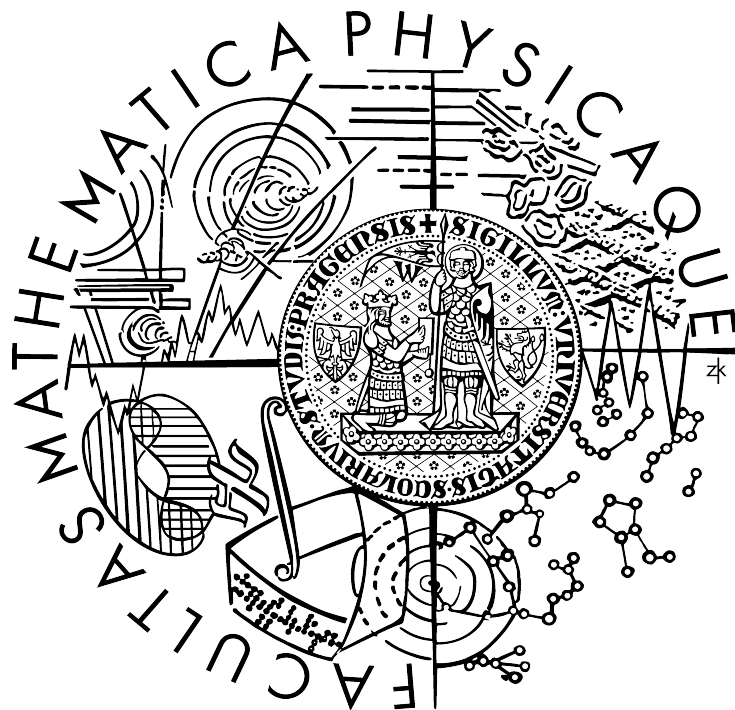


Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Ondřej Hanslík

3D hry v J2ME

Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Jakub Yaghob, Ph.D.
Studijní program: Informatika, Softwarové systémy
2010

Chtěl bych poděkovat firmě **Pixalon studios** za poskytnutí grafiky použité v implementaci enginu, za povolení k uveřejnění náhledů z technických prototypů dosud nevydaných her a za všeobecný náhled do zákulisí distribuce mobilních her.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 1. srpna 2010

Ondřej Hanslík

Obsah

Obsah.....	1
Úvod.....	5
Kapitola 1	
Mobilní telefony jako aplikační platforma.....	7
Kapitola 2	
Java na mobilních zařízeních.....	9
2.1 J2ME a midlety.....	9
2.2 Přenositelnost a fragmentace – když Java přestane být Javou.....	9
2.3 Cyklus kompilace mobilní aplikace.....	12
Kapitola 3	
Mobilní 3D hry.....	15
3.1 Úvod.....	15
3.2 Historické 3D hry.....	15
3.2.1 MDoom.....	15
3.2.2 3D Adventure.....	16
3.2.3 StrangeMaze.....	20
3.2.4 3D Adventure 2 (Green Eye).....	21
3.2.5 Paintball.....	24
3.2.6 Ostatní hry.....	25
3.3 Hratelnost a prodejnost.....	26
3.3.1 Ovládání mobilních her a dopady na hratelnost.....	26
3.3.2 Hratelnost mobilních her a důsledky na prodejnost.....	28
3.4 Optimalizace.....	29
Kapitola 4	
3D knihovny na mobilních telefonech.....	34

4.1 Úvod.....	34
4.2 Mascot Capsule v3.....	34
4.3 Mobile 3D Graphics (M3G).....	35
4.3.1 Požadavky na specifikaci.....	35
4.3.2 Strom scény a picking.....	36
4.3.3 Typy uzlů.....	37
4.3.4 M3G 2.0.....	39
4.3.5 Java binding for OpenGL ES.....	41
4.4 Využití 3D API v 2D hrách.....	41
4.4.1 Motivace.....	41
4.4.2 Vykreslení popředí 2D her pomocí 3D API.....	41
4.4.3 Vykreslení pozadí 2D her pomocí 3D API.....	43
4.4.4 Jiné kombinace 2D a 3D.....	44
Kapitola 5	
Multiplayer.....	46
5.1 Spojení dvou mobilních zařízení.....	46
5.1.1 Internet.....	46
5.1.2 Infraport.....	46
5.1.3 Kabelové spojení.....	47
5.1.4 Bluetooth.....	47
5.2 Tvorba Bluetooth her.....	47
5.3 Vlastnosti Bluetooth API.....	48
5.4 Jiná využití Bluetooth komunikace ve hrách.....	49
Kapitola 6	
Implementace enginu.....	50
6.1 Přístup – požadavky na 3D engine.....	50
6.2 Vytvoření 3D světa.....	50

6.2.1 Editor.....	50
6.2.2 3D svět.....	51
6.2.3 Exportování 3D světa.....	52
6.3 Kolizní systém.....	54
6.3.1 Vysílání paprsků – naivní přístup.....	55
6.3.2 Kolize s trojúhelníky.....	57
6.4 Viditelné objekty, míření, používání objektů.....	60
6.5 Animace světa.....	60
6.6 Multiplayer.....	61
6.7 GUI.....	61
6.8 Základ jednoduché hry.....	63
Kapitola 7	
Závěr.....	64
Zdroje.....	66

Název práce: 3D hry v J2ME
Autor: Ondřej Hanslík
Katedra: Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. Jakub Yaghob, Ph.D.
E-mail vedoucího: yaghob@ksi.mff.cuni.cz

Abstrakt: Cílem textu je uvedení do problematiky vývoje 3D her v J2ME na mobilních zařízeních ve všech stupních vývoje od počátečního návrhu až k distribuci. Podrobně je popsána historie 3D her s důrazem na použité fyzikální a grafické algoritmy. Jsou srovnány dostupné 3D technologie a knihovny. Samostatně jsou probrány možnosti multiplayer her, v závěru s detailním rozebráním technologie Bluetooth. Poslední kapitoly jsou věnovány implementaci enginu pro First Person Shooter hry na knihovně M3G.

Klíčová slova: J2ME, mobilní zařízení, M3G, 3D hry, historie, grafika, kolize

Title: 3D games in J2ME
Author: Ondřej Hanslík
Department: Department of Software Engineering
Supervisor: RNDr. Jakub Yaghob, Ph.D.
Supervisor's e-mail address: yaghob@ksi.mff.cuni.cz

Abstract: Aim of this text is to introduce readers to the difficulties of 3D game development on J2ME mobile platform in all stages from primary design to distribution. History of 3D games is described in detail with emphasis to used physical and graphical algorithms. In successive chapters, accessible 3D technologies and APIs are compared. A separate chapter examines the possibilities of multiplayer games, analyzing Bluetooth technology in detail. Last chapters are dedicated to implementing an engine for First Person Shooter games based on M3G library.

Keywords: J2ME, mobile devices, M3G, 3D games, history, graphics, collisions

Úvod

Ještě před několika lety se programování dělilo na dvě velké oblasti – programování pro desktopové počítače a programování pro servery. Poté, co se objevily mobilní zařízení s podporou nahrávání aplikací, vznikla třetí skupina – programování pro mobilní zařízení.

S dalšími dvěma skupinami má mnoho společných vlastností, ale v průběhu času programátoři zjistili, že k vývoji pro mobilní zařízení nelze přistupovat zcela stejně jako k vývoji pro jiná zařízení. To se týká primárně aplikací náročných na výkon, což jsou na mobilních zařízeních prakticky výhradně hry.

Hlavním cílem této práce je právě analýza výkonnostně nejnáročnější skupiny her a tou je skupina 3D her. Při tom se zaměříme na aktuálně nejrozšířenější mobilní platformu – J2ME.

Prvním dílčím cílem bude zmapování historie 3D her na mobilních telefonech. Zaměříme se převážně na použité algoritmy, jejich slabiny a silné stránky. Druhým cílem bude seznámit čtenáře s problémy portování jedné aplikace na široké spektrum různých zařízení podporujících platformu J2ME. Třetím cílem bude analýza možnosti *multiplayer* her na mobilních zařízeních, s detailnější analýzou technologie Bluetooth. Čtvrtým cílem bude seznámení čtenáře s knihovnou *Mobile 3D Graphics*, kde probereme její základní vlastnosti její vztah k *OpenGL*, její nedostatky a možnosti jejího využití ve hrách. Posledním a největším dílčím cílem bude implementace *first person shooter* hry s využitím knihovny *Mobile 3D Graphics*.

Kromě těchto cílů se však pokusíme čtenáři problematiku mobilních her podat uceleně a popsat vývoj her pro mobilní telefony ve všech ohledech – od přístupu k vývoji až k základům marketingových strategií při jejich distribuci. Budeme mluvit o ovládnutí, o grafické i výpočetní stránce. Nejvíce však budeme řešit problémy přenositelnosti mezi různými zařízeními a styl vývoje, který tyto problémy minimalizuje. Jedním z dalších cílů tohoto dokumentu tedy bude návod, jak přistoupit k vývoji aplikací pro mobilní zařízení a jak se vyhnout základním chybám, které mohou vývoj prodloužit.

V prvních dvou kapitolách probereme základní fakta o mobilních zařízeních, seznámíme se s různými vývojovými platformami a hlavně s odlišnostmi ve stylu vývoje oproti desktopům. Jedná se o základní poznatky nutné k pochopení dalšího textu i všech rozhodnutí, která jsme učinili během implementace.

Dále se, až na výjimky, budeme věnovat pouze části mobilních her, které jsou na vývoj nejnáročnější, a to jsou 3D hry. Nízký výkon mobilních zařízení je totiž výrazně ovlivňuje jak v grafice, tak ve výpočtech. Ve třetí kapitole do detailu probereme základní algoritmy a triky, které používaly starší 3D hry, a ve čtvrté kapitole pak postupně přejdeme k současným hrám. Na nich ukážeme možnosti nejnovějších 3D technologií a jednotlivé technologie mezi sebou srovnáme. Pokusíme se tak čtenáři podat ucelený náhled do vývoje mobilních 3D her.

Celou pátou kapitolu se budeme věnovat možnostem *multiplayer* her na mobilních zařízeních. Probereme vhodnost či nevhodnost jednotlivých přenosových technologií a podrobněji prozkoumáme technologii Bluetooth.

V průběhu všech kapitol se také budeme občas zmiňovat o herní distribuci a marketingu. Ukážeme, že rentabilita vývoje je často jedním z nejdůležitějších hledisek, která brání vytváření technologicky pokročilých her.

V šesté kapitole popíšeme implementaci enginu pro mobilní *first person shooter* hry s využitím dnes nejrozšířenější grafické knihovny. V popisu implementace probereme, jakými způsoby lze k návrhu enginu přistoupit a proč jsme zvolili právě ta řešení, která jsme zvolili. Jednotlivě se budeme věnovat editoru map, fyzikálnímu systému a grafickému systému – vždy s důrazem na problémy, které během vývoje nastaly.

V závěru se už jen velmi stručně zmíníme o možnostech využití našeho herního enginu do budoucna a zhodnotíme splnění cílů práce.

Po přečtení tohoto textu by se měl programátor umět orientovat v problematice vývoje pro J2ME platformu, znát plusy a minusy jednotlivých 3D knihoven a být připraven na potenciální rizika vývoje 3D her pro mobilní zařízení.

Kapitola 1

Mobilní telefony jako aplikační platforma

V posledních osmi letech se mobilní telefony vyvinuly z primitivních komunikačních nástrojů v novou aplikační platformu a v mnoha ohledech začaly konkurovat osobním počítačům, zejména notebookům. Dříve telefony obsahovaly pouze firmware poskytující klasické telefonní funkce, ale kolem roku 2002 se objevily první telefony s možností nahrávání nových aplikací. V podstatě najednou se objevilo hned několik programových platforem. Některé z nich už zanikly, ale mnohé se prosadily do běžného života. Z nejznámějších zaniklých můžeme jmenovat například Mophun na telefonech Ericsson a posléze Sony-Ericsson. V současnosti nejrozšířenější platformy jsou Windows Mobile, Symbian, BREW, Blackberry a Java (J2ME). Nově se prosadil Google Android a iPhone a úplnou novinkou je Bada (dosud jen na jednom zařízení od společnosti Samsung).

Windows Mobile je plnohodnotný operační systém pro „chytré“ telefony (smartphones) a aplikace určené pro něj lze programovat obdobně jako aplikace pro desktopové Windows – existuje také podpora pro .NET a dají se instalovat i virtuální stroje a emulátory pro jiné platformy (např. J2ME).

Symbian je jednodušší operační systém s mnoha omezeními. Nejvíce je podporován firmou Nokia, ale lze ho najít i na zařízeních od Sony-Ericsson, Motorola, Samsung a dalších. Aplikace lze psát v C++ (je však například nutné dodržovat určité standardy při práci s pamětí), ale také v Javě, protože každý telefon s OS Symbian obsahuje standardně i virtuální stroj J2ME. Programátoři jsou poměrně silně svázáni knihovnami a nelze se dostat ke všem vlastnostem telefonu. Od verze 9.0 musí být navíc všechny aplikace digitálně podepsané, což bylo v předchozích verzích pouze volitelné.

BREW je platforma založená na jazyku C++, rozšířená nejvíce v USA. Hlavní nevýhodou této technologie je zpoplatnění licence na používání kompilátoru. Zařízení s BREW často obsahují i podporu pro J2ME. Objevily se i komerční řešení pro převod J2ME aplikací do BREW, ale zpravidla jsou vždy nutné zásahy programátora při ladění výkonu.

Blackberry je výrobce telefonů, které jsou používány převážně ve Spojených státech amerických. Jsou typické úplnou QWERTY klávesnicí. Aplikace jsou psány v Javě a mohou být dvou typů – buď jsou to klasické J2ME, nebo nativní Blackberry aplikace. Tyto nativní mají k dispozici o něco širší nabídku API a tedy i širší přístup k nejrůznějším funkcím telefonu. Díky možnosti nativního přístupu se ovšem pro uživatele mohou stát bezpečnostním rizikem (např. odposlech hovorů) a proto Blackberry vyžaduje jejich validaci. Validace je vázána na podpis smlouvy mezi vývojářskou společností a společností Blackberry a je samozřejmě placená.

Android je systém vyvinutý společností Google založený na Linuxu. Aplikace se však pro něj mohou psát pouze v Javě za použití specializovaných knihoven. Dlužno ovšem říci, že nejsou dodržovány mnohé Javové specifikace – například je zde naprosto odlišná verze bytekódu. Od verze API 1.5 je také možné z javového kódu volat i metody v nativních knihovnách. Zvláštním specifikem je to, že aplikace se skládají z několika aktivit, což jsou struktury podobné procesům. Každá má přidělené vlastní grafické okno a přepínání mezi okny aplikace znamená i přepínání mezi aktivitami. Je možné udělat přímočarý převod z J2ME aplikace

do aplikace určené pro Android pouhou implementací knihoven z J2ME pomocí knihoven z Androidu. Jelikož platforma Android funguje na mnoha typech různých zařízení s velmi rozdílnými rozměry obrazovky, každá aplikace může definovat více skupin obrázků a zdrojů, které se pak načítají podle aktuálního zařízení. Důsledkem je však nárůst velikosti aplikačního balíku.

Nástup programovatelných aplikací pro mobilní telefony byl zaznamenán nejvíce na poli her. Pro platformy s operačními systémy existuje kromě her i velké množství jiných aplikací, ale na telefonech vybavených pouze J2ME mají hry drastickou převahu – ostatní aplikace jsou většinou velmi jednoduché utility jako kalendáře, paměti hesel, převodníky jednotek a měn, nebo weboví klienti určené pro komunikaci s jedním určitým serverem. Právě hrám a jejich vývoji se budeme více věnovat.

