: cíl (opět `GL_TEXTURE_2D`), úroveň (používá se pro mipmaping), formát dat, rozměry, ukazatel na data bitmapy atd. Zatímco v klasickém OpenGL je možné předávat texturu v jiném formátu, než je pak uložena v grafické paměti, OpenGL ES konverzi uvnitř funkce nepodporuje. Uvolnění textury provádí operace `glDeleteTextures`.

4.3.4 Kreslení

Při kreslení aktivujeme program obsahující shadery funkcí `glUseProgram` a naplníme konstanty shaderů *(uniforms)* aktuálními hodnotami. Konstanty nelze nastavit před připojením programu. Pokud používáme textury, připojíme je již zmíněným voláním `glBindTexture`, mezi jednotlivými jednotkami při jejich větším používaném množství přepínáme funkcí `glActiveTexture`. Vertex array object připojíme analogicky funkcí `glBindVertexArray`, jako parametr uvedeme jeho identifikátor.

Nakonec zavoláme funkci `glDrawElements`, která sestaví primitiva z vrcholů vertex bufferu podle indexů uložených v element array bufferu. První parametr uvádí typ kreslených primitiv (např. body, úsečky, trojúhelníky), dalšími jsou počet indexů a jejich datový typ. Jako poslední parametr uvedeme offset prvního indexu v element array bufferu.

4.4 Vývoj pro iOS

S operačním systémem iOS se setkáme v produktech společnosti Apple Inc., jmenovitě iPhone, iPad, iPod Touch a Apple TV. Vychází z desktopového operačního systému Mac OS X.

Takřka nezbytným jazykem používaným pro vývoj je Objective C, nadstavba jazyka C, v němž jsou implementována například rozhraní pro programování aplikací Cocoa a Cocoa Touch. Ohraničíme-li bloky kódu psané v Objective C podmiňujícími

makry, můžeme zdrojové soubory kompilovat pro jinou platformu již jako standardní kód v jazyce C. Existuje rovněž jazyk Objective C++, který rozšiřuje C++.

Cocoa Touch API je objektovým rozhraním nabízející širokou škálu služeb (např. správu 3D grafiky, audia, sít'ovou komunikaci) [1]. Architektura model-view-controller, které se tento celek drží, přispívá k čitelnějšímu kódu.

4.5 Xcode

Standardním vývojovým prostředím pro iOS je Xcode. Tento komplexní nástroj umožňuje provádět prakticky všechny činnosti spjaté s vývojem aplikací – samozřejmostí je překlad, debugging (monitorování aplikace pro snazší hledání chyb), ale i profilování (měření výkonu a efektivnosti) kódu. Xcode se stará i o sestavení a nainstalování aplikace do cílového zařízení, kterým může být i simulátor hardwaru, na kterém běží iOS.

Podobně jako v dalších vývojových prostředích se zdrojové kódy a jejich nastavení slučují do projektů, přičemž programátor při vytváření vybírá z předem připravených variant, které pomohou odstranit nadbytečná a stále se opakující nastavování parametrů překladu.

Položky projektu mohou být zpracovány různě. Podle nastavení se s nimi zachází jako se zdrojovým kódem, pak Xcode provádí jejich překlad, ale lze je nastavit také jako datové soubory, které se mají po kompilaci zkopírovat do cílového zařízení. Toho využijeme pro jednoduché nasazení projektu včetně souborů, na kterých je závislý (v případě grafického programu se jedná například o shadery). Nasazovaný balíček se v terminologii vývoje pro iOS nazývá *bundle*.

4.6 GLKit

Pro usnadnění programování v OpenGL ES poskytuje Apple vlastní rozšiřující framework GLKit. Abychom jej mohli využít, cílové zařízení s iOS musí podporovat OpenGL ES alespoň ve verzi 2.0. Funkcionalita, kterou poskytuje, se dá rozdělit do následujících kategorií [3]:

- **Načítání textur** z různých zdrojů. Kromě obvyklého synchronního chování je k dispozici i asynchronní načtení.
- **Matematická knihovna** poskytuje běžné optimalizované operace s maticemi, vektory a quaterniony.
- **Efekty** implementují základní sadu shaderů aplikovatelných na renderované vrcholy.
- **Komponenty Views a ViewControllers** redukují potřebné množství kódu k napsání OpenGL ES aplikace, přičemž View slouží jako panel, do kterého probíhá rendering, ViewController pak řídí i jeho průběh.

Pro multiplatformní využití většina uvedených možností nemá význam, neboť jsou pevně svázány s iOS, případně Mac OS X. Komponenty `GLKView` nebo `GLKViewController` však mohou při vývoji pomoci právě v platformně závislých částech kódu, kdy kreslicí smyčku, velikost pohledu a další ponecháme k řešení frameworku místo vlastní implementace.

5 Architektura grafického klienta

V tuto chvíli se dostáváme k samotné realizaci grafického klienta. Postupně vystavíme aplikaci tak, aby byla snadno přenositelná na více platforem a zároveň rozšiřitelná o další funkčnost. Využijeme podmíněné kompilace jazyka C++, díky které snadno volíme aktivní části kódu pomocí maker prekompileru, přičemž právě tyto části mohou obsahovat (a případně skrýt) platformně závislou funkcionalitu. Jednou z cest pro udržitelnost kódu je využití vlastností objektově orientovaného programování (objektů, jejich dědičnosti a zapouzdření) a příslušných návrhových vzorů.

Jádro enginu řídí průběh vykreslování a stará se o systémové úlohy s ním související. V naší implementaci spravuje i grafické zdroje (buffery, textury apod.) a je dále rozšiřitelné pomocí virtuálních událostních metod. Podívejme se nyní na techniky a třídy, které dohromady engine tvoří.

5.1 Platformní nezávislost

Engine využívá jednak knihovní funkce jazyka (u kterých musíme zkontrolovat jejich přenositelnost), ale i systémová volání daného operačního systému. Ta bývají značně odlišná, a proto bychom je měli oddělit již v nejspodnější vrstvě enginu, abychom nadále pracovali v jednotném prostředí. V době psaní této práce jsme cílili na dvě platformy: Windows a iOS. Způsobů, jak zajistit zprovoznění libovolné aplikace na různých zařízeních, se nabízí samozřejmě více.

Prvním, naivním řešením, je vývoj pro první platformu a následný přepis závislých částí při portování. Tento postup bychom mohli očekávat u již vyvinuté aplikace, která se definitivně stěhuje na nové zařízení. Jako nevýhoda se ukazuje rozdvojení vývoje, chceme-li stále podporovat původní platformu.

Druhou možností je vložení již zmíněné pomocné vrstvy, která sjednocuje více rozhraní operačních systémů. Tato vrstva může být buďto staticky linkovanou knihovnou rozdílnou pro každou platformu, nebo se vyskytuje přímo v kódu. V jazyce C++ jsou pak tyto části realizovány pomocí maker (viz ukázka 5.1 – implementace metody mutexu).

Jedná-li se o rozsáhlé třídy nebo jejich skupinu, přichází v úvahu i vyčlenění celého zdrojového souboru z kompilace. To zrealizujeme snadno změnou `makefile` nebo nastavením projektu v používaném IDE. Při objektovém přístupu můžeme rovněž vytvořit společného abstraktního předka, který slouží jako šablona pro konkrétní implementace.

```
1  #if defined(__IPHONE_5_0)
2      pthread_mutex_unlock(&mutex_POSIX);
3  #elif defined(_WINxx)
4      ReleaseMutex(mutex_WIN);
5  #endif
```

Listing 5.1: Ukázka multiplatformního kódu

5.2 Společný předek Generic

Pro třídy enginu navrhneme společného předka, abychom mohli případně rozšířit všechny najednou pouze triviální úpravou v něm samotném. To ale není zdaleka jediný účel třídy `Generic`. S její pomocí můžeme programátora donutit, aby se držel námi definovaného návrhového vzoru.

`Generic` staví na koncepci vlastnictví – každý objekt (kromě kořenového) má povinně svého vlastníka, který je povinným parametrem konstruktoru. Vzniká hierarchie (strom), ve které žádný prvek nesmí zaniknout před objektem, který vlastní (objekty jsou tedy mazány od listů směrem zpět ke kořenu). Objekty si ve většině případů ukládají pouze ukazatel na svého vlastníka, ukazatele na vlastněné uchováváme jen pro specifické účely (bude rozebráno později u objektů, kterých se tato vlastnost týká).

Ve vzniklé hierarchii je možné celkem jednoduše zasílat zprávy přes vlastníky až ke kořenu. `Generic` obsahuje dvě metody, které tuto funkcionalitu zajišťují: `ProcessMessage` a virtuální `ProcessMessageEx`. `ProcessMessage` má jeden povinný parametr (identifikační číslo zprávy) a tři nepovinné (přídavné informace). Pokud v daném objektu není zpráva rozpoznána, je předána svému vlastníkovi. Rozpoznání a zpracování zpráv implementujeme v potomcích právě uvnitř virtuální metody `ProcessMessageEx`. Zasílání zpráv je jedním z často využívaných prvků enginu –

výhodou (navzdory mírné režii, která ho provází) zůstává, že namísto předávání mnoha proměnných vznikajícímu objektu si o ně zažádá sám.

Společný předek `Generic` není využíván u triviálních objektů, které by pomocnou funkcionalitu nevyužily a měly by značný vliv na výkon enginu (např. u matic a mutexů).

5.3 Systémové a řídicí třídy

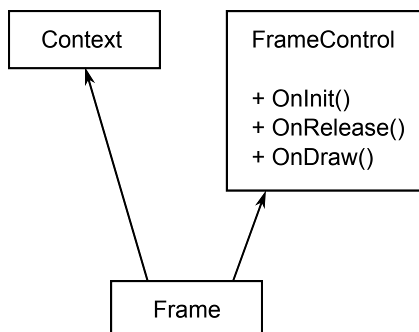
Třídy `Context`, `Frame` zajišťují správu systémových zdrojů – kontextu OpenGL a oblasti vykreslování, což v případě Windows znamená okno, pro iOS pak komponentu GUI roztaženou na celou plochu displeje. Zastřešují tedy platformně specifická nastavení objektem společným navenek.

`Context` je pevně vlastněn objektem typu `Frame`. Implementace vytvoření kontextu OpenGL je rozdělena pro každou platformu zvlášť (`CreateContextWindows` a `CreateContextIOS`), kompiluje se opět jen blok kódu příslušící cílové platformě. To samé platí i pro metody starající se o uvolnění. Zatímco na platformě iOS řeší třída `Context` spíše propojení enginu s frameworkem `GLKit`, ve Windows odpovídá přímo za správu zdrojů týkajících se kontektu. Programátorovi je tak k dispozici i volání `SwapBuffers`, která na platformě Windows zajišťuje manuální záměnu back a front bufferu.

O správu okna (v podobě příslušící zvolené platformě) se stará abstraktní třída `Frame`. Zde byl platformně závislý kód pro lepší čitelnost přesunut do potomků `WindowsFrame` a `IOSFrame`, neboť kromě funkcí či metod, které jsou volány, se liší i celkové pojetí. Opět platí, že ve Windows řídí a spravuje objekt všechny zdroje v kódu sám, ale v iOS figuruje jako prostředník mezi enginem a Cocoa Touch API.

Pro ovládání průběhu vykreslování vznikla třída `FrameControl`, přičemž jejím vlastníkem je přímo objekt typu `Frame`. Ten pak volá událostní metody `FrameControl`, mezi něž patří:

- `OnInit` – metoda, která je volána před prvním vykreslením,
- `OnRelease` – metoda volaná při uvolňování instance `Frame`,



Obrázek 5.2: Diagram vlastnictví tříd.

- `OnDraw` – metoda volaná na začátku každého snímku.

Instance `Frame` získá ukazatel na zvolený `FrameControl` pomocí zprávy s identifikací `MESSAGE_SET_FRAME_CONTROL`, která je zaslána v jeho konstruktoru (uspořádání tříd viz obrázek 5.2). Děděním třídy můžeme překrýt nebo rozšířit událostní metody a vytvořit tak nový průběh kreslení snímku.

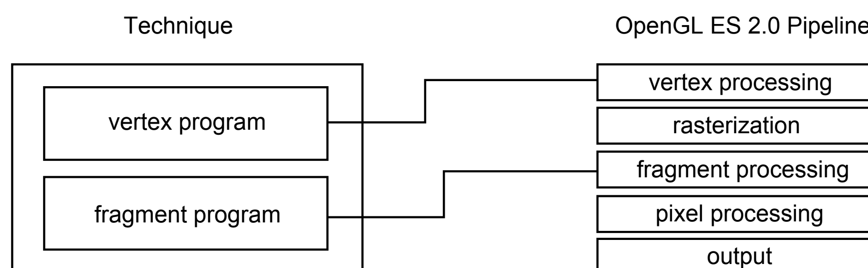
5.4 Grafické zdroje

Přesuňme se k třídám vyšší úrovně, které obalí API OpenGL a OpenGL ES a skryjí tak drobné implementační rozdíly. Pro rendering musíme spravovat techniku kreslení (využíváme vertex a fragment shader), trojúhelníkové meshe (vertex array, element array) a materiál na ně aplikovaný (textura).