Kromě toho se zaměříme pouze na platformu J2ME, protože programátory oproti ostatním platformám nejvíce omezuje a oproti programování pro PC obsahuje mnohá specifika. Na ostatních platformách (WM, Symbian, BREW) lze psát aplikace přímo v C/C++ nebo dokonce assembleru a principy nejsou příliš odlišné od aplikací na desktopech - rozdílné jsou hlavně programové knihovny, nikoliv princip aplikací. Naproti tomu J2ME má oproti Javě na desktopu i mnohá další omezení.

Kapitola 2

Java na mobilních zařízeních

2.1 J2ME a midlety

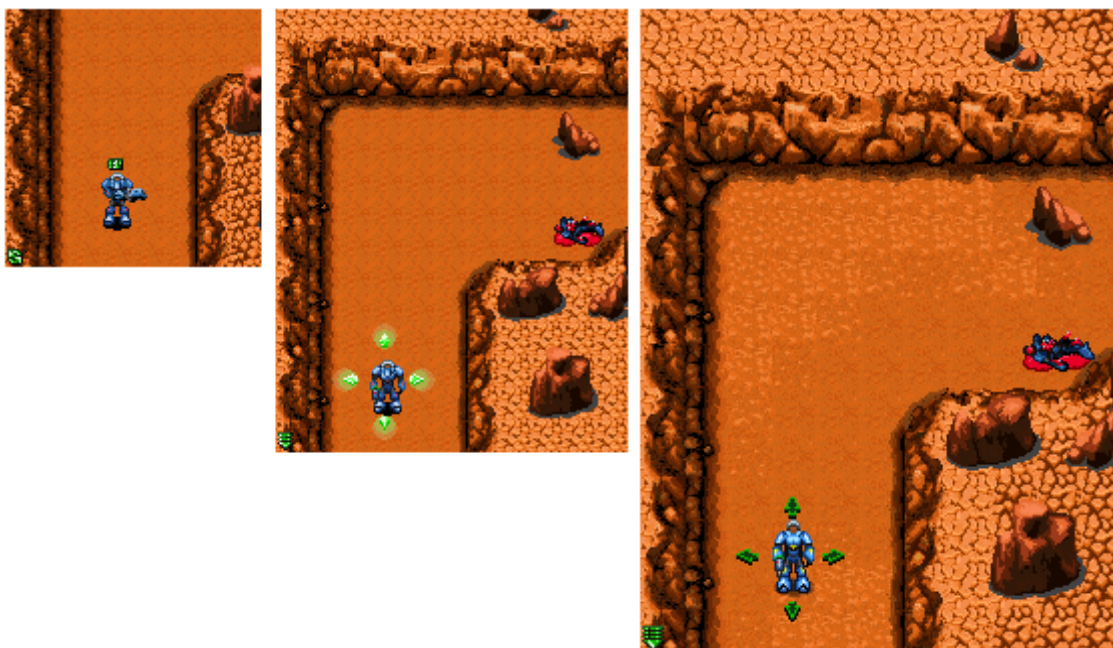
J2ME neboli Java Microedition je označení pro soubor několika javových knihoven – zejména jde o specifikace knihoven CLDC [1] a MIDP [2]. CLDC jsou v zásadě zjednodušené verze Java balíků *java.lang* a *java.util* z J2SE. MIDP jsou knihovny pro ovládání GUI a komunikaci. Používá se specifikace jazyka Java verze 1.3 a specifikace bytekódu verze 1.1. Při návrhu byl kladen důraz na jednoduchost virtuální stroje, který měl být původně implementovatelný do 100 kB paměti (tzv. KVM – *kilo virtual machine*).

J2ME aplikace (tzv. *midlets* – z Microedition Device) jsou svou strukturou velice podobné appletům – jedna hlavní třída zajišťuje životní cyklus aplikace a přístup k jediné instanci displeje, přes kterou lze přepínat jednotlivé obrazovky aplikace nebo na něj přímo kreslit. Každá obrazovka (*Screen*) může obsahovat formulářové prvky systému (seznamy položek, vstupní políčka apod.) nebo na ní lze vše kreslit pomocí grafických primitiv (*Canvas*). Drtivá většina her používá jen jednu obrazovku, na kterou vše kreslí pomocí grafických primitiv, a nepoužívá žádné nativní formuláře, neboť u nativních formulářů nelze dobře kontrolovat vzhled - a vzhled je pro hry velmi důležitý. Formulářové prvky jsou typičtější pro neherní aplikace, ale i ve hrách se musí občas použít – například pro vkládání textu uživatelem.

Struktura aplikací se pak už mnoho neliší od desktopových aplikací. Mnohé knihovny jsou i velice podobné. Existují však značné rozdíly ve stylu programování, protože API nemají příliš mnoho možností a telefony samy nutí programátora se ještě více omezovat.

2.2 Přenositelnost a fragmentace – když Java přestane být Javou

Java byla původně vytvořena jako přenositelný multiplatformní jazyk, a to byl i jeden z důvodů, proč se zdálo, že je vhodná jako univerzální programová platforma pro různorodá mobilní zařízení. Je pravda, že i pro mobilní telefony lze vytvořit aplikaci, která funguje na každém J2ME zařízení (tzv. generická aplikace), ale u her (resp. u profesionálních her) se tento přístup prakticky nikdy nepoužívá. Problémem je totiž grafický vzhled. Mobilní aplikace vždy využívají celý displej telefonu – a ty jsou svým rozlišením velice různorodé. Je zřejmé, že nelze například použít stejné grafické prvky na displej s rozlišením 128x128, 128x160, 176x208, 208x208, 240x240, 260x320 a 320x240, 480x640, 600x800, 800x320 (což nejsou zdaleka všechna používaná rozlišení). Protože ani poměr stran není zachován, nelze změnit velikost obrázků až za běhu, ale vše musí být předpřipraveno předem (scaling obrázku není na většině telefonů ani snadné udělat) – proto se pro každou velikost displeje vytváří speciální verze, se specifickými velikostmi obrázků a rozložením grafických primitiv na obrazovce tak, aby byla celá obrazovka zařízení optimálně využita a grafický dojem byl co nejlepší. Dále se třeba rozlišují telefony podle podpory průhlednosti (plný alfa kanál nebo jen rozlišení na průhledné/neprůhledné), nebo podle možností překlápět obrázky za běhu (na mnohých telefonech to není dostatečně rychlé, nebo to vůbec neumí).



Obrázek 1: Ufo Afterlight - rozdíly rozlišení 128x128, 176x208, 240x320

Není ani možné používat fonty, které jsou na telefonech běžně k dispozici, neboť jsou dostupné jen tři velikosti – označené jako malá, střední a velká. Na každém telefonu má každý font bohužel jinou reálnou pixelovou velikost a často chybně informují o velikosti například u písmen s diakritikou, nebo dokonce některé diakritické znaky neobsahují vůbec. Proto je nutné pro dodržení jednotného vzhledu a správné lokalizace používat vlastní rasterové fonty (vektorové není možné implementovat, telefony neumí kreslit křivky).

Dalším problémem přenositelnosti jsou drobné rozdíly mezi jednotlivými platformami. J2ME má verzi konfigurace (CLDC 1.0 nebo 1.1), která určuje, zda je na daném zařízení možné používat čísla v pohyblivé řádové čarce, a verzi profilu (MIDP-1.0, 2.0 nebo 2.1), která určuje možnosti grafických knihoven.

I když telefon operace v řádové čarce podporuje, často to není hardwarová podpora, takže mohou hru velmi zpomalovat a je nutné se jim vyhnout v časově náročných výpočtech, zejména uvnitř těl cyklů. Obvykle lze skutečnou rychlost zjistit teprve po vyzkoušení přímo na zařízení.

První verze profilu měla grafické knihovny značně omezené, takže výrobci mobilních telefonů je začali rozšiřovat vlastními přídatnými API, která se navíc často lišila model od modelu. Tato API poskytovala různé funkce jako např. dynamickou změnu velikosti obrázků, vibrace nebo přehrávání zvuků. Nejvíce používané bylo tzv. Nokia UI API [3]. Jde o velmi dobře navrženou knihovnu, která přidala možnosti přehrávání zvuků, přístup k vibracím, podsvětlení, možnost využití celé obrazovky (odstranění pruhu s ukazatelem signálu a baterie), přímého vykreslování pole pixelů na displej (dříve jen možnost kreslit předpřipravené obrázky) a několik dalších pomocných funkcí. Šlo vlastně o základní funkce potřebné pro vytváření kvalitních her. Díky tomuto API byla většina složitějších her tvořena pouze pro telefony Nokia. To posléze donutilo ostatní výrobce telefonů, aby Nokia UI API začali podporovat i ve svých zařízeních (všechny Sony-Ericsson modely počínaje K700i). I dnes se tato knihovna stále používá, novinkou je třeba API pro vytvoření odezvy pro uživatele na dotykových displejích.

Druhá verze MID profilu (MIDP2) obsahuje prakticky vše, co tehdejší Nokia API (zvuky pomocí samostatného Multimedia API [4]) a v ideálním případě proto není třeba staré nativní knihovny používat (s výjimkou portací na staré modely). Bohužel, často jsou programátoři nuceni používat je i na některých nových zařízeních.

Největším problémem pro přenositelnost mezi jednotlivými zařízeními totiž není jen různá podpora J2ME knihoven, kterých se už objevilo velké množství (*sockets*, SVG grafika, 3D grafika, streaming videa z internetu, přístup k GPS, ovládání pomocí akcelerátoru, přístup ke kameře telefonu, přístup k druhému displeji telefonu, SMS, Bluetooth apod.), ale hlavně rozdíly v jejich implementaci, chyby v implementaci a nedodržování specifikací. Různá podpora knihoven je způsobena tím, že některé části knihoven byly specifikovány jako volitelné a některé specifikace mají více verzí. Pro potřeby her se tento problém týká hlavně Multimedia API a Bluetooth API. Nikde není definováno, jaké formáty zvuků musí jít přehrát, takže podle typu telefonu je nutné vytvořit různé verze se zvuky v různých formátech (MIDI, WAV, AMR, MMF, MP3), případně i podle toho, jestli je telefon schopen přehrát více zvuků zároveň a které kombinace formátů je schopen přehrát zároveň. Telefon dále může například podporovat změnu hlasitosti zvuků, ale zase jen u určitých formátů, nebo přehrávat zvuky jen určitých vzorkovacích frekvencí a rozlišení. Některé další formáty lze pak případně přehrát pomocí starých nativních knihoven. Při vývoji je bohužel nutné s těmito odlišnostmi počítat. Bluetooth API se budeme podrobněji věnovat později.

Jako příklad toho, že i přesná specifikace může být špatně implementovaná, jmenujme například chybu na Nokiích řady s40 5th FP1 (telefony představeny v roce 2008), kde jsou při vykreslování některá primitiva posunuta o jeden pixel oproti správné pozici. Takový rozdíl se zdá malý, ale pro uživatele je na malém displeji velice dobře viditelný a je nutné vědět, jak tuto chybu obejít.

Počet chyb v implementaci byl dříve velmi vysoký, ale s novějšími telefony naštěstí podstatně poklesl. Nechvalně proslulá byla zejména společnost Samsung – na některých telefonech aplikace nešly ani spustit a bylo nutné postupně odstraňovat celé bloky kódu a postupně je vracet zpět, aby se zjistilo, kde je problém – a často ho nebylo možné najít. Jedním z takových případů byla třeba kompilace kódu do nativních instrukcí při načítání třídy na Samsungu D500, která se nedokázala vypořádat s dlouhými metodami nebo některými složitějšími bloky kódu. Ukončení aplikace bez jakéhokoliv chybového hlášení bylo poměrně běžné (opět např. model D500). Mimochodem, vývojové prostředí Samsungu nebylo možné na PC spustit a oficiální radou bylo otevřít *exe* soubor v hexaeditoru a změnit ručně hodnoty některých bytů. I ostatní výrobci měli mnoho zvláštních problémů - například změna síly podsvětlení displeje na symbianových Nokiích měla za následek vypnutí podsvětlení (tedy temnou obrazovku) až do chvíle, než byla aplikace ukončena a telefon restartován.

Výskyt chyb dal vzniknout internetovým knihovnám s popisem chyb na jednotlivých modelech a postupem, jak je obejít. Kvůli složitějším chybám je často potřeba pro daný telefon vytvořit specifickou verzi.

Dnešní vývoj probíhá tak, že telefony jsou rozděleny do kompatibilních skupin (těchto skupin může být 80 a víc) a každá taková skupina má svoje definované vlastnosti. V kódu je nutná přítomnost direktiv preprocesoru, které jsou vypínány a zapínány podle vlastností dané skupiny, a všechny další zdroje (obrázky, zvuky apod.) jsou vybírány taktéž podle skupiny. Bez preprocesoru se v mobilní Javě v podstatě nedá vyvíjet, protože jinak by bylo nutné spravovat desítky verzí zdrojového kódu.

Dalším oříškem pro přenositelnost jsou paměťové a rychlostní nároky telefonu. Problémem paměti je například omezení na maximální velikost aplikace. Na starých Nokia telefonech bylo toto omezení kolem 63-65 KB a optimalizace na velikost se stala skutečně velkou vědou. Nešlo jen o zmenšování grafiky, ale i o zmenšování kódu – například počet tříd aplikace bylo lepší držet na minimu, protože každá třída znamenala samostatný soubor a tedy samostatný kompresní slovník v distribučním archívu. Celkově všechny postupy vedly k malé čitelnosti kódu (např. běžná vykreslovací metoda mívala několik tisíc řádků). U novějších telefonů se limit posunul ke 130 KB, na aktuálních telefonech je velikost aplikace omezena většinou 1 MB (pokud není neomezena), což už nečiní větší problémy, jelikož průměrná velikost mobilní hry se pohybuje mezi 150 KB a 600 KB.

Při vývoji se musí počítat i s nízkou pamětí za běhu a často uvolňovat nepotřebné zdroje (tj. v Javě důsledně nastavovat reference na *null* a volat garbage collector). Tento problém je bohužel znásoben jedním špatným rozhodnutím ve specifikaci MIDP. Podle ní musí být totiž všechny načtené obrázky uchovávány v paměti v raw RGBA formátu, ať už byl původní formát obrázku jakýkoliv. Důsledkem toho je, že například obrázek na celý displej 240x320 zabírá v paměti telefonu 0,5 MB, z celkové dostupné paměti např. 3 MB. Stačí si představit kolik běžná plošinová hra používá malých obrázků (např. různých dlaždic pozadí může být až 100) a je vidět, že s pamětí je nutné velice šetřit.

Posledním problémem je rychlost. Samotné primitivní operace už jsou rychlé dostatečně (např. naše poslední testy ukázaly, že Nokia z roku 2008 dokáže vypočítat 10000 celočíselných součtů za čas jedné milisekundy). Větší problém než rychlost výpočtů bývá ovšem rychlost některých metod, zejména těch vykreslovacích. To pak omezuje maximální hodnotu FPS a tedy vizuální zážitek pro hráče. Jako příklad můžeme uvést, že na některých Nokiích je vykreslení obrázku o několik řádů rychlejší než vykreslení zrcadleného obrázku. Když je na těchto telefonech potřeba zrcadlení (např. sprite postavičky hráče musí mít možnost být otočen doleva i doprava), zpravidla se zrcadlení nedělá realtime, ale buď se předpřipraví v runtime paměti, nebo je třeba do zdrojových souborů umístit obě verze obrázků (což však zdvojnásobuje paměťové nároky na uložení zdrojů v distribučním archívu).