5.4.1 Technika kreslení

OpenGL ES 2.0 vyžaduje definici vertex a fragment programu použitých přímo uvnitř renderovací pipeline. Programovatelné prvky jsou sdruženy uvnitř třídy `Technique` (vztah mezi pipeline a třídou znázorněn na obrázku 5.3), která poskytuje i několik pomocných metod, z nichž nás nejvíce zajímají tři veřejné. Metoda `Load` načte z daných umístění textové soubory obsahující zdrojový kód v jazyce GLSL,

provede kompilaci, linking a v případě chyby (ladíme-li program) vypíše log překladu do konzole. `Apply` aktivuje program uložený uvnitř dané instance. Pokud z nějakých důvodů potřebujeme přistoupit ke grafickému programu přímo, k dispozici je i getter `GetProgramId` vracející jeho identifikátor.



Obrázek 5.3: Vztah mezi třídou `Technique` a pipeline OpenGL ES.

Pro překlad techniky slouží privátní metody `CompileShader` a `LinkProgram`. `CompileShader` je univerzální pro vertex i fragment shader, což odpovídá rozhraní OpenGL, kde se mezi nimi rozlišuje pouze pomocí příznaku, který předáme funkci `glCreateShader`. `LinkProgram` se pak stará o jejich propojení ve výsledný grafický program.

5.4.2 Trojúhelníkové meshe

Po implementaci způsobu kreslení se zaměříme na obsah – trojúhelníkové meshe, tedy objekty složené z trojúhelníků. Za tímto účelem existuje v enginu třída `Mesh`, která spravuje potřebné objekty OpenGL API: vertex buffer a element array buffer. Zatímco ve vertex bufferu jsou uloženy všechny vrcholy objektu, element array buffer obsahuje informace o jejich použití v trojúhelnících – jeden vrchol může být využit pro více ploch.

O naplnění bufferů se stará veřejně přístupná metoda `Load`, která si za parametry bere počet vrcholů a odkazujících indexů tvořících trojúhelníky a dále pole samotných vrcholů a indexů. Vrchol definujeme pozicí, normálovým vektorem a texturovacími souřadnicemi. Pro testovací účely je vhodné mít možnost snadno vytvořit různé generované objekty, a proto byly naimplementovány i statické metody `CreateCube` (vytvoření meshe krychle) a `CreateCylinder` (vytvoření meshe válce). Způsob jejich

fungování je jednoduchý: uvnitř vytvoříme novou instanci třídy `Mesh`, vygenerujeme pro ni pole vrcholů a indexů, která předáme metodě `Load`, a vzniklou instanci vrátíme jako výsledek.

Na trojúhelníkové meshe máme možnost i aplikovat materiál. Ten definuje chování pro jednotlivé složky osvětlení (ambient, diffuse, specular). Každá složka je reprezentována hodnotami $(r,\ g,\ b,\ a)$ v rozsahu od nuly do jedné. To umožňuje měnit barvu objektu pouze v globálním měřítku, proto byla přidána i možnost mapování textury.

Vykreslení meshe provedeme voláním `Draw`. Tato metoda nikterak nezasahuje do techniky kreslení, pouze připojí texturu materiálu, je-li přítomna, a zavolá funkci OpenGL API `glDrawElements`. V případě, že mesh není inicializován, neprovádí metoda nic.

5.4.3 Textury

Textura představuje bitmapu mapovanou na povrch objektu. Ten pak působí detailněji bez potřeby zvyšování počtu vrcholů nebo jeho dělení na více objektů s různými materiálovými vlastnostmi. Třída `Texture` obaluje funkce OpenGL a OpenGL ES potřebné právě pro manipulaci s texturami.

Třída je pojata skutečně minimalisticky, poskytuje pouze metodu pro načtení (`Load`), uvolnění (`Release`), připojení (`Bind`) a odpojení (statická metoda `Unbind`) textury. Při volání `Load` zadáváme rozměry, formát, typ filtrování a samotná data textury. OpenGL ES nepodporuje konverzi mezi datovými formáty, proto ani tato třída neumožňuje zadat cílový formát textury na grafické kartě, použije se zkrátka formát zdrojových dat. Metoda `Release` uvolňuje pouze objekty rozhraní OpenGL, samotná instance uvolněna není.

Pro testování vznikla statická metoda `CreateCheckerboard`, která vytvoří novou instanci třídy a vygeneruje černobílý šachovnicový vzor do čtvercové textury zadaného rozměru. Tuto instanci pak získáme jako návratovou hodnotu.

5.5 Třídy obsluhující scénu

Shrneme-li předchozí úsilí, máme připravenou komunikaci se systémem, kontrolu kreslení snímku a různé grafické zdroje. Tyto objekty enginu jsou (z velké části) na sobě nezávislé, a proto přichází chvíle, kdy je logicky propojíme třídou, která provádí rendering celé scény a uchovává k tomu účelu potřebná data.

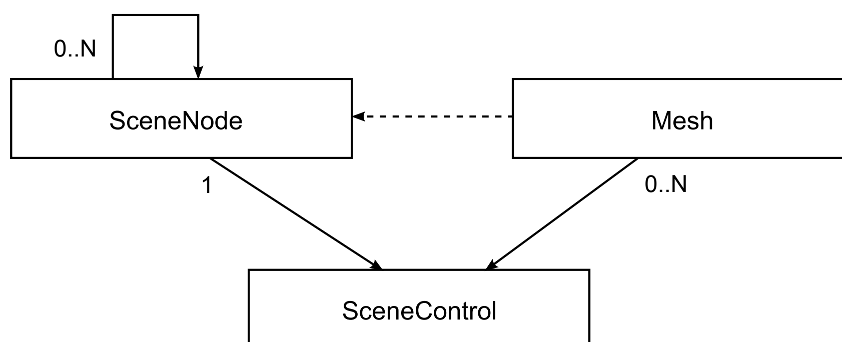
`SceneControl` je potomkem třídy `FrameControl`, přičemž má na starost více úloh – slouží jako databáze grafických objektů i jako výkonný prvek pro jejich rendering. Z toho důvodu byly rozšířeny událostní metody `OnInit`, `OnRelease` a `OnDraw`, které v předkovi neobsahují žádný výkonný kód.

Kromě primitivních objektů, jako jsou techniky kreslení či nastavená barva pozadí, obsahuje třída množinu grafických zdrojů, kdy každý z nich je identifikován unikátním řetězcem. K našemu účelu jsme zvolili samovyvažovací binární strom, konkrétně jeho implementaci ve standardní knihovně jazyka C++: třídu `std::map`. V těchto strukturách ukládáme meshe, textury, ale i uzly scény.

Instance třídy `SceneNode` odpovídá jednomu uzlu grafu scény. Tento uzel, jak již bylo popsáno na str. 7, obsahuje svoji transformační matici a je ovlivněn transformacemi nadřazených uzlů. Hierarchie je zrealizována opět pomocí vlastnictví, objekt nicméně po svém vytvoření zasílá ještě zprávu pro své zaregistrování ve vlastníkovi, jinak bychom vzniklý graf nebyli schopni procházet (existovaly by pouze hrany ve směru ke kořenu). Každá instance si tedy ukládá opět množinu podřízených uzlů, které vlastní. Vykreslení celého podstromu od zvoleného uzlu pak provedeme pomocí metody `Draw`, přičemž kořen je uložen v instanci `SceneControl`.

Každému uzlu můžeme přiřadit mesh, který se pak bude kreslit ovlivněný příslušnými transformacemi. Jeden mesh může být přiřazen více uzlům scény (opačně nikoliv) voláním metody `SetMesh`, která jako parametr vyžaduje řetězec, pod nímž je uložen uvnitř instance třídy `SceneControl`. Zadáme-li prázdný řetězec či neexistující název, pouze odstraníme odkaz. Vyhledání meshe je založeno na vyslání zprávy, kterou vyhodnotí `SceneControl`, nalezne odpovídající záznam a vrátí ukazatel zpět odesílateli zprávy. Obecné vztahy mezi třídami viz obrázek 5.4.

Při průchodu grafem scény musíme udržovat stav aktuálních transformací a další údaje pro rendering. Zatímco v OpenGL ES 1.0 jsme mohli tuto činnost přenechat



Obrázek 5.4: Třídy `SceneControl`, `SceneNode` a `Mesh`.

funkcím rozhraní (`glTranslate`, `glPushMatrix` atd.), verze 2.0 žádné obdobné nenabízí, a tak vznikla třída `InstanceParams`, která je určitým způsobem napodobuje. Třída obsahuje zaprvé stálé hodnoty, které se týkají celého renderingu (např. adresy proměnných shaderů) – tedy v případě, že používáme stále stejnou techniku kreslení. Zadruhé zde nacházíme zásobník transformačních matic používaný metodami `Push` a `Pop` a aktuální (akumulující se) transformace.

Ačkoliv jsme nyní již schopni renderovat celou scénu, rozšíříme engine ještě o jednu třídu, která představuje kameru. Třída `Camera` uchovává svoji projekční matici, přičemž jejím vlastníkem je vybraný uzel scény, který tak určuje její pozici a orientaci. Vztah pro nalezení matice pohledu již byl rozebrán na str. 7, nyní potřebujeme jen získat všechny transformační matice uzlů ležících na cestě od kořenu scény k dané kameře. Opět využijeme mechanismu zasílání zpráv, kdy každý uzel zapíše do bufferu svoji transformaci a předá tutéž zprávu svému vlastníkovi. Teprve třída `SceneNode` označí zprávu jako vyřešenou a odesílatel (kamera) převezme buffer s transformacemi.

6 Rozhraní grafického klienta

Třída `SceneControl` se v tuto chvíli stala autonomně funkčním celkem, který sice řídí průběh vykreslování scény a stará se o příslušné uložené zdroje, nicméně neexistuje prozatím žádný komunikační kanál, který by proces mohl zvenčí ovlivnit. Doplníme tedy třídu o metody, které poskytnou dohromady rozhraní ovládající scénu. Předpokládáme, že jsou buď volány z vlákna, ve kterém byl vytvořen kontext OpenGL, nebo z vlákna, jemuž je kontext nasdílen. V takovém případě ošetřujeme přístup ke sdíleným proměnným pomocí mutexů.

6.1 Meshe a textury

Grafickými zdroji, které spravujeme skrze rozhraní scény, jsou meshe a textury, přičemž platí, že textury lze přiřadit materiálům více meshů. Techniky kreslení (jako zdroj) zpřístupněny nebyly, engine je používá pouze vnitřně.

Uložení meshe či textury provádíme pomocí operace `SetMesh` a `SetTexture`. Prvním parametrem je jedinečný řetězcový identifikátor objektu (jmenný prostor textur a meshů není sdílen), druhým pak ukazatel na již vytvořenou instanci grafického zdroje. Při operaci je převzato vlastnictví objektu a třída `SceneControl` tak zodpovídá za jeho uvolnění. Nulový ukazatel (`NULL`) je chápán jako žádost o manuální uvolnění příslušného uloženého objektu.

Čtení provádíme analogicky k zápisu metodami `GetMesh` a `GetTexture`. Opět jako parametr předáváme identifikující řetězec a instanci hledaného objektu získáme jako návratovou hodnotu. Není-li pod zadaným jménem uložen žádný zdroj, je vrácena hodnota `NULL`.

6.2 Uzly scény a kamery

Manipulace s uzly scény je poněkud odlišná. Jelikož uzel již ve svém konstruktoru zasílá svému vlastníkovi zprávu se žádostí o registraci, není potřebné posky-

tovat operaci `set`, ale právě vyhledání onoho vlastníka. K tomu slouží metody `FindRootSceneNode` a `FindSceneNode`. První zmíněná vrátí ukazatel na kořenový uzel scény, který doporučujeme nemodifikovat – měl by sloužit pouze jako neměnný vlastník podřízených uzlů. `FindSceneNode` poskytne ukazatel na uzel scény odpovídající předanému identifikujícímu názvu, a pokud není nalezen, návratovou hodnotou je `NULL`.

Vytvoření uzlu scény pak spočívá ve volání konstruktoru třídy `SceneNode`, kdy jako vlastníka objektu předáváme vybraný nadřazený uzel scény. Konstruktor má však ještě druhý povinný parametr, a to jméno vytvořeného uzlu. Odstranění provedeme voláním `delete`, přičemž mechanismus zpráv zajistí zrušení registrace v nadřazeném uzlu a správci scény.

Kamery vytváříme opět pomocí jejich konstruktoru, jako vlastníka volíme vybraný uzel scény, jehož transformace ovlivňuje matici pohledu. Kamery jako takové nenesou informaci o svém jméně, a proto třída `SceneControl` poskytuje metody `SetCamera` a `GetCamera`, kdy příslušnému řetězci přířadíme ukazatel na danou kameru (odpovídá již zmíněné logice uchovávání meshů a textur).

Mezi kamerami přepínáme operací `SetActiveCamera`, i tentokrát předáváme identifikační řetězec. Chceme-li zjistit, která kamera je momentálně používána pro rendering, použijeme metodu `GetActiveCamera`.

6.3 Ostatní operace rozhraní

S pomocí rozhraní lze nastavit barvu pozadí. To provedeme voláním metody `SetBackground`, jejíž paremetry jsou složky $RGBA$ v rozsahu od nuly do jedné. Můžeme rovněž vyhledat nepoužité meshe metodou `FindUnusedMeshes`, případně je rovnou odstranit pomocí `DeleteAllUnusedMeshes`.