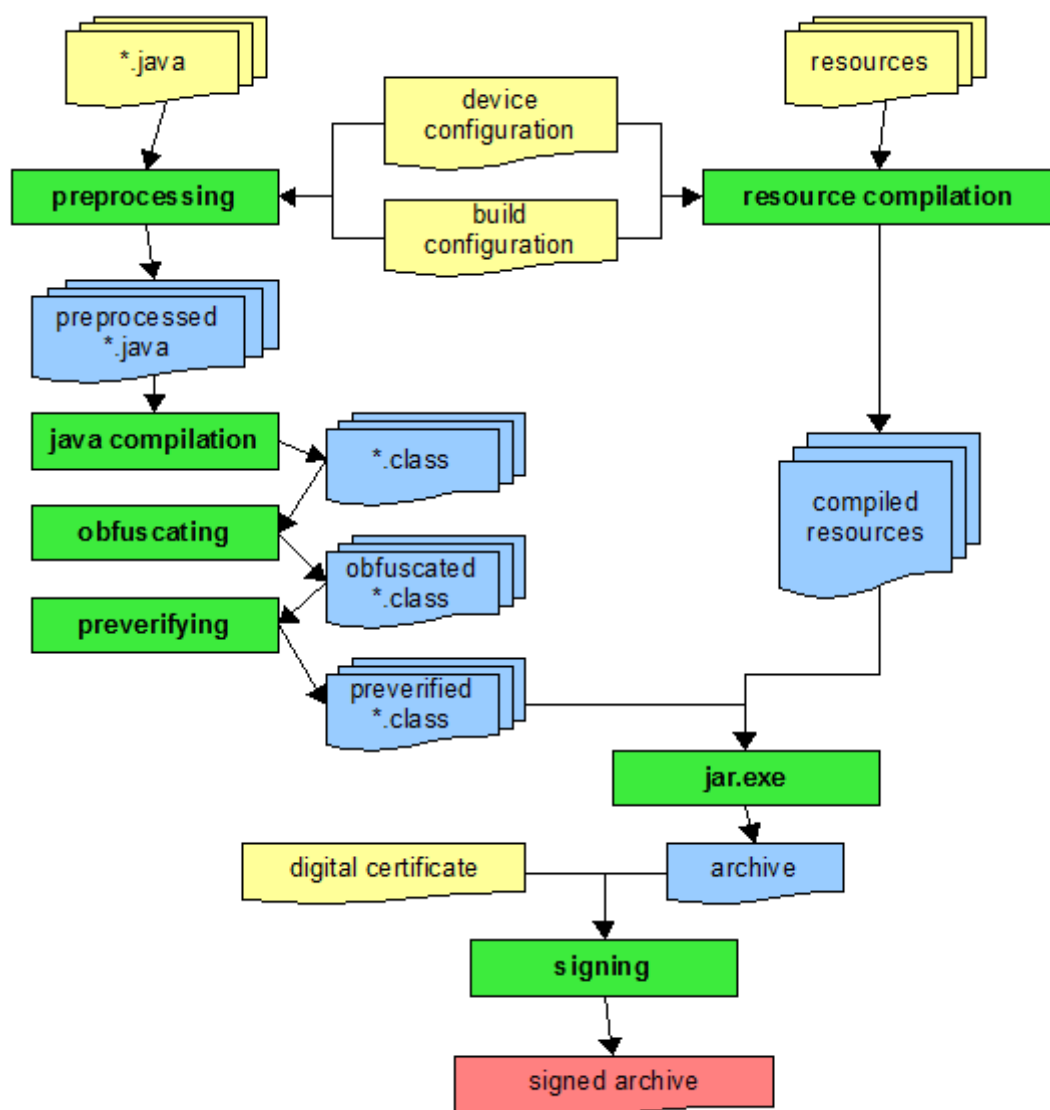
Většina MIDP2 telefonů už ovšem podporuje JIT kompilaci, ale paradoxně to někdy vytváří další problémy. Například symbianové Nokia z řady N70 kompilují zvlášť každý blok kódu těsně před tím, než se poprvé provede. Po spuštění hry to způsobuje nepříjemné sekání při každé dosud neprovedené herní akci a je nutné to řešit automatickým průchodem všech větví kódu během načítací fáze ještě před spuštěním samotné hry.

2.3 Cyklus kompilace mobilní aplikace

Když shrneme předchozí poznatky, dojdeme k tomu, jak vypadá typický building distribučních balíčků aplikace. Na začátku jsou zdrojové kódy a další zdrojové soubory (obrázky, zvuky, data úrovní, nastavení hry, texty v různých jazycích) a seznam skupin telefonů, které požaduje koncový distributor. Pro každou skupinu je třeba mít předpřipravené kompilační skripty, které obsahují přepínače preprocesoru, maximální velikost paměti, velikost obrazovky, kódy nativních kláves, dostupné knihovny, velikost ikony aplikace apod. Zdrojové kódy jsou pak nejprve preprocesorovány. Nejběžněji se používá preprocesor *antenna* [5], který je rozšířením kompilačního nástroje *ant* [6], ale lze použít i běžný C preprocesor. Ten není s Javou sice tak dobře propojitelný, ale umožňuje například inlinovat některé metody a tím podstatně zrychlit výpočty. Poté se provede běžná Java kompilace

do bytekódu. Kompilátor je třeba správně nastavit - J2ME používá specifikaci jazyka Java verze 1.3 a specifikaci bytekódu verze 1.1.

Dalším krokem je zpravidla obfuskace zkompileovaných tříd (*obfuscator*, zatemňovač). Původním smyslem obfuskátoru bylo chránění kódu oproti zpětné dekompilaci, takže jde zejména o nahrazení jmen tříd, metod a atributů velmi krátkými a nic neříkajícími názvy („a“, „b“ apod.), čímž však také dochází k podstatnému zmenšení velikosti distribučního balíku, často méně než na jednu polovinu. Mimo jiné zabraňují prozkoumávání kódu i tím, že ho upravují tak, aby po dekompilování nebyla možná opětovná kompilace. Tedy použitím vlastností bytekódu, které nejsou legální v Java jazyce, např. rozlišení metod jen pomocí návratového typu nebo využívání instrukce *GOTO*. Moderní obfuskátory, jedním z nejlepších je bezesporu *Proguard* [7], však poskytují i velmi důkladnou optimalizaci kódu, která dokáže například odstranit abstraktní třídy a rozhraní nebo inlinovat metody (výběr metod pro inlinování se provádí automaticky) a velmi zrychluje výsledný kód.



Obrázek 2: Buildovací proces J2ME aplikací

Před umístěním tříd do distribučního balíčku je ještě třeba provést jejich preverifikaci. Běžný javový virtuální stroj při načítání každou třídu verifikuje. Proces verifikace však potřebuje příliš mnoho paměti (udává se až 50 kB, [8]) a zejména na starších zařízeních ho nebylo možné implementovat. Proto se část tohoto procesu přenáší do buildovacího procesu. Z kroků verifikace se provádí převážně paměťově náročné kontroly těl metod a kontroly používání zásobníku každé metody. Výsledek je pak uložen do atributu metody *StackMap*, aby mohl být využit během verifikace na mobilním zařízení.

Ostatní zdrojové soubory jsou zpravidla převedeny do binární formy (není obvykle nechat např. nastavení úrovní v XML formátu) a poté spojeny do několika málo souborů, čímž se zrychlí načítání a zlepší komprese v archívu. Všechny soubory aplikace jsou následně spojeny a zkomprimovány do distribučního jar archívu.

Po této fázi se někdy distribuční balík také digitálně podepisuje, ale vzhledem k tomu, že různé telefony obsahují různé kořenové certifikáty a je nutné tedy mít několik různých digitálních certifikátů (které jsou placené), tak se tento krok zpravidla vypouští. Digitální podpisy se více týkají aplikací, které používají funkce, jež mohou narušit bezpečnost a při jejichž použití se telefon ptá uživatele na potvrzení – např. přístup k souborům, nahrávání z mikrofону, přístup k internetu nebo posílání textových zpráv.

Pro celý buildovací postup si firmy tvoří vlastní řešení. Jelikož vytváření skriptů pro jednotlivá zařízení není možno dělat bez detailního testování na těchto zařízeních a je to tedy velmi zdlouhavé a drahé, jde i o jedno z největších tajemství vývojářských firem.

Existují i komerční řešení (např. *J2ME Polish* [9]) a některé firmy dokonce nabízejí portování hry z jediné hlavní verze (tzv. master verze). Externí portování už hotové hry ovšem nikdy nedosahuje takových kvalit, jako když na něj programátoři hry myslí už dopředu.

Kapitola 3

Mobilní 3D hry

3.1 Úvod

Když probereme jednotlivé hry, které se na mobilních telefonech objevily, zjistíme, že jsou většinou poměrně jednoduché. Mimo nejrůznější logické hry, jako jsou tetris, skládačky apod. jde obvykle o hry, kde se pozadí skládá z dvojrozměrných dlaždic a na popředí se pohybují jednoduché animované postavičky, autíčka apod. Někdy se tímto způsobem vytváří i izometrický 3D dojem (např. hry *Townsmen*, *Age Of Empires*), ale nejde o skutečnou třídimenzionální grafiku. Stejně tak i kolize se počítají jen na dvojrozměrné čtvercové síti. Samozřejmě existují i výjimky, můžeme najít hry se složitějším fyzikálním modelem a propracovanější grafikou (někdy i vektorovou), ale skoro vždy jde zase jen o 2D.

I přes velká omezení mobilních telefonů se nakonec objevily i 3D hry. Ještě před několika lety bylo velice složité takovou hru naprogramovat. Jedinou možností bylo vytvořit celý grafický engine z ničeho, ale přesto se první taková hra objevila už na jedněch z prvních barevných telefonů. Dnes už je situace o mnoho jednodušší. Dnešní telefony už standardně obsahují i grafickou knihovnu pro zobrazení 3D virtuálního světa, takže odpadá nízkoúrovňové programování grafiky a místo toho se práce programátoru přesouvá ke zdokonalování fyzikálního systému, zejména detekce kolizí. S novějšími telefony se dokonce objevují i integrované grafické karty a API pro jejich ovládní.

V tomto textu se budeme věnovat nejdříve starším hrám – podíváme se, jak byly implementovány, a zejména se zaměříme na implementaci grafického systému, jelikož fyzika byla zpravidla velmi jednoduchá. Poté prozkoumáme javové 3D knihovny a možnosti jejich využití.

3.2 Historické 3D hry

3.2.1 MDoom

První 3D hrou pro mobilní telefony byl bezpochyby *MDoom*. Tato hra se vlastně objevila ještě dříve než byly k dispozici reálné telefony s Javou – byla tedy odladěna jen na prvních emulátorech J2ME. Na reálných telefonech se posléze ukázalo, že je tato hra příliš pomalá a v podstatě se ani vůbec hrát nedala. Šlo spíše jen o technický experiment. Grafický cyklus byl velice primitivní – pomocí API v té době ještě nebylo možné poslat na displej přímo pole pixelů, ale daly se kreslit jen předpřipravené obrázky. V jednom cyklu tedy vždy došlo pouze k přemazání obrazovky a vykreslení zdi. Všechny zdi byly stejně vysoké a mohly vést pouze ve dvou kolmých směrech. Fyzicky šlo tedy pouze o 2D mřížku – s obdobným pojetím světa se setkáme u všech starších 3D her.

Vykreslování zdi pak probíhalo jednoduše – nejdříve se transformační maticí spočítaly obrazy všech bodů na mřížce a poté se kreslilo odzadu dopředu. Pro vykreslení zdi se používaly dva černobílé obrázky – oba měly výšku o něco větší než byla výška obrazovky. Když byla zeď těsně u hráče, obrázek se vykreslil celý vystředěný na střed obrazovky. Vzdálenější zdi pak měly oříznutou spodní a horní část, takže vizuálně vypadaly menší a tedy vzdálenější. Jeden

z obou obrázků byl o něco tmavší a tím bylo řešené primitivní stínování – světlejší zdi byly zdi blízké k hráči, vzdálenější byly tmavší.

Pohyb hráče nebyl plynulý – otáčení probíhalo v osmi nespojitých krocích, stejně tak i chůze probíhala skokově a pouze po středu čtvercových dlaždic. Vzhledem k tomu, že zdi byly všude stejné, nedalo se ve hře vůbec orientovat, protože pohledy do všech stran vypadaly takřka identicky.

Tuto hru připomínáme pouze z historických důvodů, z technického hlediska byla spíše odstrašujícím příkladem. Vůbec neřešila rychlost mobilních telefonů a vykreslování bylo velice pomalé (byly kresleny i neviditelné zdi). Když jsme tuto hru v roce 2003 zkoušeli na jednou z prvních telefonů s Javou, Nokii 6610, dosahovali jsme rychlosti zhruba 1 FPS. Tento telefon byl přitom v té době jednoznačně jeden s nejrychlejších – dokonce i do novějších telefonů stejné řady Nokia montovala pomalejší procesory.

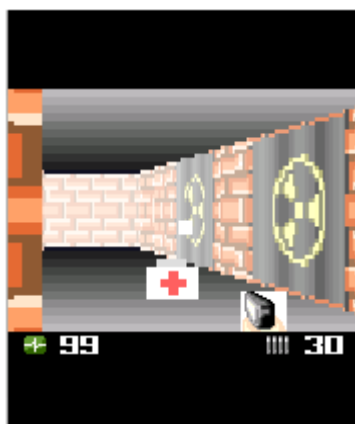


Obrázek 3: MDoom

3.2.2 3D Adventure

První skutečnou 3D hrou pro J2ME byla *3D Adventure* od slovenského programátora Ondřeje Škovrana, která se objevila během roku 2004. Inspirací pro tuto hru byla očividně klasická PC hra *Wolfenstein 3D* z roku 1992.

První zajímavou věcí, které si můžeme všimnout je skutečnost, že hry pro mobilní telefony jsou za PC hrami pouze asi o 12 let pozadu. A to mluvíme o interpretovaném kódu, který se i se všemi obrázky musel vejít do 64 KB!



Obrázek 4: 3D Adventure

3D Adventure byla jako jedna z prvních her založena na používání tehdy nového Nokia API, které umožňovalo vytvořit si pixelový buffer a najednou ho poté vykreslit na obrazovku (dříve jen grafická primitiva – čtverce, kruhy, čáry a předpřipravené obrázky). Hru to omezovalo prakticky jen na tři řady telefonů – Nokii series 30 (jediný barevný telefon 3510i, displej 96x65), první Nokie series 40 (již zmíněná 6610 a kompatibilní, 128x128) a první Nokie series 60 (3650, 7650, 6600, symbianové Nokie s displejem 176x208). Z hlediska distribuce se prodej takové hry příliš nevyplácel, protože barevné telefony s Javou nebyly ještě příliš rozšířené, ale vývojáři v té době byli ještě v naprosté většině pouzí nadšenci, kteří pracovali jednotlivě a nesnažili se vývojem vydělávat.