Poskytován je rovněž picking objektů ve scéně. Zavoláme metodu `Pick`, jako parametry předáme souřadnice kurzoru v rozsahu odpovídajícímu rozlišení displeje a referenci na řetězec, do kterého chceme uložit jméno vybraného uzlu scény. Pokud proběhl picking úspěšně (našli jsme objekt pod kurzorem), metoda vrací booleovskou hodnotu `true` a dodá název vybraného uzlu. Picking byl implementován metodou

kódování objektů scény různou barvou a následným čtením pixelu z framebufferu (popis techniky viz kapitola 3.4.2). Pokud bychom vyvíjeli klienta nad OpenGL ES 3.0, který podporuje rendering do více framebufferů zároveň, mohli bychom picking dále optimalizovat. Spolu s vykreslením scény na obrazovku by se zároveň do druhého připojeného framebufferu zaznamenaly identifikační barvy objektů, a tak by metoda `Pick` prováděla už jen zpětné čtení a identifikaci. V případě OpenGL ES 2.0 nezbývá než scénu renderovat ve dvou průchodech.

7 Ukázkový protokol pro ovládání klienta

Samotné rozhraní pro manipulaci s objekty vyskytujími se ve scéně pokryje pouze přímá volání z aplikace, která engine přímo využívá (ten je k ní staticky či dynamicky připojen jako knihovna). Zavedeme-li textový protokol, který je následně přeložen aplikací na odpovídající volání metody enginu, získáme možnost ovládat scénu například vstupem ze souboru, interaktivně psaním příkazů do konzole nebo přijímáním zpráv přes síť'.

7.1 Příkazy protokolu

V době psaní této práce nebyla dána přesná forma protokolu, který by měl být klientem využit. Z tohoto důvodu jsem přikročil k navržení vlastního jednoduchého protokolu, který dostatečně demonstruje ovládání prvků scény. Zvolil jsem imperativní paradigma (protokol se skládá z příkazů).

Všechny objekty jsou identifikovány na základě svého jednoznačného jména (řetězce znaků). Následuje vysvětlení jednotlivých příkazů protokolu, hranaté závorky značí proměnnou, kulaté závorky nepovinnou část příkazu, svislá čára výběr z několika možností. Příkazy jsou ukončeny středníkem a mohou pokračovat přes více řádek, na jednom se jich však nesmí nacházet více než jeden.

7.1.1 Správa meshů

Níže popsané příkazy zacházejí s trojúhelníkovými modely (meshes), které slouží jako zdroje pro kreslení v uzlech scény.

`mesh [name] generate cube [a];`
Vygenerování krychle o délce hrany a a její uložení pod názvem *name*.

`mesh [name] generate cylinder [r] [h];`
Vygenerování válce o poloměru r, výšce h a jeho uložení pod názvem *name*.

`mesh [name] load [data];`
Načtení trojúhelníkového meshe a jeho uložení pod názvem *name*, proměnná *data* představuje pole vrcholů zapsaných postupně jako souřadnice vrcholů a normál.

`mesh [name] material ambient|diffuse|specular [r] [g] [b] [a];`
Nastavení vybrané složky materiálu meshe *name* na barvu *(r, g, b, a)*.

`mesh [name] delete;`
Odstranění meshe *name*.

7.1.2 Správa uzlů scény

Příkazy spravující uzly scény zařizují umístění instancí meshů ve scéně, změnu jejich transformace nebo jejich odstranění.

`node [name] create ([parent]);`
Vytvoření uzlu scény a jeho uložení pod názvem *name*. Nepovinný řetězec *parent* určuje název nadřazeného uzlu, není-li přítomen, nadřazeným se stává kořen hierarchie scény.

`node [name] mesh [meshname];`
Přiřazení meshe uloženého pod názvem *meshname* uzlu scény *name*.

`node [name] identity;`
Nastavení implicitní (jednotkové) transformační matice uzlu *name*.

```
node [name] multmatrix
[a11] [a12] [a13] [a14]
[a21] [a22] [a23] [a24]
[a31] [a32] [a33] [a24]
[a41] [a42] [a43] [a44];
```

Násobení transformační matice uzlu *name* zadanou maticí čísel $a_{11} \ldots a_{44}$.

`node [name] translate [x] [y] [z];`
Násobení transformační matice uzlu *name* maticí translace definovanou vektorem (x, y, z).

`node [name] rotate x|y|z [angle];`
Násobení transformační matice uzlu *name* maticí rotace definovanou osou a úhlem otočení *angle* zadaném ve stupních.

`node [name] scale [x] [y] [z];`
Násobení transformační matice uzlu *name* maticí škálování definovanou vektorem (x, y, z).

`node [name] delete;`
Odstranění uzlu scény uloženého pod názvem *name*.

7.1.3 Správa kamery

Popišme nyní příkazy zacházející s kamerami. Na rozdíl od implementovaného rozhraní pro manipulaci se scénou nebyly do protokolu zařazeny všechny operace (např. nastavení projekční matice kamery), neboť slouží pouze pro základní demonstrační účely.

`camera [name] create [nodename];`
Vytvoření kamery a její umístění do uzlu nazvaného *nodename*, následně je uložena pod jménem *name*.

`camera [name] activate;`
Aktivování kamery nazvané *name* a její následovné používání pro rendering scény.

`camera [name] delete;`
Smazání kamery nazvané *name*, byla-li aktivní, engine přepíná na defaultní kameru.

7.1.4 Dávka příkazů

Dávka příkazů slouží k jejich načtení ze souboru a následnému provedení. Uživatel si je tak může opětovně vyvolat při libovolném běhu programu bez nutnosti psát celý skript znovu, což lze využít například pro načítání meshů z rozsáhlých souborů (samozřejmě obsahem dodržující definovaný protokol).

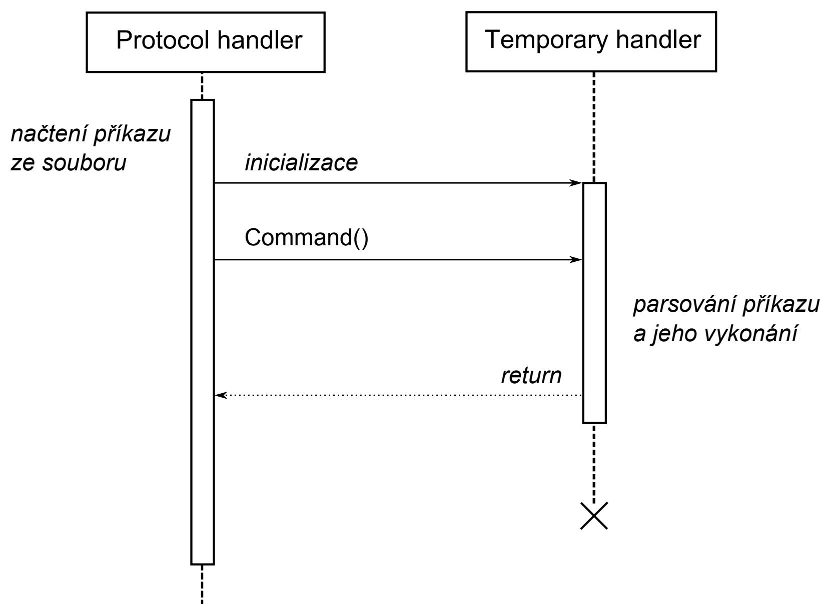
`batch [filename];`
Spuštění skriptu ze souboru nazvaného *filename*, jméno nesmí obsahovat bílé znaky. Vnořené volání skriptů je povoleno, avšak nekonečná rekurze není ošetřena.

7.2 Interpretující třída

V klientské aplikaci překlad protokolu na volání metod rozhraní scény provádí třída `DemoProtocolHandler`. Poskytuje navenek pouze konstruktor a veřejnou metodu `Command`. V konstruktoru se instance propojí s rozhraním scény (`SceneControl`) a inicializuje proměnné instance. Metodu `Command` voláme s parametrem `cmd`, což je klasický řetězec terminovaný nulovým znakem. Nazpět získáme ukazatel na řetězec obsahující chybovou hlášku, který je nulový v případě, že příkaz proběhl v pořádku.

Příkaz je parsován uvnitř objektu za pomoci standardní knihovny jazyka C, na základě prvního přečteného slova se kód větví do privátních metod obsluhujících již konkrétní kategorie příkazů (meshe, uzly scény atd.). Při první nalezené chybě provádění končí a do privátní proměnné `lastError`, na kterou je volajícímu následně vrácen ukazatel, je zapsáno chybové hlášení.

Poněkud speciální úlohu plní příkaz `batch`, při němž čteme zdrojový soubor obsahující množinu příkazů. Načítáme postupně řádek po řádce a plníme textový buffer, než nenalezneme ukončující znak (středník). Protože potřebujeme zachovat stav interpretující instance, vytvoříme dočasně novou, která příkaz zpracuje (voláním její metody `Command`). Situace je znázorněna na obrázku 7.1. Pokud se v dávkovém souboru vyskytne opět `batch`, proces bude totožný s výše uvedeným.



Obrázek 7.1: Diagram zpracování příkazu `batch`.

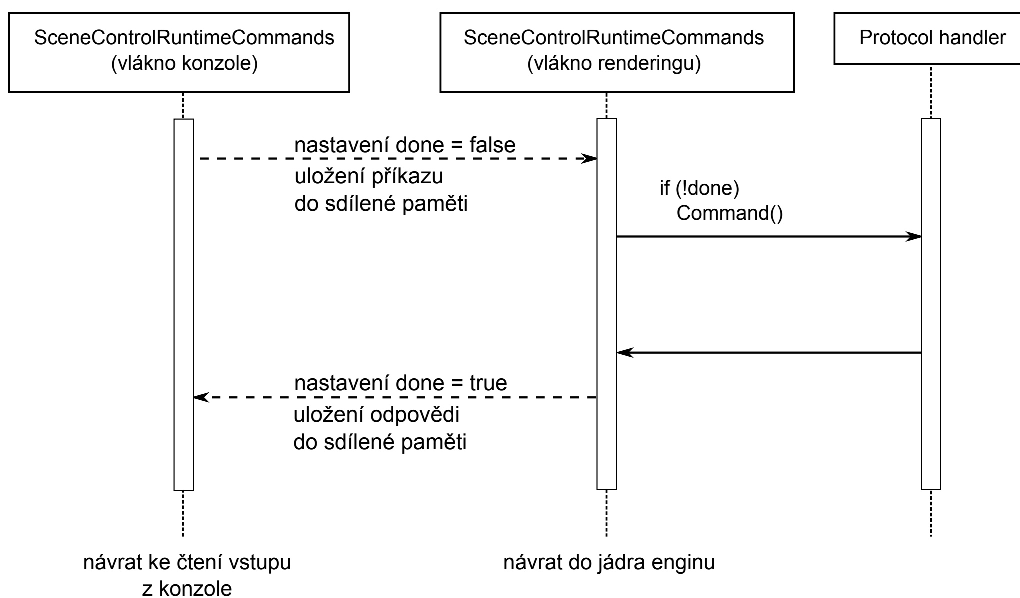
7.3 Interaktivní zadávání příkazů ve Windows

Pro demonstrační účely bylo rovněž vhodné doprogramovat konzoli, do které zadáváme jednotlivé příkazy protokolu a jejich vliv okamžitě pozorujeme v renderovacím okně. Toto rozšíření bylo realizováno pouze na platformě Windows, kde má uživatel k dispozici klávesnici a možnost uspořádání oken aplikací – na rozdíl od iOS.

Konzoli vytvoříme v samostatném vlákně za pomoci volání funkce rozhraní `WinAPI` `AllocConsole` a provedeme přesměrování standardního vstupu a výstupu. Načítáme vstup do bufferu do chvíle, než přijde ukončovací znak (středník). V tuto chvíli vlákno vyšle požadavek na vykonání příkazu vláknu, ve kterém běží engine. Uvědomme si, že metody rozhraní scény nesmíme volat z jiného, protože kontext OpenGL nesdílíme napříč vlákny. Hlavní vlákno musíme proto uzpůsobit tak, aby dokázalo příkazy přijímat. Zdědíme tedy `SceneControl` do nové třídy `SceneControlRuntimeCommands`, přičemž rozšiřujeme metodu `OnDraw` a nabízíme novou: `Command`.

`Command` je blokující metoda volaná z vlákna konzole – její úlohou je uložit příkaz uvnitř instance `SceneControlRuntimeCommands`, čekat na jeho dokončení a vrátit výsledek (případné chybové hlášení). Virtuální metoda `OnDraw`, která je

volána předkem z vlákna enginu, kontroluje, jestli v instanci není připraven příkaz k vykonání. Pokud ano, je proveden a výsledek opět uložíme uvnitř instance, odkud si ho přečte konzolové vlákno. K sdíleným proměnným je přístup ošetřen pomocí mutexů, metoda `Command` čeká aktivně ve smyčce na nastavení proměnné `done` na hodnotu `true`. Komunikace mezi vlákny je přiblížena na obrázku 7.2.



Obrázek 7.2: Komunikace mezi vlákny (konzolí a správcem scény).

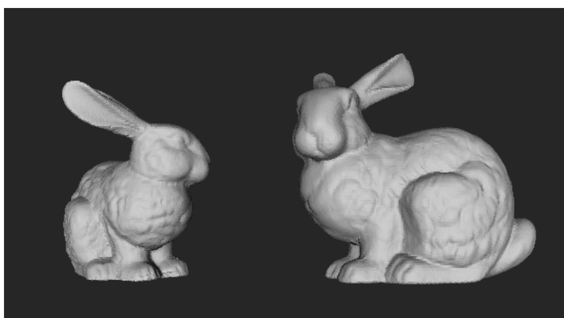
8 Ukázkové scény

Pro otestování správné funkčnosti grafického klienta jsme připravili několik testovacích scén, na kterých byly doladěny případné nedostatky. Využili jsme příkazu `batch`, abychom mohli načítat scénu v nativně podporovaném ukázkovém protokolu. Scény postupně testují různě obtížné situace pro zpracování, od velmi jednoduché scény až po načítání relativně složitých meshů.