Z hlediska fyziky měla tato hra mnohé společné s hrou *MDoom*. Svět byl ještě stále dvou-dimenzionální mřížka. Pohyb a řešení kolizí už však bylo o něco propracovanější. Zatímco *MDoom* umožňoval pouze otáčení o 45 stupňů a pohyb o pevnou délku kroku dopředu a dozadu, *3D Adventure* měla otáčení detailnější (48 kroků po 7,5 stupních), krok byl kratší a byly umožněny i úkroky do stran. *MDoom* v podstatě ještě kolize řešit nemusel, protože hráče nikdy nepustil do dostatečné blízkosti ke zdi. *3D Adventure* už to řešit musela, ale stále šlo o výpočetně velmi jednoduché řešení. Čtverce v mapě byly rozděleny na plné a volné, každý čtverec dále obsahoval informace o tom, jaké zdi se v něm mají zobrazit (tj. typ textury). Každý krok hráče v mřížce se rozdělí na horizontální a vertikální souřadnici, oba kroky se pak provedou zvlášť, a pokud by některý z nich prošel do přílišné blízkosti plného čtverce, krok se zkrátí. Mezera mezi zdi a hráčem je nutná kvůli jednoduchosti grafického enginu – pokud by se hráč dostal příliš blízko, vidí skrze zeď. Na rozdíl od *MDoom* tu již tedy byly odděleny informace pro kolize a pro vykreslování (typ textury zdi) - v první verzi hry byl dokonce jeden blok za grafickou zdi omylem označen jako prázdný a zdi šlo projít.

Grafická část hry je podstatně zajímavější než fyzika. První věcí, které si všimne hráč 3D her na PC je fakt, že nelze měnit úhel pohledu z horizontální roviny směrem nahoru a dolů. Pohled hráče tedy vždy směřuje vodorovně a navíc je nastaven tak, že je vždy ve výšce poloviny zdi. Když nakreslíme symbolickou vodorovnou čáru přes střed obrazovky, zjistíme, že horní polovina je zrcadlovým obrazem dolní poloviny. Díky tomu lze veškeré výpočty provádět pouze pro dolní polovinu obrazu a pak vyplnit spolu s pixely v dolní polovině zároveň i pixely v polovině horní.

Dalším trikem je vykreslení podlahy (a stropu) – z náhledů to není vidět, ale při hraní je zřejmé, že podlahy se nemění. Hra obsahuje jen statický obrázek evokující vzrůstající vzdálenost.

Následující pozorování se může zdát triviální, ale umožňuje zásadní zjednodušení grafického výpočtu. Když se podíváme na jednotlivé sloupce pixelů, zjistíme, že v jakoukoliv chvíli se v každém sloupci zobrazuje jen jedna zeď, tj. nejbližší zeď v jednom sloupci plně překrývá jakoukoliv zeď v pozadí. Sloupec obrazovky je tedy rozdělen na tři souměrné části - strop, zeď a podlahu. Výška zdi ve sloupci je navíc nepřímo úměrná vzdálenosti zdi.

Prvním krokem algoritmu je zjištění, jaká část mapy bude vykreslována. To je zařízeno jednoduše – vždy se prochází všechny zdi v oblasti mapy o rozměrech $N \times N$. Pozice tohoto čtverce v mapě je daná pozicí hráče a úhlem jeho pohledu (úhel určuje jednu z osmi oblastí okolo). Pro každý vrchol (roh nějakého čtverce mapy) ve vykreslované oblasti jsou poté spočítány jeho souřadnice na obrazovce. Pro každý vrchol je nejprve zjištěna jeho relativní souřadnicová vzdálenost od hráče (dvou-dimenzionální vektor) a na ní je aplikovaná rotační matice podle úhlu pohledu hráče. Získáme tak souřadnice $[x1, z]$, kde $x1$ je vzdálenost bodu

od středu obrazovky (doleva záporná, napravo kladná) a z je vzdálenost mezi hráčem (resp. spodní hranou obrazovky) a daným vrcholem (záporná pro body za hráčem). Poté se aplikuje jednoduchá perspektivní projekce

$$x = x1 \cdot \left(1 - \frac{z}{|z| + 1} \right)$$

Stejně bychom mohli získat i ypsilonovou souřadnici, pokud bychom použili stejnou rovnici s polovinou výšky obrazovky místo $x1$ (výška zdi je rovna výšce obrazovky, tedy vzdálenost podlahy od středu obrazovky je polovina výšky). Pro pozdější zpracování je ovšem výhodnější uložit pro každý vrchol poměr $z/(|z| + 1)$. Tento poměr je záporný pro body za obrazovkou a s rostoucím z také roste (i když maximálně jen k číslu 1). Má tedy stejné vlastnosti, jaké má vzdálenost od obrazovky, a lze ji použít jako hodnotu zbufferu. Zbuffer bude pole těchto hodnot o velikosti rovné šířce obrazovky, protože dle předchozích pozorování je potřeba jen jedna hodnota pro celý sloupec.

Nyní je potřeba projít veškeré mapové vrcholy, které jsou obsaženy ve vykreslované oblasti mapy, a zpracovat všechny zdi v oblasti. Textura, která se zobrazuje na každou zeď, vždy začíná na levém kraji zdi a končí na pravém kraji zdi. Je zřejmé, že sloupec pixelů textury, který se objeví v nějakém sloupci pixelů na obrazovce, nezáleží na poměrné výšce daného sloupce (tj. nezáleží na tom, kde v daném sloupci začíná podlaha a strop), ale záleží pouze na poměrné šířce zdi a na vzdálenosti daného sloupce od levého okraje zdi. Jinak řečeno, pro každý sloupec na obrazovce je jednoznačně určeno, který sloupec pixelů textury se v něm objeví. Po zpracování všech zdí chceme o každém sloupci vědět jaká zeď (typ textury) se tu má vykreslit, do jaké výšky od středu obrazovky zeď dosahuje a který sloupec textury tu má být vykreslen. Pro uložení hodnot jsou tedy potřeba jen tři pole o velikosti šířky obrazovky (zbuffer označuje zároveň výšku zdi). Algoritmus vykreslení zdí je následující:

1. Pokud je vzdálenost levého i pravého kraje zdi záporná (zeď je za obrazovkou a není vidět), potom skončí.
2. Do proměnných $x1$, $x2$ ulož levý a pravý roh zdi. Pokud $x1 = x2$ (zeď je kolmá na obrazovku a není vidět), potom skončí.
3. Pokud $x1 > \text{šířka obrazovky} / 2$ nebo $x2 < - \text{šířka obrazovky} / 2$ (celá zeď je mimo obrazovku a není vidět), skončí.
4. Spočítej směrnici přímky od bodu $[x1, z1]$ do $[x2, z2]$, kde $z1$ a $z2$ jsou vzdálenosti obou bodů. Každý bod zdi je potom na této přímce. Tj. pro každé x v intervalu $\langle x1, x2 \rangle$ můžeme zjistit jeho vzdálenost.
5. Projdi všechny body x v intervalu $\langle x1, x2 \rangle$.
 - 5.1. Pro x spočítej vzdálenost z pomocí směrnice přímky.
 - 5.2. Pokud je z kladné (bod je před obrazovkou) a z je menší než $zbuffer[x]$ (je blíže než poslední vykreslená zeď v tomto sloupci), potom pokračuj v těle cyklu, jinak jdi na další bod.
 - 5.3. Do $zbuffer[x]$ dosad' spočtenou vzdálenost.

5.4. Do $textureType[x]$ dosad' index textury na vykreslované zdi.

5.5. Spočítej index sloupce v textuře pro sloupec obrazovky x

$$pixelColumnIndex[x] = (x - x_1) \cdot \frac{textureWidth}{x_2 - x_1}$$

Tento vzorec rozprostře texturu rovnoměrně od levého kraje do pravého kraje zdi. Výraz $x_2 - x_1$ reprezentuje zobrazenou šířku zdi.

Nyní už lze přistoupit ke spočítání zobrazených pixelů. Na vykreslení potřebujeme pixelový buffer o velikosti obrazovky (ve hře byl nakonec použit buffer o něco menší, protože spodní část displeje byla věnována konzoli). Do něj se ukládají barvy pixelů po sloupcích. Pro každý sloupec:

1. Nejprve je hodnota uložená v $zbufferu$ přepočítána na výšku zdi h (tedy na vertikální vzdálenost od podlahy ke stropu)

$$h = height \cdot (1 - zbuffer[col]) = height \cdot \left(1 - \frac{z}{|z| + 1}\right)$$

2. Dále je spočítán poměr r výšky zdi a výšky zobrazované textury

$$r = \frac{h}{textureHeight}$$

3. Pro každý sloupec obrazovky již známe sloupec pixelů textury, který se do něj bude zobrazovat. Tento sloupec vysoký $textureHeight$ se rovnoměrně rozprostře na řádky zdi. Nepoužívá se žádný algoritmus pro zjištění optimálního pixelu z textury, index pixelu je získán celočíselným zaokrouhlením.

4. Řádky nad zdi a pod zdi jsou vyplněny šedou barvou podle vertikální vzdálenosti od středu obrazovky – čím dále od středu, tím tmavší.

Přes vykreslené pozadí je možné vykreslit další herní objekty či NPC postavy. Algoritmus je obdobný – znovu se projdou vykreslované čtverce v mapě pro zjištění, co se má vykreslit. Objekty jsou poté seříděny a vykresluje se odzadu. Každý objekt se vykresluje po sloupcích, pro každý sloupec je zjišťováno, zda není překryt zdi. Odlišnost je pouze v tom, že výpočet není souměrný pro horní i spodní část obrazovky a musí se zároveň kontrolovat průhlednost jednotlivých pixelů daného objektu.

K algoritmu je třeba dodat ještě několik implementačních poznámek. Hra byla určena na telefony bez jakékoliv podpory výpočtů v pohyblivé řádové čárce, takže vše bylo nutné počítat v celočíselné aritmetice s fixní řádovou čárkou (posouvání o 10 bitů). Pro rotační matici je nutné mít uložené hodnoty sinů a cosinů pro každý možný úhel natočení. Kvůli rychlosti byla většina násobících operací převedena na postupné přičítání do proměnné, čímž vznikaly různé drobné chyby. Ty však na tomto rozlišení nebyly rozeznatelné.

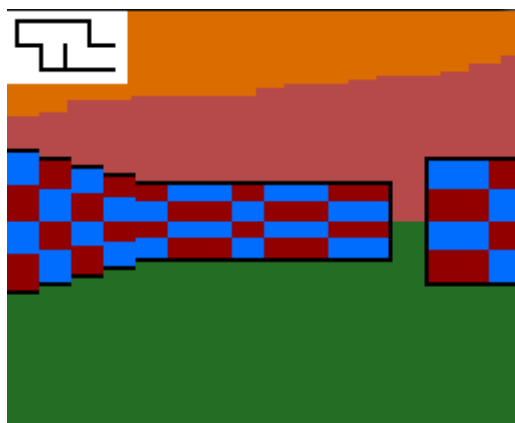
V algoritmu jsme se několikrát zmiňovali o paměťových výhodách. Ty byly pro tuto hru kritické. Nokia 6610 poskytovala programátorům cca 212 KB runtime paměti. Asi polovina z ní byla zaplněna samotným programem, část alokovanými daty. Telefon využíval

pro nativní reprezentaci pixelu 2-bytový formát (4 bity na každou komponentu) - herní plocha bez konzole byla 128x90 px veliká, tedy zabírala asi 20 KB, dále hodně zabíraly obrázky textur a objektů (textura 32x16 px, uložena pouze spodní polovina, druhá získána zrcadlením). Bylo nutné šetřit na všech místech.

Celkově vzato šlo o velmi dobře implementovanou hru, která překonala omezující podmínky mobilních telefonů své doby. Až na drobnosti dané hardwarem měla i vynikající hratelnost (avšak pouze tři herní levely).

3.2.3 StrangeMaze

Hra z roku 2003 je jedinou genericky fungující 3D hrou, běží dokonce i na starých černobílých telefonech. Generičnost bylo možné zajistit díky velmi jednoduchému hernímu konceptu, nenáročnému na ovládání. K ovládání jsou potřeba pouze čtyři směrové šipky – otáčení, zrychlování a zpomalování. Hráči k výhře stačilo pouze vyjít z bludiště (mapa byla zobrazena v levém horním rohu), ale závodil s dvěma NPC protivníky, kteří se ho snažili porazit. NPC se pohybovali částečně náhodně, částečně se drželi jedné strany bludiště - pro hráče to znamenalo jednoduše snažit se projít bludiště co nejrychleji, ale vzhledem k náhodnosti AI nebyla obtížnost vysoká.



Obrázek 5: StrangeMaze

Vykreslování bylo možné také provádět obecně, protože nepoužívalo žádné obrázky, ale pouze grafická primitiva - převážně jednobarevné obdélníky. Kreslilo se přímo na obrazovku, takže nebyly potřebné žádné velké grafické buffery a díky vykreslování pomocí primitiv nebyl nutný ani přístup přímo k pixelům a tedy k nativním API.

Samotné vykreslování bylo skutečně velmi jednoduché. Nejprve se vykreslilo pozadí podle úhlu natočení. Tady nešlo o 3D, ale pouze o horizontální posouvání 2D pozadí složeného z různě vysokých obdélníků simulujících kopce. Zdi se poté vykreslovaly jednoduchým raytracingem v úrovni středu obrazovky. Mapou byl vyslán horizontální paprsek a když dorazil ke zdi, byla spočítána výška zdi a zjištěna relativní poloha bodu dopadu vůči rohům zdi. Podle těchto údajů pak bylo možné vykreslit do obrazovky sloupec zdi složený z různobarevných obdélníků a ohraničený černými linkami na spodním a horním konci. Hra umožňovala nastavit kvalitu vykreslování na pět různých úrovní - ty pak ovlivňovaly, kolik paprsků bylo na šířku obrazovky posláno a tedy jak široké sloupce zdí byly kresleny pro každý paprsek.

StrangeMaze je jediná 3D hra, u které existuje rozsáhlé měření rychlosti na různých telefonech, ale z poněkud paradoxních důvodů. Hra samotná v sobě obsahuje jednoduchý benchmark, který měří čas potřebný na provedení několika tisíc násobení celých čísel. Výsledná hodnota je poté převedena na počet výpočtů za sekundu. Problematické je, že autoři hry tento benchmark udávali v FPS a zveřejnili ho na svých webových stránkách. Kupující tak získali dojem, že hra skutečně běží s rychlostí 60 FPS na Nokiích (obnovovací frekvence např. telefonu Nokia 3650 byla přitom pouhých 12 Hz). To nastartovalo vlnu diskuzních fór, na které hráči posílali reálně dosažená FPS jako protiváhu oficiálním výsledkům. Díky tomu víme, že na Nokiích běžela hra průměrnou rychlostí 10-12 FPS při nejhorší kvalitě (měření byla provedena na verzi hry upravené jedním z programátorů tak, aby měřila čas nejen výpočtů, ale i vykreslování). Tento výsledek byl na mobilní telefony více než slušný, ale vzhledem k jednoduchosti hry by bylo možné výpočty ještě dále zrychlit. Na starých černobílých telefonech značky Siemens byly oficiální výsledky 2 FPS, což znamenalo, že v realitě byla hra nehratelná – a nikoliv kvůli pomalému překreslování, ale právě kvůli pomalým výpočtům.

Výrazně také chyběl double buffer, takže jednotlivé prvky obrazu byly vykreslovány postupně a na rychlejších telefonech se stávalo, že další snímek byl vykreslován ještě dříve než skončilo vykreslování snímku předchozího, což mělo za následek problikávání zdi během hráčova pohybu.

3.2.4 3D Adventure 2 (Green Eye)

Vývoj této hry má na svědomí opět Ondřej Škovran. Když budeme zkoumat engine, zjistíme, že má mnohé společného s enginem prvního dílu. Je však podstatně dokonalejší. Bohužel však má i některé slabiny.

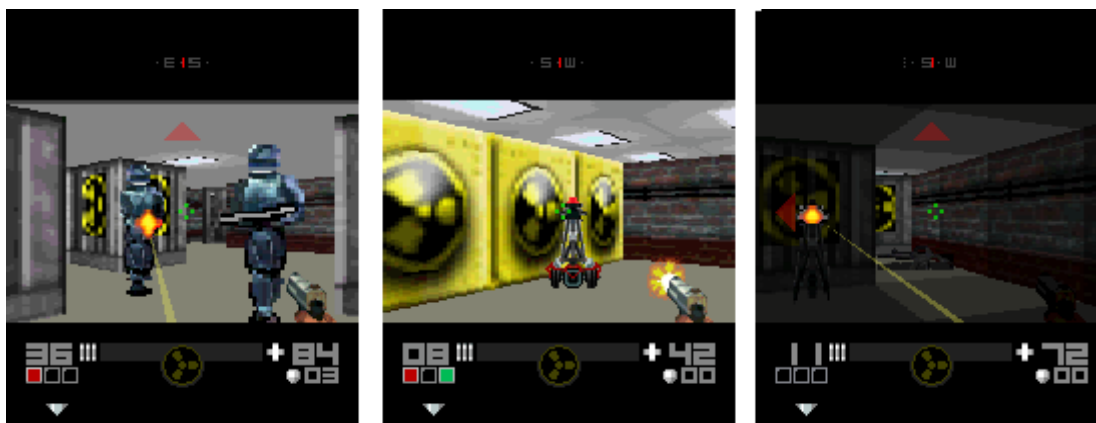
Fyzikální systém se až na detaily nezměnil a nebudeme se jím příliš zabývat. Svět je stále rozdělen mřížkou na plné a prázdné bloky, ale vzhledem k vylepšení grafiky už zde nemusí být mezera mezi hranou mřížky a hráčem. Hráč se tedy může pohybovat po celé ploše každé prázdné buňky.

Hlavní zdokonalení jsou na poli grafického enginu. V prvním díle se kreslily pouze zdi, zatímco podlahy a stropy byly vyplněny neutrálním obrázkem. V druhém díle je už strop a podlaha každé krychličky vyplněna barvou a lze zde dokonce zobrazit i texturu. Dalším vylepšením nového enginu je zavedení primitivního stínování. Každá krychlička má level designerem předdefinovanou hodnotu stínování od nuly (plné světlo) do sedmi (skoro tma). Stín se pak aplikuje na každou stěnu, strop a podlahu dané krychličky a velmi výrazně zlepšuje celkový vizuální dojem. Hráč má skutečně dojem, že se pohybuje v místnosti s fungujícím osvětlením.

Stínování je provedeno velmi jednoduše. Při generování pixelů pro danou zeď, podlahu či strop se program podívá na hodnotu stínu v dané krychličce a čím je stín hlubší, tím větší průhlednost (nižší hodnotu alfa kanálu) pixelu nastaví. Pixely se poté vykreslují přes černé pozadí a tedy čím větší průhlednost, tím větší ztmavení barvy při kombinaci s černou barvou na pozadí. Výhodou tohoto triku je, že výpočty míchání barev se přenášejí z interpretovaného javového kódu přímo do nativního kódu virtuálního stroje a jsou tedy rychlejší.

Samotné vykreslování zdi je prakticky stejné jako v předchozím díle. Vzhledem k menším paměťovým omezením už ale není nutné tolik šetřit texturami a ty už proto nemusí být

souměrné podle horizontální osy. Výpočet je však stále souměrný, pouze se pixel nekopíruje přímo do horní poloviny obrazovky, ale najde se v textuře pomocí hodnot vypočtených pro dolní polovinu obrazovky.



Obrázek 6: 3D Adventure 2

Protože engine je celkově složitější a tedy výpočtově pomalejší, provádějí se výpočty pouze pro každý druhý sloupec obrazovky a pixely se poté horizontálně zdvojují, tj. počítá se pro menší rozlišení obrazovky.

Z hlediska vykreslovacích algoritmů je největší rozdíl v první části výpočtů – ve zjišťování, které zdi se mají vykreslit. Starší engine vykresloval vždy pevnou oblast mapy. Nový engine používá opačný postup, konceptuálně podobný raytracingu. Prochází každý pixel dolní částí obrazovky směrem odspoda nahoru a pro každý počítá inverzní projekci. Tedy zjišťuje, jaký bod mapy by se zde měl zobrazit.

Pokud zvolíme vhodnou velikost čtverce mapy (tj. počet možných souřadnic v jednom čtverci), vyjde nám z inverzní projekce dvojice souřadnic, která určí nejen mapový čtverec, z něhož lze získat informace o stínování, textuře apod., ale určí také řádek a sloupec pixelu v textuře podlahy. Desetinné výpočty provádíme v celých číslech s fixní pozicí desetinné čárky. Pro získání indexů čtverce v mapě stačí bitově posunout souřadnici o desetinnou čárku plus velikost čtverce mapy - ve hře je zvolena jako 256 jednotek, na desetinná místa je použito 10 bitů, celkem tedy jen posun doprava o 18 bitů. Index do textury je pak určen vymaskováním části bitů za pomyslnou desetinnou čárkou – jejich počet je určen velikostí textury (ve hře jsou použity textury 32x32 px, tedy jde o 5 bitů za desetinnou čárkou).

Základem projekce je vzorec $\frac{1}{0.5+z}$, přičemž vzdálenosti za hráčem ($z < 0$) jsou počítány jako se vzdáleností $z = 0$. To je důležité pro vykreslování zdí, které mají jeden roh za obrazovkou a jeden před obrazovkou. Zjednodušení výpočtu pro tyto body způsobuje lehké deformace zdí, ale tyto deformace nejsou na malém rozlišení mobilních telefonů příliš viditelné.

Výsledné souřadnice pro převod bodu v mapě na bod na obrazovce (pokud neuvažujeme rotaci a dělení konstantou), jsou tyto:

$$x1 = x \cdot \left(\frac{1}{0.5 + z} \right)$$

$$y = (\text{height} / 2) \cdot \left(\frac{1}{0.5 + z} \right)$$

Konstanta *height* zde není viditelná část obrazovky, ale virtuální výška. Algoritmus virtuálně kreslí na čtvercovou obrazovku *width* x *width*, ze které však vykresluje pouze střední část obrazu. Virtuální výška obrazu je tedy rovná šířce obrazovky. Tím algoritmus dosahuje *aspect ratio* rovno jedné a nedochází k deformacím.

Výpočty projekce se dají provést i inverzně – pro každý bod $[x1, y]$ na obrazovce lze vypočítat bod $[x, z]$ v mapě, který se na něj zobrazí. Převod $y \rightarrow z$ je nezávislý na x , takže ho lze předpočítat pro každý řádek obrazovky. Druhý převod je složitější, ten ovšem není prováděn absolutně, ale raději se pro každý řádek (čili proměnnou y) uloží rozdíl v x mezi dvěma horizontálně sousedícími pixely.

Při vykreslování podlahy (strop se kopíruje) se procházejí řádky spodní poloviny obrazovky odspodu. Pro každý řádek se nejprve spočítají souřadnice prostředního pixelu (tj. $x = 0$). To se provede pouhou aplikací rotace na předpočítané hodnoty inverzní projekce. Dále se spočítá vektor $(xstep, zstep)$, který uvádí posun pozice od předchozí, pokud se posuneme na obrazovce o jeden pixel – jde opět jen o použití rotace na předpočítané hodnoty. Nyní se postupuje od středu nejprve k pravé straně obrazovky a poté od středu k levé straně a vždy se počítá mapová pozice bodu na obrazovce pouze přičtením vektoru.

Pro každý bod se vykreslí barevný pixel do bufferu. Pokud jde o plný mapový čtverec, tato skutečnost je zaznamenána. Podle pozorování u prvního dílu hry už víme, že v jednom sloupci obrazovky vždy může být pouze jedna zeď a pokud obrazovku procházíme od nejbližších bodů, pak ta první, na kterou narazíme je ta, kterou je třeba vykreslit. Tyto nejbližší zdi po každý sloupec uložíme do pole o velikosti rovné šířce obrazovky. Z tohoto pole je po průchodu celé obrazovky vytvořen seznam zdí pro vykreslení (duplicitní záznamy jsou v poli vždy v sousedních indexech).

Samotné vykreslení zdí pak proběhne stejně jako v prvním dílu. Vzhledem k tomu, že výpočty probíhají v celých číslech s omezenou přesností, dochází ke vzniku chyb. Tyto chyby se pak kumulují zejména při procházení spodní části obrazovky a postupném přičítání vektoru k mapové pozici. Kvůli tomu se může při počítání projekce pro zeď stát, že nebude po vykreslení na podlaze přesně umístěna a mezi ní a podlahou zůstane v několika sloupcích pixel prázdného místa. Tento problém naštěstí nebylo třeba řešit, protože engine překresloval vždy celou podlahu, ať už byla později zakryta zdí, nebo zůstala viditelná, takže na obrazovce nezůstávaly prázdné pixely.

Z implementačních detailů je třeba poznamenat, že vykreslování podlahy se ve skutečnosti neprovádí na celou jednu polovinu obrazovky, ale končí několik pixelů od jejího středu. Tam už se chyba výpočtu příliš kumuluje a vzhledem k malé velikosti vzdálených mapových čtverců dochází k jejich přeskokování. Při chůzi tak na obzoru mohou blikat zdi podle toho, jestli čtverec obsahující zeď byl přeskočen nebo ne. Hra tuto skutečnost řešila vhodným level designem – v mapách byly jednotlivé místnosti natolik malé, aby se chyba vzdálených zdí neprojevovala. Nám se ji sice podařilo na několika místech navodit, ale pro běžného hráče byla díky dobře navrženým mapám prakticky neviditelná.

Druhou chybou enginu je skutečnost, že kreslí zdi pouze na mapových čtvercích, které jsou před hráčem, resp. pixely jejich podlah jsou na obrazovce. Pokud se však hráč příliš přiblíží k nějaké stěně, její spodní hrana musí zákonitě zmizet pod hranu obrazovky a algoritmus ji nevykreslí. Proto se prozkoumává ještě i několik pixelů pod obrazovkou – tři pixely po pravé i levé straně obrazovky, v různých vzdálenostech pod její spodní hranou. Tak jsou zpracovány i mapové čtverce natolik blízko ke hráči, že už jejich podlahy (a stropy) nejsou viditelné.

Ostatní grafické prvky se vykreslovaly opět stejně jako v prvním díle hry. Novinkou byly jen složitější animace NPC. Jejich sprity už byly rozdělené do fragmentů a animace se prováděla ne přepínáním celého obrázku, ale skládáním fragmentů spritu do jednotlivých snímků animace.

Na první pohled byla tato hra opravdu vynikající, měla nádhernou grafiku, pěkně navržené levely, zajímavé efekty (např. noční vidění – buffer displeje prošel ještě barevnou transformací do odstínů zelené), ale jejím problémem byla rychlost. Grafický engine byl na tehdejší telefony příliš složitý. Většinu jeho chyb poté vylepšila hra *Paintball*.

3.2.5 Paintball

O této hře, kterou jsme implementovali v roce 2004, se nebudeme rozepisovat příliš obšírně. Její grafické algoritmy jsou založeny převážně na algoritmech *3D Adventure 2*.

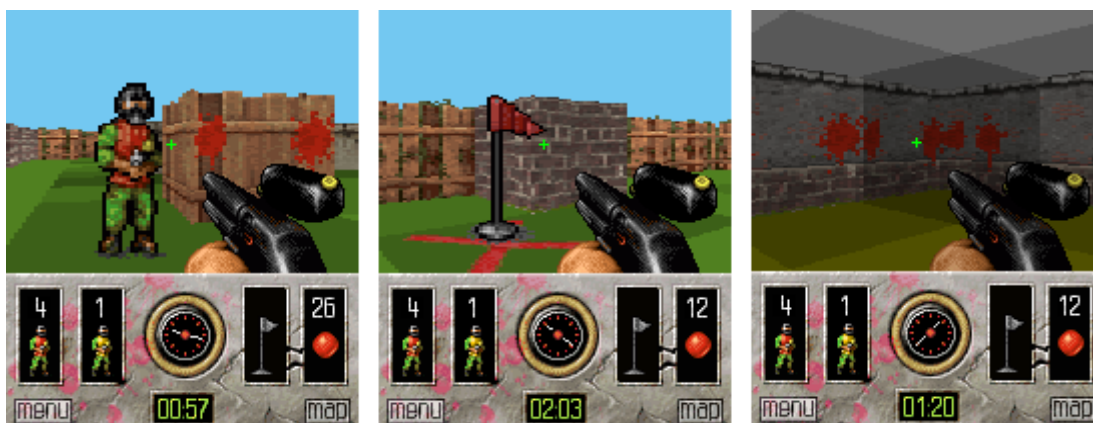
Když jsme před lety *3D Adventure 2* rozebírali, zjistili jsme, že navzdory velmi dobrému nápadu nebyly herní algoritmy dobře optimalizované. Hned ze začátku se nám podařilo dosáhnout několika zrychlení převodem násobení na postupná přičítání. Dále se daly aplikovat běžné javové rychlostní optimalizace (viz dále).

Hlavním rychlostním problémem bylo ovšem chování algoritmu, pokud se hráč dostal do příliš velké blízkosti k některé zdi. Původní algoritmus totiž nejdříve musel překreslit všechny pixely na obrazovce podlahami a stropem a poté přes tyto pixely ještě vykreslit zdi. Čím blíže byla zeď, tím více pixelů zdi bylo nutné přepočítat a překreslit. Pokud hráč stál těsně u zdi, mohla dokonce zeď pokrývat celou obrazovku a tedy pro každý pixel bylo nutné spočítat barvu podlahy i barvu zdi, i když pouze barva zdi byla viditelná!

Toto dvojí počítání celý engine podstatně zpomalovalo. Na první pohled to nemuselo ani být vidět, ale stačilo vstoupit do úzké chodby (stěny po obou stranách) a už bylo zpomalení silně znatelné. Při postavení čelem přímo proti zdi pak rychlost klesala i pod 1 FPS. A to mluvíme o rychlejších telefonech; například na Nokii 3100 byla rychlost tak malá, že hra byla naprosto nehratelná.

Po základních optimalizacích bylo nutné zaměřit se na kód vykreslování podlah a zajistit, aby přestal přepočítávat místa, která byla posléze zakrytá zdi. V principu šlo o velmi jednoduché vylepšení. Prvním krokem bylo upravit kód tak, aby se každý řádek obrazovky neprocházela nejprve doprava od středu a pak doleva od středu, ale místo toho lineárně zprava doleva. To znamenalo pouze spočítat pozici v pravém rohu a pak postupně odečítat vektor posunutí až do levého rohu. Druhým krokem pak byla úprava, aby se toto okno [levý roh, pravý roh] s postupujícími řádky mohlo zmenšovat. To lze do algoritmu implementovat velmi snadno. Pokaždé, když se při procházení řádku narazí na mapový čtverec obsahující zeď, otestuje se, zda je prozkoumávaný pixel jedním z rohů okna a pokud ano, daný roh se posune a okno se zmenší. Zrychlení tímto způsobem samozřejmě neúčinkuje vždy – nefunguje například v případech, kdy je vidět pouze jediná zeď uprostřed obrazovky. To ovšem není případ, který

by byl příliš častý. Typicky je vždy vidět zeď alespoň po jedné straně obrazovky a pak se pro každý sloupec zdi zredukuje prozkoumávání celého sloupce na prozkoumání jednoho jediného pixelu. Sice nezvětší nejlepší dosažitelné FPS engine, ale odstraní se případy extrémního zpomalování.