8.1 Použité meshe

Jako testovací meshe byly zvoleny dva modely. Prvním je známý model *Utah Teapot* (viz obr. 8.1), druhým *Stanford Bunny* (viz obr. 8.2), který se skládá již ze značně vyššího počtu trojúhelníků.

Obrázek 8.1: Model Utah Teapot.



Obrázek 8.2: Model Stanford Bunny z více úhlů pohledu.

Tyto modely jsou standardně k dispozici ve formátu *Wavefront OBJ*[6] s koncovkou souboru `OBJ` či *Polygon File Format*[7] s koncovkou `PLY`. Ani s jedním formátem

neumí náš grafický klient pracovat, a proto musela být provedena jednoduchá konverze do formátu odpovídajícímu protokolu. Zkonvertovaný mesh je pak možné načíst pomocí příkazu `batch`.

8.2 Benchmark

Zaměřme se nyní na výkon aplikace. Na čtyřech ukázkových scénách (screenshoty v příloze) byly měřeny časy načtení a renderu jednoho snímku. Do načtení je zahrnuta interpretace protokolu včetně I/O operací a vytváření objektů OpenGL. Je třeba brát v potaz, že uvedená měření se vztahují k jedné hardwarové konfiguraci (Intel Core i5-3350P 3.10Ghz, 8GB RAM, Radeon HD 7750) a byla vícekrát opakována pro poskytnutí průměrných výsledků. Vzhledem k tomu, že pro iOS nebyl protokol již testován, hodnoty jsme získali pouze na platformě Windows. Při testování byla samozřejmě vypnuta vertikální synchronizace.

Obsah scén se různí nejen počtem kreslených primitiv. První scéna se skládá pouze z krychle, na kterou je aplikován materiál. Druhá obsahuje dvě instance komplexnějšího modelu *Stanford Bunny*, třetí jednu instanci modelu *Utah Teapot* s několika válci. Poslední scéna spíše demonstruje sílu vzájemných závislostí uzlů (renderují se pouze krychle), přičemž žádná výrazná režie se do výsledku nepromítla.

Výsledky měření jsou zaznamenány v tabulce 8.1. Rendering při vypnuté vertikální synchronizaci plně vytížil grafickou kartu i procesor (respektive jedno jádro).

Scéna	prázdná	Demo 01	Demo 02	Demo 03	Demo 04	Demo 05
Počet instancí	0	1	2	3	11	10000
Počet trojúhelníků	0	12	138902	15960	132	120000
Doba kreslení jednoho snímku [ms]	0,115	0,131	0,647	0,237	0,131	16,5
Doba načtení [ms]	—	0,65	920	217	0,92	179

Tabulka 8.1: Výsledky měření výkonu aplikace.

9 Závěr

Práce si kladla za cíl vytvořit tenkého grafického klienta, který je schopný pracovat na mobilních zařízeních a dokáže přijímat požadavky na rendering skrze své rozhraní nebo definovaný protokol. Vzhledem k vrstvené architektuře s jasně definovaným rozhraním nečiní problém doprogramovat jiný vstup, například nový, robustnější komunikační protokol založený na X3DOM či JSON syntaxi.

Grafický klient splňuje v základní podobě kompatibilitu s prostředím návrhu scény v programu GraphWorX softwarové sady GENESIS. Předpokládáme, že vytvořené scény lze snadno pomocí naší aplikace zobrazit.

Testování výkonu aplikace poskytlo uspokojivé výsledky, neboť i na mnohonásobně pomalejším než měřeném hardwaru by obraz působil plynule (více než třicet snímků za vteřinu). Při měření nebyly zpozorovány žádné větší odchylky výkonu, které by uživatel mohl vnímat jako nepříjemná zaseknutí obrazu.

Na vytvořený engine a aplikaci je možné plynule navázat tvorbou další vrstvy řídící animace a komunikaci se serverem, případně rozšířit funkcionalitu jádra enginu nejenom v grafické oblasti. V úvahu připadá rozšíření logování, podpora složitějších (například průhledných) materiálů a implementace multiplatformních socketů pro síťovou komunikaci. Další platformou, na níž by se klient mohl vyskytovat po dopsání platformně závislých částí zdrojového kódu, je například OS Android.

Klient ve své současné podobě nemusí sloužit pouze pro průmyslové vizualizace, ale drobnými úpravami bychom mohli docílit jednoduchého jádra například pro vykreslování her. Vzhledem ke zvoleným technonologiím a jazyku není další vývoj enginu významně omezen.

Literatura

[1] *iOS Developer Library* [online]. [cit. 17.4.2013]. Dostupné z: http://developer.apple.com/library/ios.

[2] *GENESIS64^{TM}* [online]. [cit. 15.4.2013]. Dostupné z: http://www.iconics.com/Home/Products/HMI-SCADA-Software-Solutions/GENESIS64.aspx.

[3] *Introduction to GLKit* [online]. [cit. 17.4.2013]. Dostupné z: http://developer.apple.com/library/ios/#documentation/GLkit/Reference/GLKit_Collection/Introduction/Introduction.html.

[4] *OpenGL® ES Common Common-Lite Profile Specification* [online]. 2004. [cit. 7.3.2013]. Dostupné z: http://www.khronos.org/registry/gles/specs/1.0/opengles_spec_1_0.pdf.

[5] *OpenGL® ES Common Profile Specification (Version 2.0.25)* [online]. 2010. [cit. 7.3.2013]. Dostupné z: http://www.khronos.org/registry/gles/specs/2.0/es_full_spec_2.0.25.pdf.

[6] *Wavefront OBJ File Format Summary* [online]. 2013. [cit. 3.4.2013]. Dostupné z: http://www.fileformat.info/format/wavefrontobj/egff.htm.

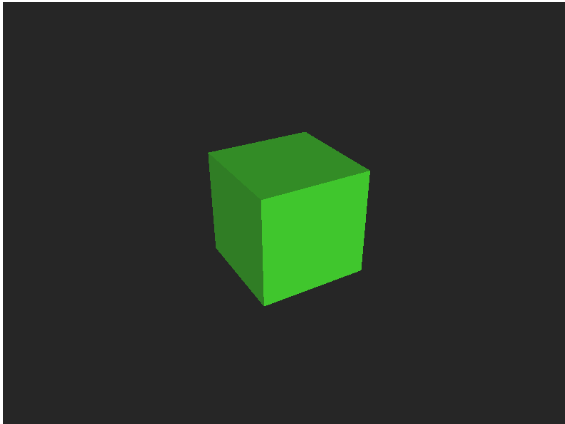
[7] BOURKE, P. *PLY - Polygon File Format* [online]. 2013. [cit. 3.4.2013]. Dostupné z: http://paulbourke.net/dataformats/ply/.