Obrázek 7: Paintball

Na některých telefonech (již zmíněna 3100) byla rychlost engine stále velice pomalá, což jsme vyřešili volitelnou nižší kvalitou grafiky – při zvolené nižší kvalitě byly pixely podlahy počítány jen pro každý lichý řádek a sudé řádky byly kopií řádků předchozích. Tento režim vylepšoval rychlost v průměru o 20-25%.

Paintball zaváděl do engine i další specifika – při vykreslování zdi se textura pokrývala ještě maskou se skvrnami od barvy. Masky se nastavovaly po každém zásahu zdi střelou. Viditelnějším rozdílem oproti předchozím hrám bylo to, že šlo o první hru, kde se hráč nepohyboval v uzavřených místnostech, nýbrž ve venkovním prostoru. Tento rozdíl se líbil většině tehdejších recenzentů, ale šlo pouze o jednoduchý trik – zatímco podlahy byly kresleny zelenou barvou (tráva), stropy byly kresleny nestínovanou modrou barvou (nebe). Vrchní část některých zdí pak byla také modrá a evokovala, že zeď je nižší než ostatní zdi. Samozřejmě však nebyla průhledná. Stejně tak i střední část obrazovky (obzor) musela být přebarvována neutrální barvou, pokud se v ní nezobrazovaly žádné zdi.

I když *Paintball* oproti *3D Adventure 2* výrazně vylepšil engine, hrátelnost byla mnohem nižší, především kvůli tristnímu game designu a špatné umělé inteligenci NPC.

3.2.6 Ostatní hry

Prozkoumali jsme ty nejzajímavější hry, samozřejmě to však nebyly všechny. Objevilo se několik klonů *3D Adventure*, např. *Stalk'n'shoot*. Ani jeden z těchto klonů však neměl dobře vyladěnou grafiku a často se vyznačovaly kulatými stěnami. Z toho lze vyvodit, že algoritmická část kódu byla částečně okopírována, ale ne zcela pochopena.

3D Adventure 2 žádné klony již neměla (s výjimkou *Paintballu*), ale na stejném engine byla autorem postavena nejméně jedna další hra lišící se pouze grafikou a level designem. Jedinou další 3D hrou na úrovni pixelů, o které víme, byl *Outbreak3D* z roku 2004. Ten však opět z velké části opět používal engine z *3D Adventure*, ale vylepšil fyzikální systém (nepotřeboval neprůchozí bloky v mapě), přidal jednoduchou orientační mapu a měl podstatně lepší ingame animace

Ostatní 3D hry byly spíše pseudo 3D. Pouze se snažily o vyvolání dojmu, že se hráč pohybuje ve třech rozměrech. Většinou šlo o závodní hry, kde byl 3D dojem vytvářen skládáním silnice z obdélníků a skokovým (nikoliv plynulým) zmenšováním vzdálených objektů.



Obrázek 8: Pseudo 3D hry

3.3 Hratelnost a prodejnost

3.3.1 Ovládání mobilních her a dopady na hratelnost

Než pokročíme k popisu nových 3D API, je zapotřebí zmínit ještě jedno specifikum mobilních telefonů, které má velký odraz na návrhu her a jejich výsledné hratelnosti, a tím je ovládání. Přizpůsobit vzhled hry na všechny možné typy telefonů je sice pracné, ale v podstatě to není nijak technicky náročné. Samotné ovládání hry se ovšem nesmí na různých modelech příliš lišit, takže musí být navrženo tak, aby ho bylo možné na většině telefonů bez problému implementovat.

Nejběžnější telefony mají 12 číselných tlačítek (10 číslic, hvězdička, křížek), dvě softwarové klávesy (Potvrdit a Zpět) a směrové klávesy (čtyřsměrný joystick) se středním potvrzovacím tlačítkem (stisknutí joysticku). Takový je ideální případ, se kterým počítá většina her. Většina telefonů má kromě toho další tlačítka, která nejsou z Javy ovladatelná a bývá na nich namapována některá systémová funkce (zelené a červené tlačítko, tlačítko menu na Symbianech, ovládání hlasitosti apod.). Naopak některá mohou být z Javy ovladatelná – třetí softwarové tlačítko, tlačítko pro mazání textu „C“. Kdyby se dalo pracovat pouze s tímto jednoduchým modelem, situace by byla ještě poměrně jednoduchá.

Klávesy se rozlišují zprvu podle číselného kódu, který je zpravidla číselnou reprezentací znaku dané klávesy (resp. prvním zobrazeným znakem, pokud je jich možno klávesou napsat více), za druhé podle tzv. herních akcí (*game actions*), což jsou kódy přidělené výrobcem telefonu některým klávesám. Jsou to kódy *RIGHT*, *LEFT*, *UP*, *DOWN*, *FIRE*, *GAME_A*, *GAME_B*, *GAME_C* a *GAME_D*.

V praxi bohužel nelze obvykle pro ovládání hry použít číselné kódy kláves, protože klávesy nebývají rozloženy stejným způsobem. Ve většině případů sice telefon má typické rozložení numerických kláves, ale najdou se výjimky. Klasickým případem je jeden z prvních Symbianových telefonů Nokia 3650 s klávesnicí rozloženou do kruhu.

Dále jsou tu ovšem i telefony s úplnou písmennou klávesnicí (případně s klávesnicí spojující dva znaky na klávesu místo typických tří). Zde už kódem klávesy nebývá znak 0-9, ale právě znak písmena na klávese. Pak je možné nějakou herní akci implementovat například jako reakci na stisknutí klávesy „z“. Ale bohužel poloha tohoto písmene se liší podle národního provedení daného zařízení – je odlišná na mezinárodní QWERTY klávesnici, německé QWERTZ nebo francouzské AZERTY. Dalším problémem jsou tzv. zasouvací (*sliding*) telefony, které umožňují zasunout větší část klávesnice pod displej telefonu, takže je dostupný pouze joystick. Distributoři obvykle požadují, aby hra byla ovladatelná i v tomto módu. Samostatnou kapitolou jsou pak telefony s dotykovým ovládním. Ty obvykle (ne vždy) samy do javových aplikací vkládají virtuální dotykovou klávesnici, ale ta opět zobrazuje v defaultním nastavení pouze joystick a softwarové klávesy.

Z těchto důvodů se ovládní většiny her omezuje pouze na čtyři směrové klávesy, středové potvrzovací tlačítko (*FIRE*) a dvě softwarové klávesy. V některých speciálních situacích, kdy tyto klávesy nestačí, je možné použít i některé další, ale většinou to znamená preprocesorové větvení konstant kláves pro různé telefony a poloha těchto kláves nesmí mít kritický vliv na hratelnost. Klávesy mimo joystick bývají obvykle mapovány podle defaultních herních akcí (viz výše) tak, aby měly shodnou funkci jako joystick (typicky 2 – nahoru, 4 – doleva, 6 – doprava, 8 – dolů, 5 – výstřel).

Na hrách, které zkoumáme, můžeme toto omezení vidět například v tom, že v podstatě neproběhly žádné pokusy o ovládní zorného úhlu kamery ve svislé ose. Čtyřsměrný kurzor umožňuje efektivně implementovat pouze chození a otáčení v horizontální rovině. V pozdějších hrách na takové pokusy ještě narazíme, ale ovládní nikdy není příliš pohodlné. Omezení lze obejít například tak, že se hra rozděluje do „chodícího“ a „bojujícího“ módu, kdy klávesy reagují odlišným způsobem. V jedné 3D hře je zase umožněno i chození po kolmých stěnách, čímž se úhel kamery efektivně mění při chůzi. Do budoucna však není jednoduché ovládní trvale udržitelné, protože hry jsou čím dál tím složitější a potřebují ovládat více herních akcí.

Hráče PC her samozřejmě napadne, proč neumožnit nastavení ovládacích kláves přímo v menu hry, aby si každý hráč mohl ovládní upravit podle klávesnice na svém zařízení a podle svých zvyků. To je přirozeně možné a je to i jednoduše implementovatelné. Bohužel to odporuje celkovému trendu mobilních her, které jsou v drtivé většině případů navrženy jako tzv. *casual* hry. To znamená, že hráč chce po prvním spuštění co nejdříve začít hrát a hra tedy musí mít už na začátku nastaveno ovládní tak, aby se dala pohodlně hrát. Stejný trend se pak přenáší i do *long-play* her.

Mluvíme tu ale spíše o novějších hrách. Když se vrátíme ke *3D Adventure*, zjistíme, že největším problémem hratelnosti nebylo rozložení kláves. Hra byla přece určena jen pro několik málo telefonů a variabilita typů klávesnic byla v té době nízká, takže nebylo problémem vyladit ovládní zvlášť na každém cílovém zařízení (šlo v podstatě jen o dvě verze – Nokia 3650 a ostatní). Problémem byla spíše jedna z nemilých vlastností hardwaru tehdejších telefonů a to byla ta, že klávesnice nedokázala registrovat stisk více kláves najednou. Hra potřebuje informaci o stisknutí a uvolnění klávesy, ale jakmile byla jedna klávesa stisknuta, ostatní byly mrtvé, dokud nebyla první klávesa uvolněna. Pro *3D Adventure* to znamenalo, že hráč se například nemohl zároveň pohybovat a střílet, což bylo velmi nepohodlné a pro hráče obtížně pochopitelné.

Tato vlastnost hardwaru není tak překvapivá. Staré mobilní telefony bez aplikační podpory nepotřebovaly stisk více kláves najednou – a autoři původní specifikace MIDP 1.0 si to zřejmě neuvědomili a požadavek současného stisku více kláves nebyl do specifikace přidán. Upřesněno to bylo až v MIDP 2.0.

Dnes se tento problém znovu opakuje při implementaci ovládání na dotykových displejích. Rezistentní displeje *multitouch* podporovat nemohou vůbec, kapacitní sice mohou, ale ne vždy je to implementováno. Ovládání na virtuální klávesnici nebo konzoli pak nutí hráče provádět vždy jen jednu herní akci zároveň.

3.3.2 Hratelnost mobilních her a důsledky na prodejnost

Přestože mobilní hry měly už od počátku svojí existence problémy s hratelností – ať už kvůli špatnému ovládání, trhané grafice, pomalosti, nebo prostě špatnému návrhu, přesto mohly být velice úspěšné.

Poměrně brzy se totiž ukázalo, že hratelnost není při prodeji rozhodující, pokud si hráč nemůže před koupí hru vyzkoušet. Zde se opět projevuje efekt *casual her*. Když lze celou hru dohrát během jedné půlhodiny, nelze vytvořit hratelné demo, které by hráče motivovalo ke koupi plné verze hry. To, co prodává mobilní hru je tedy v první řadě jméno (resp. známý brand, tj. značka, kterou lidi znají) a na druhém místě grafický vzhled, který je možné vidět na náhledech ze hry. Dnes se postupně prosazují i videa ze hry, které dávají potenciálnímu kupci ještě lepší informaci o hře. Jak u náhledů, tak u videa jde ale vždy pouze o záběry z emulace hry na PC, tedy jak by vypadala na dokonalém telefonu - uživatel tedy nikdy nezjistí nic o tom, jak bude vypadat na jeho pomalejším telefonu s menším rozlišením. Paradoxně tak často hráči kupují něco odlišného, než si myslí, že kupují. Může to jít až do extrému. Jestliže chce vývojářské studio vytvořit nějakou pokročilejší hru (zejména třeba při použití 3D), nemůže ji vydat pouze pro telefony, na kterých funguje dobře. Mezi hlavní distribuční kanály patří prodejní weby mobilních operátorů a ty mají seznamy telefonů, pro které hra musí být portována, jinak se vůbec o jejím vydání nezačne uvažovat (tzv. *handset list*). Na těchto seznamech však nejsou jen ty nejnovější telefony, ale bohužel i ty nejstarší.

Při vývoji technicky pokročilé hry se tedy musí udělat nejméně dvě verze, z nichž jedna poběží i na slabších telefonech. Prakticky se však vždy dělá ještě tzv. *tiny* verze, která je určena pro staré slabé telefony (např. pro ty, pro které byla určena *3D Adventure*). Ta bývá velice jednoduchá, ve stylu jednoduchých skákaček a logických hříček. Finančně totiž není únosné dělat další verzi kódu a grafiky pro slabé telefony, když mají na trhu podíl jen několika málo procent. Vývojářské firmy si postupně vytvářejí několik enginů pro tyto *tiny* hry a v případě potřeby pouze vloží novou zjednodušenou grafiku a nové levely. Tyto verze nejsou vlastně určeny k prodeji, ale musí být k dispozici, aby hra mohla být vůbec vydána. Bohužel se pak objevují případy, kdy si hráč v dobré víře koupí hru, jejíž náhledy se mu líbí, a dostane jednoduchou hříčku za cenu plné hry. Distributoři zpravidla po stížnosti vrací peníze, ale jde o minoritní prodejní podíl.

Teoreticky ani není složité uživateli hru předvést dopředu. Sami jsme na objednávku implementovali emulátor mobilních her pro webové stránky, na kterých si hráč mohl hru dopředu vyzkoušet a pak se teprve rozhodnout. Emulace končila po předem nastaveném časovém intervalu (např. jedna minuta) a šlo tedy skutečně pouze o vyzkoušení, kdy uživatel mohl vidět hru, ale nemohl si jí skutečně zahrát. Překvapivě však o tento produkt nebyl

ze strany distributorů a prodejců zájem. Argumentovali tím, že by jim to snížilo prodejnost. Na trhu mobilních her v současnosti na hratelnosti prostě nezáleží. Proto jsou na špičkách žebříčku stahovanosti hry podle filmů, knížek, případně předělávky PC her. Jsou to známé brandy. Je třeba také zmínit, že žebříčky stahovanosti her zobrazené na webových stránkách mobilních operátorů nejsou pro statistiky stahování určující, protože místa na nich jsou upravována podle marketingové strategie při nasazování hry do prodeje – tj. distributor si může místo na žebříčku na určitou dobu zaplatit. Pokud jde o důležité prodejní místo, kde se pozice na žebříčku nedá přímo zakoupit, bohatší vývojářská studia neváhají samy zaplatit a stáhnout několik tisíc kusů hry, aby se na žebříček dostala. Tento trend můžeme snadno vidět například u nových iPhone her, kde existuje pouze jediný prodejní server s omezenými možnostmi vyhledávání a pozice na žebříčku stahovanosti je tak pro prodej klíčová.

Do budoucna se ale bude význam hratelnosti mobilních her nepochybně zvyšovat. Se stále lepším výkonem mobilních telefonů se objevují i *long play* hry (např. titul *Galaxy on Fire* od Fishlabs) s propracovaným příběhem. Vzhledem k dražšímu vývoji budou tyto hry prodávány za vyšší cenu a hráči se budou více rozmyšlet než si takovou hru koupí. Větší vliv na rozhodování tak získají recenze a demoverze, tj. samotná hratelnost hry.

Dnes se hratelnost hry na prodeji projevuje spíše druhotně. Nemá vliv na prodej hry samé, ale může ovlivnit povědomí hráčů, kteří pak budou mít větší zájem koupit si další díl té samé hry, případně jinou hru od stejné vývojářské společnosti. Před několika lety se například na prvním místě v prodeji držela série her *Yeti Sports* – ač šlo v podstatě o primitivní hry, byly velice zábavné a hráči si nadšeně kupovali další díly - v poměrně krátké době jich vyšlo sedm. Jiným případem je třeba strategie *Townsmen*, která se dočkala již pátého dílu. Vývojářská společnost, která díky svým hrám získala dobré jméno a dobře prodává skoro cokoli, je například Fishlabs. Naproti tomu třeba společnost Gameloft v počátku své existence získala dobré jméno díky kvalitně řemeslně zpracovaným hrám, ale v poslední době vydává tuctové hry stavěné na stejném enginu, takže hráčovi nepřináší prakticky nic nového a prodejnost se snižuje. Díky dobrému jménu je však neustále vyšší než průměrná.

Pozice společnosti Fishlabs [10] je nyní natolik dobrá, že se odvážili opustit model prodávání her přes distributory a jejich nejnovější titul *Galaxy on Fire 2* je volně stažitelný z jejich webových stránek, ale část hry je zamknutá, dokud hráč nezaplatí pomocí premium sms, které jsou posílány přímo ze hry.

3.4 Optimalizace

Než se podíváme na tvorbu 3D her na modernějších telefonech, zmíníme se o optimalizačních tricích, které se ve starších hrách používaly. Je důležité je znát, protože existují mnohé případy, kdy je nutné podobné triky použít i na moderních zařízeních.

Paměťové optimalizace se týkaly především celkové velikosti aplikace a částečně už jsme se o nich zmínili v úvodu. Jejich podstatou bylo zhuštění kódu a zdrojových dat tak, aby byly co nejlépe zkomprimovány v zipovém archívu. U kódu šlo tedy především o zmenšení počtu tříd (tedy počtu souborů v archívu). Například hra *Paintball* měla pouhé tři třídy - hlavní třídu aplikace, třídu herní obrazovky obsahující zároveň veškerý kód hry a třetí třída řešila ukládání perzistentních dat a internetovou komunikaci (bylo možné stahovat nové mapy z internetu).

Speciální kapitolou je zmenšování velikosti grafických dat. Novější zařízení běžně zvládají grafické formáty JPEG a PNG, ale PNG je kvůli lepší přenositelnosti stále de facto

standardem pro data obrázků. JPEG se kvůli menší podpoře a ztrátové kompresi používá jen výjimečně. Otázkou tedy je, jak grafická data PNG obrázků uložit co nejúsporněji.

Prvním trikem je sloučení menších obrázků do jednoho většího obrázku – typicky se používá pro fragmenty 2D spritů nebo pro dlaždice pozadí. Rozřezat načtený obrázek za běhu na jednotlivé menší obrázky, nebo vykreslit jen část obrázku už pak není problém. Velikost dat je zmenšena tím, že místo mnoha pomocných PNG dat v jednotlivých souborech budeme mít tato data pouze jednou. To se velmi projeví zejména u velkého množství malých obrázků, kdy pomocná data (hlavičky, kontrolní součty apod.) mohou zvětšit grafická data až o 50%. Pokud jsou navíc obrázky paletované, paleta bude uložena také jen jednou.

Při dobré znalosti PNG formátu se dají uplatnit i pokročilejší triky. V mnoha hrách najdeme shodné obrázky lišící se pouze použitými barvami (tj. paletou). Zdroje se pak dají zmenšit tak, že jeden obrázek uložíme celý a u ostatních pouze paletu (*PLTE chunk*, [11]). Tím se lehce zvýší náročnost načítání, protože je nutné nejdříve zkombinovat paletu s ostatními daty, ale vzhledem ke zmenšení celkové velikosti aplikace se to jistě vyplatí.

Na všechny výsledné obrázky je dobré ještě použít nástroj *pngout* [12], který odstraní všechny nepotřebné PNG *chunks* (např. textová data) a vybere nejlepší možný způsob uložení, aby byl výsledný soubor co nejmenší.

Na starších Nokia telefonech se používal ještě úplně jiný způsob uložení grafických dat a to uložení přímo jako binární pole barev jednotlivých pixelů. Oproti uložení pomocí PNG měl výhodu, že mohl využít nativní dvoubytový barevný formát (displej s 4096 barvami, 4 bity na každou komponentu) a barvy se při načítání nedeformovaly přepočtem ze čtyřbytového formátu, takže obrázky zůstaly takové, jak je grafik navrhl. Vzhledem k tomu, že Java archiv i PNG používají ZIP kompresi, vlastně se jen odstranila jedna přebytečná komprese. Úspora dat oproti dobře uloženým PNG obrázkům byla malá až žádná, ale v té době ještě nebyly k dispozici nástroje typu *pngout*, takže získání optimálně uložených souborů bylo obtížné.

Podobně se řeší všechny ostatní typy dat – např. všechny zvuky se sloučí do jednoho souboru, všechny data úrovní do jednoho souboru apod. Kromě zmenšení aplikace je to výhodné i kvůli tomu, aby data nešly z aplikace dostat zpět. Některé vývojářské společnosti data i jednoduše kódují, ale to je podle našeho názoru zbytečné, protože není těžké najít a dekompileovat třídu, která dekódování zajišťuje.

Tímto jsme probrali postupy, jak zmenšit celkovou velikost aplikace. Mezi paměťové optimalizace zaměřené na runtime paměť patří například důsledné využívání polí oproti spojovým seznamům. Nejdůležitější je však hlídat alokace velkých objektů. Pokud se najednou alokuje velké množství paměti, zpravidla to má za následek časté volání garbage collectoru a tedy „sekání“ aplikace. Za běhu hry je proto nutné vyhnout se jakýmkoliv větším alokacím a vše načítat před samotným spuštěním hry nebo herního levelu. Zkušenost ukazuje, že hráč spíše přetrpí delší čekání před spuštěním hry než špatnou plynulost hry.

Důležitější jsou ovšem optimalizace kódu na rychlost. V dnešní době už tyto optimalizace nejsou natolik podstatné, protože všechny novější telefony už v sobě mají JIT překladače, takže se optimalizacemi zabýváme spíše z historických důvodů. Mnohé starší postupy jsou dnes už také zakomponovány do optimalizační procedury kvalitních obfuskátorů, takže je není nutné provádět ručně. Některými úpravami je však možné dosáhnout zrychlení i dnes, ovšem ve speciálních případech. Rychlost je stále mnohem výraznějším faktorem než

nedostatek paměti, takže se vyplatí během načítání vygenerovat do paměti struktury (např. vyhledávací struktury, předkreslené obrázky), které později zvýší rychlost výpočtů nebo vykreslování.

Klasickým zrychlovacím postupem je zmenšení počtu volání metod. V mobilních hrách byla tato technika prováděna ad absurdum. Každé zbytečné volání bylo třeba odstranit i za cenu velkého znepráhlednění kódu. Herní třída měla zpravidla jen dvě velmi dlouhé metody – vykreslení a update hry. Důležité bylo nevolat žádnou metodu uvnitř cyklů s mnoha iteracemi, což způsobovalo hluboce zanořené kódy. Všechny proměnné byly zpravidla veřejné, aby k nim bylo možné přistupovat přímo bez použití getterů a setterů (dnes už getters odstraňuje přímo obfuskátor). Výhodu měli pouze programátoři používající pro vývoj preprocesor jazyka C, protože ten dokáže metody inlinovat sám. Problémem bylo, když v cyklu byla potřeba volat některou z knihovnických metod. V tu chvíli bylo nutné knihovnickou metodu implementovat ručně. Mezi takové metody patří například generování náhodného čísla (v knihovnické implementaci jde o dvouřádkový algoritmus) nebo výpočet absolutní hodnoty.

Dnes už naštěstí existuje pro J2ME profiler [13], takže lze odhalit úzká hrdla v kódu. Jde zatím o poměrně novou záležitost navázanou na emulátor a nedává přesné výsledky oproti reálným zařízením. I tak ovšem praxe ukazuje, že úzké hrdlo nalezené pomocí profileru bývá úzkým hrdlem i na zařízeních. Zpravidla jde o opakující se kód v dlouhých cyklech, tedy ve výpočetních částech kódu. Úzká hrdla ve vykreslování se většinou týkají přímo jednotlivých implementací na reálných zařízeních, takže profiler je obvykle nenajde. Pokud je výpočetní část příliš náročná (např. u 3D her), ve většině případů ke zrychlení pomůže jakákoliv, i malá, optimalizace, protože výpočetní kód se opakuje v každém herním cyklu.

Při optimalizacích je nutné ovšem splnit základní pravidlo – myslet na optimalizace už během vývoje, ale provádět je až poté, co je kód odladěn a spolehlivě funguje. Každá optimalizace totiž znepráhledňuje kód a jen stěží se v něm pak hledají chyby. Nejvíce je to vidět například po ručním inlinování metod.

Na novějších telefonech je inlinování nutné použít například v situaci, kdy je v cyklu potřeba zaokrouhlení reálného čísla. V tu chvíli je lepší reálná čísla nahradit celými čísly s fixní desetinnou čárkou a zaokrouhlení provádět bitovými operacemi. Při iteraci přes každý pixel obrazovky pak zrychlení dosahuje i stovek procent. Dalším typickým inlinováním objektů je odstranění javového dynamického pole (*java.util.Vector*). Ačkoliv z hlediska rychlosti nevyhází vůbec špatně, jeho problémem je, že přístup k položkám sebou nese režii volání metod, takže se mu snažíme vyhnout v cyklech. Jelikož se tato struktura často používá pro implementaci zásobníků a front, nahrazujeme ji polem o dostatečné velikosti.

Dalšími známými technikami je nepoužívání některých javových jazykových struktur. Jde především o synchronizované bloky kódu a zachytávání výjimek. Každý průchod těmito strukturami na starých telefonech trval od 20 do 100ms.

Ostatní zrychlovací postupy [14] už mají vliv podstatně menší, ale když jsou použity všechny, zrychlí kód o 5% až 10% podle zařízení. Zdá se to málo, ale při měření FPS to znamená, že z průměrných 8-10 se počet snímků za sekundu zvětší o 1-3 podle zařízení (měřeno na telefonech Nokia a Sony-Ericsson při vývoji hry *Paintball*, FPS nemusí být přímo úměrné času výpočtů). Mezi tyto techniky patří následující:

1. Procházení pole odzadu. Srovnáme-li kód:

```
for (int i = 0; i < size; i++)
```

a

```
for (int i = size - 1; i >= 0; i--)
```

je zřejmé, že v druhém případě se při každé iteraci ušetří načítání hodnoty proměnné (konstanty). Navíc javový bytekód umožňuje provést porovnání s nulou pomocí jedné instrukce, zatímco na porovnání s číslem jsou potřeba instrukce tři.

2. Přístup k lokální proměnné místo přístupu ke globální proměnné. Například pokud v cyklu přistupujeme ke globálnímu bufferu pixelů:

```
short[] screenBuffer = new short[width * height];

public void update() {
    for (int i = screenBuffer.length; i >= 0; i--) {
        screenBuffer[i] = ...
    }
}
```

Lze dosáhnout zrychlení v řádu jednotek procent, pokud na začátek metody přidáme kód

```
short[] screenBuffer = this.screenBuffer;
```

Klíčem k tomuto zrychlení je pochopení, jak virtuální stroj přistupuje k lokální a globální proměnné. Pro přístup ke globální proměnné je potřeba instrukcí nejprve načíst identifikátor proměnné a tímto identifikátorem se pak indexuje hašovací tabulka s adresami hodnot proměnných. Lokální proměnná je naproti tomu jen anonymní číselný index, kterým se indexuje zásobník metody – jde tedy vlastně jen o přístup k položce pole. Je zřejmé, že po nahrazení globální proměnné lokálním přístupem se instrukce začnou provádět přes jednodušší, tedy rychlejší datovou strukturu. Tato technika se využívá hlavně na starších telefonech; na novějších už nepřináší žádné zrychlení, protože JIT pravděpodobně během iterace adresu proměnné kešuje. Naproti tomu se však kód určitě nezpomalí.

3. Zjednodušení primitivních operací:

Dnešní komerční benchmarky ([15]) dokáží o mobilních zařízeních zjistit mnoho zajímavých informací. Je zřejmé, že odstranění násobení a dělení pomocí sčítání zrychlí kód vždy. Většinou však zrychlí kód i převedení dělení na násobení u čísel s plovoucí řádovou čárkou. Dělení je totiž jednou z nejpomalejších operací vůbec. Překvapivější je ovšem zjištění, že přístup k položce pole je zhruba čtyřikrát pomalejší než celočíselné násobení, takže se vyplatí uložit si položku pole do lokální proměnné, než ji začneme zpracovávat. Zde je důvodem javový test pro překročení limitů pole, který se provádí při každém indexování.

4. Odstranění polymorfismu:

Statické metody jsou vždy rychlejší než virtuální metody. Pokud dále máme například třídu A a třídu B, která je potomkem A, je dvakrát rychlejší volat virtuální metody přímo na typu B, než na typu A.

5. Odstranění ladícího kódu:

Během vývoje a ladění se do kódu přidává mnoho příkazů, které nejsou pro odladěný kód nezbytné. Mezi ně patří například ladící výpisy, *asserts*, kontroly správnosti parametrů metod apod. Ve výsledném kódu by se tyto věci neměly nikdy objevit, proto i v Javě je dobrým zvykem obklopit je direktivami preprocesoru, aby se v distribučním balíku neobjevily.

Kapitola 4

3D knihovny na mobilních telefonech

4.1 Úvod

Od doby, kdy byly do mobilních telefonů přidány 3D knihovny, se už v podstatě nedělají hry na úrovni pixelů. Vývoj takové hry je totiž náročný a vývojáři vždy stojí před problémem, jak hru implementovat na telefonech s pomalým vykreslováním pixelů na displej. Použití 3D knihoven je mnohem univerzálnější. Ještě před pár lety je mělo jen několik telefonů, ale dnes už stěží najdeme zařízení, které je nemá.

Samozřejmě i tak je poměrně obtížné 3D hru implementovat. Stále zde zůstává problém, jak vytvořit portace her pro staré, málo používané telefony. Obvykle je pak nutné vytvořit verzi bez 3D a s 3D. Problematická je i rychlost těchto knihoven, která může záviset například na velikosti použitých textur nebo na množství světel ve scéně, případně na velikosti celé scény. Bývá proto nutné opět i zdrojové soubory (levely, scény, textury) upravit pro každou jednotlivou portaci. Jednou z univerzálních výhod 3D knihoven je skutečnost, že při jejich implementaci musí zařízení splňovat i specifikaci CLDC 1.1, tj. v API jsou dostupná čísla s pohyblivou řádovou čárkou, goniometrické funkce a druhá odmocnina.

Přítomnost 3D knihoven lze ovšem využít i jinak než ve 3D hrách. Existuje několik způsobů, jak lze pomocí nich zvýšit výkon nebo vzhled 2D her. Na tyto techniky se také podíváme.

4.2 Mascot Capsule v3

Mascot Capsule v3 je kódový název pro implementaci 3D knihovny dostupnou na telefonech značky Sony-Ericsson, Sprint a několika málo dalších. Tuto knihovnu vyvinula japonská společnost HI CORPORATION na objednávku výrobců mobilních telefonů [16].

Implementace je založena na dvou formátech datových souborů pro grafická data a na několika třídách nutných pro jejich zobrazení. Oba soubory jsou textové – jejich formát by se dal přirovnat k velmi zjednodušenému VRML. Jeden z formátů (*BAC6*) slouží pro reprezentaci 3D objektů – vrcholy, normály, barvy, polygony. Tyto objekty jsou oproti dalším 3D knihovnám opravdu velmi jednoduché – materiály jsou například určeny pouze difuzní barvou a jedním desetinným číslem, které určuje, jak moc se světlo od objektu odráží. Objekty nejsou propojené v jednom stromu scény, ale je zde možnost vytvořit kostěné modely, tj. scéna je vlastně les stromů, kde každý uzel má svou transformační matici a 3D objekt (resp. pole polygonů).