[8] REINERS, D. Scene Graph Rendering. *Proceedings of IEEE Virtual Environments*. 2002.

[9] ZHAO, H. et al. Fast and Reliable Mouse Picking Using Graphics Hardware. *International Journal of Computer Games Technology*. 2009.

A Příloha: Ukázky scén

Scéna Demo 01



```
mesh greencube generate cube 5;
mesh greencube material diffuse 0.2 0.9 0.1;
node node_greencube create;
node node_greencube mesh greencube;
node node_greencube rotate x 30;
node node_greencube rotate y 30;
```

Scéna Demo 02

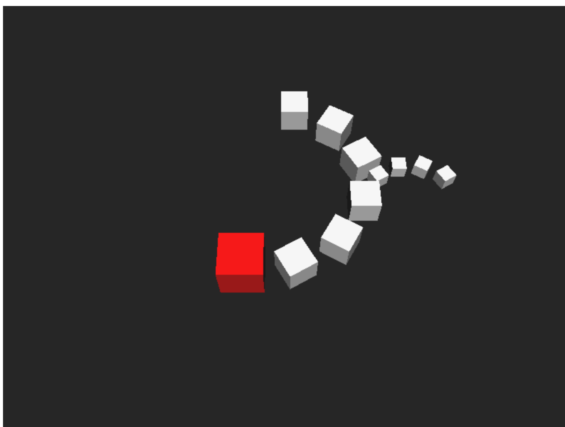
```
batch bunny.model;

node node_bunny create;
node node_bunny mesh bunny;
node node_bunny translate 4 -2 0;

node node_bunny2 create node_bunny;
node node_bunny2 translate -10 0 0;
node node_bunny2 rotate y 130;
node node_bunny2 scale 0.8 0.8 0.8;
node node_bunny2 mesh bunny;
```

Scéna Demo 03

```
batch teapot.model;
mesh teapot material diffuse 0.9 0.6 0.3;
mesh cylinder generate cylinder 5.5 0.5;
mesh cylinder material diffuse 0.4 0.05 0.05;
mesh cylinder material specular 0.4 0.05 0.05;
mesh cylinder2 generate cylinder 4 0.5;
mesh cylinder2 material diffuse 0.1 0.2 0.6;
mesh cylinder material specular 0.4 0.4 0.05;

node node_cyl create;
```

```
node node_cyl mesh cylinder;
node node_cyl translate 0 -2 0;
node node_cyl rotate x 20;
node node_cyl rotate y 165;
node node_cyl scale 2 2 2;

node node_cyl2 create node_cyl;
node node_cyl2 mesh cylinder2;
node node_cyl2 translate 0 0.5 0;

node node_tea create node_cyl;
node node_tea mesh teapot;
node node_tea translate 0 0.3 0;
node node_tea scale 1.7 1.7 1.7;
```

Scéna Demo 04

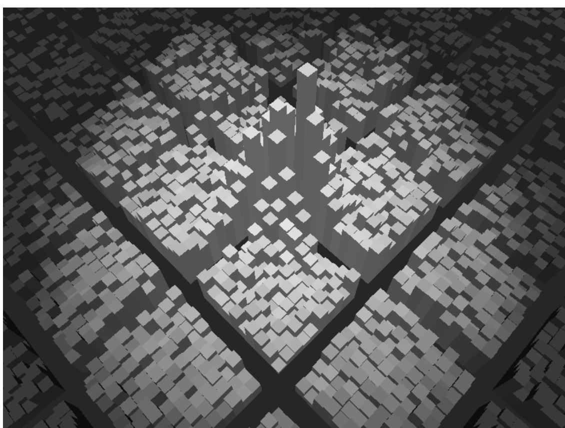


```
mesh rootcube generate cube 1;
mesh rootcube material diffuse 1 0 0;

mesh cube generate cube 0.7;
mesh cube material diffuse 1 1 1;

node n1 create;
```

```
node n1 mesh rootcube;
node n1 translate -2 -2 0;
node n1 rotate x -30;
node n1 scale 2 2 2;

node n2 create n1;
node n2 mesh cube;
node n2 translate 1.2 0 0;
node n2 rotate z 30;

node n3 create n2;
node n3 mesh cube;
node n3 translate 1.2 0 0;
node n3 rotate z 30;

node n4 create n3;
node n4 mesh cube;
node n4 translate 1.2 0 0;
node n4 rotate z 30;

node n5 create n4;
node n5 mesh cube;
node n5 translate 1.2 0 0;
node n5 rotate z 30;

node n6 create n5;
node n6 mesh cube;
node n6 translate 1.2 0 0;
node n6 rotate z 30;

node n7 create n6;
node n7 mesh cube;
node n7 translate 1.2 0 0;
node n7 rotate z 30;
```

```
node p1 create n3;
node p1 mesh cube;
node p1 translate 2 0 0;
node p1 rotate z -30;
node p1 scale 0.5 0.5 0.5;

node p2 create p1;
node p2 mesh cube;
node p2 translate 1.2 0 0;
node p2 rotate z -30;

node p3 create p2;
node p3 mesh cube;
node p3 translate 1.2 0 0;
node p3 rotate z -30;

node p4 create p3;
node p4 mesh cube;
node p4 translate 1.2 0 0;
node p4 rotate z -30;
```

Scéna Demo 05



Výpis příkazů protokolu je pro tuto scénu velmi rozsáhlý, proto je přiložen pouze na CD.

B Příloha: Použité verze shaderů

Jazyky GLSL a GLSL ES se v drobných ohledech různí, avšak při dodržení konkrétních verzí jim lze dodat téměř totožnou podobu. Pro variantu *ES* byla vybrána verze 1.0, pro desktop 1.5.

Vertex shader získává vstup z vertex bufferu přes proměnné označené klíčovým slovem `attribute`. Výsledek předává přes proměnné tentokrát označené jako `varying`. Propojení shaderů zajišťuje linking, který z obou shaderů k sobě přiřadí `varying` proměnné se stejným názvem. Výstup fragment shaderu zajišťuje předpřipravená globální proměnná `gl_FragColor`.

V novějších verzích (např. GLSL ES 3.0) je možné namísto klíčového slova `varying` používat označení `in` pro vstup a `out` pro výstup. Nutnost používání globální proměnné `gl_FragColor` ve fragment shaderu potom odpadá.

C Příloha: Obsah CD

Tato příloha popisuje adresářovou strukturu přiloženého CD.

doc	bakalářská práce ve formátu PDF
images	obrázky uvedené v příloze A
program	grafický klient
binary	přeložené binární soubory
documentation	automaticky generovaná dokumentace
libraries	staticky linkované knihovny
source	zdrojové soubory
visual_studio	soubory projektu Microsoft Visual C++
protocol	scény ukázkového protokolu
tex	zdrojové soubory bakalářské práce