Druhý souborový formát (*TRA4*) pak definuje animace na kostěných modelech jako změny jednotlivých transformačních maticí snímek po snímku.

Samotná knihovna se dá jen stěží použít pro vytvoření nových dat – sestavování jednotlivých datových příkazů by bylo příliš obtížné. Knihovna se skládá z pouhých 10 tříd. Z toho jedna třída obsahuje jen tabulky celočíselných hodnot pro sinus a kosinus (4096 úhlů) a výpočet druhé odmocniny pro celá čísla (*Util3D*). Další třída je vektor (*Vector3D*) se standardními vektorovými operacemi (skalární i vektorový součin, normalizace). Dále je zde třída reprezentující matici s afinními transformacemi (*AffineTrans*). Jednotlivé třídy se dají použít

vlastně jen na načtení a zobrazení objektů, vytvoření světel, změnu polohy kamery a přepínání animací.

Práce s touto knihovnou je obtížná, ale její výhodou je, díky nízkourovňovosti, dobrá zobrazovací rychlost. Problémem je však její malé rozšíření na mobilních telefonech, takže často můžeme najít hry, které jsou pro většinu telefonů implementované pomocí M3G (viz dále), ale pro telefony s Mascot Capsule v3 je vytvořena speciální verze za použití této knihovny. S postupně se zlepšující rychlostí telefonů to však přestává být nutné. M3G má navíc podstatně širší možnosti, takže přechod na Mascot Capsule v3 znamená pro hru vždy podstatná omezení, případně nutnost některé funkce implementovat ručně. Mnoho z hotových implementací lze však snadno dohledat na internetu (například kolize pomocí paprsků v M3G pomocí MCv3 – viz dále, [17]).

4.3 Mobile 3D Graphics (M3G)

4.3.1 Požadavky na specifikaci

Tato knihovna je známa též pod označením své specifikace JSR-184. Dnes používaná verze 1.1 vznikla v roce 2005 jako společná práce expertní skupiny složené z hlavních výrobců mobilních telefonů a společnosti Sun pod vedením společnosti Nokia. Mezi hlavní požadavky, které měla tato specifikace splňovat, patří následující [18]:

- Musí podporovat zobrazení grafu scény (*retained mode*).
- Musí podporovat přímé zobrazení jednotlivých objektů nezařazených do scény (*immediate mode*).
- Oba způsoby zobrazování je možné směřovat a používat nezávisle na sobě.
- Po zkušenostech s fragmentací knihoven na mobilních telefonech bylo rozhodnuto, že API nesmí obsahovat žádné volitelné části, ale všechny musí být pokaždé podporovány (což se ne zcela povedlo).
- Musí existovat možnost, jak importovat celý graf scény najednou, ale zároveň musí existovat možnost, jak jednotlivé objekty vytvářet za běhu. Graf scény musí být uložen v binární formě pro usnadnění implementace načítání a pro zmenšení datových souborů.
- API musí být efektivně implementovatelné nad OpenGL ES [21]. Tento požadavek velmi ovlivnil celkovou strukturu knihovny.
- Pro operace s desetinnými čísly nesmí zavádět nový datový typ, ale musí používat nativní jazykové typy *float* a *double*. Je však třeba, aby efektivně pracovala i na přístrojích bez hardwarové podpory desetinných operací, proto by měla být co nejvíce založena na celých číslech.
- Knihovna musí být na mobilním zařízení efektivně implementovatelná na 150 KB paměti. Během zobrazování pak musí být minimalizovaná práce garbage collectoru, tedy alokování nových objektů musí být co nejvíce omezeno.

- Knihovna musí být dobře propojitelná s jinými mobilními grafickými knihovnami, zejména s MIDP (hlavně s třídou *Graphics*) a AWT (používané na PDA telefonech).
- Knihovna musí být dobře škálovatelná nejen na zařízení se slabým výkonem, ale i na telefony s hardwarovou podporou operací s desetinnými čísly a případně i s grafickými čipem.



Obrázek 9: Hra postavená na M3G API

4.3.2 Strom scény a picking

Specifikace obsahuje možnost vytvářet graf scény – je to zajištěno zejména třídami *Node* a *Group*. *Node* je nadtřídou všech uzlů v grafu – obsahuje hlavně informaci o uzlu předka a lokální transformační matici, ale i několik dalších užitečných společných vlastností. Uzlu lze například nastavit automatickou orientaci transformační matice směrem k jednomu nebo dvěma jiným referenčním uzlům. K jedné zvolené ose referenčního uzlu lze orientovat osu *z*, k jedné ose druhého referenčního uzlu pak orientovat osu *y*. Po zavolání metody *align* na uzlu se pak automaticky nastaví orientace všech uzlů v podgrafu. Tato funkce je velmi užitečná zejména při natáčení objektů směrem ke kameře.

Každý uzel má dále *alpha factor*, který násobí *alpha* kanál difuzní barvy všech materiálů a pixely textur v podgrafu. Při vhodném nastavení blendingu ho tak lze využít třeba pro postupné mizení nebo objevování objektů. Každý uzel má také přepínače pro úplné vypnutí zobrazení a aktivace *picking*.

Nejzajímavější vlastností uzlů je však možnost definovat tzv. *scope*. Jde o 32-bitovou masku, která umožňuje sdružování objektů mimo strom grafu. Pokud dva objekty mají v masce alespoň jeden bit společný, pak jsou ve stejné *scope* skupině. Toto seskupování lze pak využít při zobrazování – pokud je tato maska nastavena kameře (třída *Camera*), kamera zobrazuje pouze objekty ve své skupině. Stejně tak světla (třída *Light*) osvětlují pouze objekty ve stejné skupině. Omezení *scope* lze využít i pro *picking*, čímž se výrazně zrychlí.

Pro strom scény je pak důležitá také třída *Group*, která jako jediná může být rodičem pro uzly grafu. Na každém podgrafu definovaném instancí této třídy lze pak provést *picking*. V M3G se *picking* provádí jako vyslání paprsku z bodu v prostoru v určeném směru a s určenou *scope*, případně jako vyslání paprsku z roviny kamery. Pokud paprsek narazí na nějaký zobrazený objekt, programátor získá cílový uzel, vzdálenost od počátku paprsku a normálu a souřadnice textury v bodě dotyku paprsku.

Využití *pickingu* v M3G bylo původně zamýšleno, mimo jiné, pro snadnou implementaci kolizí, a v mnoha hrách tak bylo i používáno. Jakmile je však hra složitější (např. *first person shooter*), implementace kolizí pomocí paprsků se jeví jako velmi problematická. Později se k tomu ještě vrátíme.

Speciální podtřídou *Group* je třída *World*, která slouží jako kořen grafu scény a kromě poduzlů ke grafu připojuje také instanci třídy *Background*, jež má v sobě informaci o barvě či obrázku na pozadí scény.

4.3.3 Typy uzlů

Dalšími uzly v grafu jsou instance třídy *Camera*. Kamera obsahuje pouze 4x4 projekční matici a metody pro její nastavení (speciálně pro paralelní a perspektivní kameru). Pozice kamery pak určuje výhled avatara. V *retained mode* lze přepínat aktivní kameru mezi kamerami v grafu scény, v *immediate mode* je pak nutné přímo předat kameru, která bude pro zobrazení použita. V obou módech kamery zobrazují pouze uzly ve stejné *scope*.

Osvětlení scény je určeno instancemi třídy *Light*. Naproti Mascot Capsule v3, které podporuje jen směrové (*directional*) a ambientní (*ambient*) světlo, M3G podporuje také všesměrné (*omni*) a kuželové (*spot*). Jejich výpočetní náročnost je už vyšší, hlavně kvůli postupnému zeslabování intenzity (*attenuation*). Pro M3G je kritickým parametrem počet aktivních světel ve scéně. Každé další světlo přirozeně zvyšuje složitost zobrazovacích výpočtů, kterých na mobilních telefonech nesmí být příliš mnoho. Proto M3G zavádí i omezení na maximální počet světel platných zároveň pro jeden 3D objekt. Na zařízeních je tato hodnota obvykle nastavena na 2 nebo 4 světla. Použití více světel nemá za následek chybu, ale implementace si může jakoukoliv heuristikou vybrat, která světla nepoužije.

Dalšími potomky třídy *Node* jsou už dále pouze samotné 3D objekty. Ty se dělí do dvou hlavních podtypů. Za prvé jsou to instance třídy *Sprite3D*. Jak už název napovídá, instance této třídy jsou v podstatě jen 2D obrázky (*sprites*), které se ovšem automaticky natáčejí směrem k aktivní kameře. Prakticky lze takové objekty vytvořit i bez použití *Sprite3D*, ale je to složitější technicky, neboť je nutné vytvořit geometrii, na kterou bude promítnuta textura, definovat správně souřadnice textury, ale také po každém pohybu kamery upravovat natočení výsledného objektu směrem ke kameře (např. pomocí již zmíněné metody *align*). *Sprite3D* toto vše dělá automaticky. Na rozdíl od jinak vytvořených objektů dokáže navíc správně vyřešit chování průhledných pixelů obrázku. Tj. pokud je např. vyslán *picking* paprsek, tak se zastaví jen na neprůhledných pixelech a na průhledných projde skrz.

Všechny další typy 3D objektů jsou reprezentovány třídou *Mesh* a jejími potomky. Základní typ spojuje pouze geometrii a vzhled objektu. Geometrie je zadána instancemi dvou objektů – *VertexBuffer* a *IndexBuffer*. Jak již názvy napovídají, *VertexBuffer* v sobě obsahuje jednotlivé vrcholy a ke každému z nich normálu a případné souřadnice textury – každý typ dat je uložen zvlášť jako instance třídy *VertexArray*. *IndexBuffer* pak určuje způsob, jakým se z vrcholů sestavují polygony. Oddělení pozic vrcholů od indexů usnadňuje znovupoužití dat mezi různými 3D objekty.

IndexBuffer je abstraktní třída. Ve verzi 1.1 specifikace M3G (aktuálně používaná) existuje pouze jedna její implementace a tou je *TriangleStripArray* umožňující vytváření trojúhelníků ze série indexů vrcholů – první tři indexy určují trojúhelník a každý další index určí

trojúhelník s předchozími dvěma indexy. Podobných sérií (*strips*) může být libovolně mnoho, ale pro ušetření paměti je samozřejmě vhodné použít jich co nejméně.

Zajímavým prvkem dat je skutečnost, že všechny souřadnice jsou udávány jako jednobajtová nebo dvoubajtová celá čísla. To nejen opět výrazně šetří paměť, ale také to na mobilních zařízeních šetří výpočty (viz cíle M3G specifikace). Ve výsledku je však nutné tato data stejně převést na desetinná čísla. To má na starosti právě *VertexBuffer*, který k souřadnicím vrcholů a textur přidává desetinnou hodnotu *scale* a vektor *bias*. Výsledné souřadnice se pak získají vynásobením celočíselných hodnot hodnotou *scale* a přičtením *bias* k jednotlivým komponentám. Výjimkou jsou normálové souřadnice, které se automaticky přepočítávají lineárně na rozmezí hodnot [-1, 1].

Vzhled objektu *Mesh* je dán instancí třídy *Appearance*. Ta spojuje instance tříd *Material* (barvy *diffuse*, *specular*, *ambient* a *emissive* a hodnota *shininess*), *Texture2D* (obrázek textury, způsob jejího roztahování a afinní transformace), *Fog* (lineární nebo exponenciální mlha) a instance tříd se zobrazovacími vlastnostmi *CompositingMode* a *PolygonMode*.

Vytváření textury za běhu aplikace se provede nejprve převedením nativního obrázku z grafického API mobilního zařízení na instanci třídy *Image2D*. V podstatě lze o převedení ARGB reprezentace na reprezentaci dat po jednotlivých barevných složkách. Textura je pak vytvořena přímo z instance *Image2D*. Pokud jsou data v *Image2D* za běhu změněna, mělo by to být automaticky reflektováno i ve zobrazené textuře. Problém ovšem je, že změna dat v *Image2D* musí být provedena na úrovni oddělených barevných složek, takže buď je nutné mít už předpřipravená data nového obrázku k přímému nakopírování, nebo pouze na datech provést nějaké jednoduché filtrování barevných složek. Pokud chce programátor na textuře aplikovat geometrická primitiva (kreslení čar, obdélníku apod.), je nutné vždy vytvořit novou instanci třídy *Image2D* a vytvořit celou novou texturu. To je však náročné jak na paměť, tak na výpočetní čas a tedy proveditelné jen v několika málo situacích (více viz využití M3G v 2D hrách). Textura musí mít vždy rozměry obdélníku o velikosti hrany rovné mocnině čísla 2. Největší použitelné textury mohou mít rozměry 256x256, ale použití tak velkých textur výrazně snižuje vykreslovací rychlost. A mnoho zařízení se s nimi nedokáže vypořádat ani paměťově, takže se je buď vůbec nepodaří vytvořit, nebo po několika zobrazovacích cyklech dojde k přeplnění paměti a aplikace se ukončí.

Třída *Mesh* má dvě podtřídy, které rozšiřují její chování o specifické vlastnosti. Jednou je *MorphingMesh*, která kromě základního *VertexBuffer* obsahuje ještě několik dalších cílových *VertexBuffer* a dynamickou hodnotu váhy pro každou z nich. Výsledné souřadnice polygonů jsou pak dány součtem souřadnic základního a cílových bufferů po aplikování jednotlivých vah. Lze tak snadno měnit polygony z tvaru do tvaru.

Druhou podtřídu je *SkinnedMesh*, který umožňuje vytváření kostěných modelů. Kromě základních polygonů obsahuje vlastní strom uzlů, který definuje kostru. Každý uzel v kostře je pak možné připojit k sérii vrcholů v kořenových polygonech (kostra je zpravidla strom uzlů *Group* bez grafických dat). Na tyto vrcholy je pak aplikována stejná transformace jako na uzel (kost), ke kterému jsou připojeny. Každé takové připojení kosti k polygonu je vážené a výsledná transformace je dána váženým součtem všech transformací od připojených kostí.

K pochopení M3G už zbývá jen popsat metody animace stromu scény. Animace jsou definované přímo v nadtřídě všech M3G objektů *Object3D*. Ke každému objektu je možné přidat instance třídy *AnimationTrack*. Každá z nich definuje jednu z řady animovatelných

vlastností (např. difuzní barva, hodnoty transformačních matic, úhel světla, úhel výhledu kamery, váhy *MorphMesh* apod.) a sekvence hodnot v klíčových snímcích *KeyframeSequence*. Každá sekvence definuje množinu hodnot, referenční časy pro každou z nich a jeden z několika způsobu interpolace mezi nimi. Po zavolání metody *animate* s aktuálním animačním časem jako parametrem se pak automaticky aplikují všechny animace na danou instanci a všechny uzly v jejím podstromě. Pro programátora je pak použití velmi prosté, protože všechny animace jsou přímo zakomponovány do scény.

Vykreslení scény je už jednoduché pomocí třídy *Graphics3D*. Její instance se naváže k instanci nativní třídy *Graphics* (ať už pro kreslení do obrázku nebo na obrazovku) a buď se vykreslí celá scéna použitím metody pro vykreslení instance *World (retained mode)*, nebo se ručně nastaví aktivní kamera, přidají světla a vykreslí se jakýkoliv uzel přímo (*immediate mode*). Pak jsou ovšem ignorovány jakákoliv další světla ve vykreslovaném uzlu.

V popisu implementace našeho enginu se budeme zabývat výhradně touto knihovnou. Je totiž nejrozšířenější – je implementována na drtivé většině telefonů, které se objevily od začátku roku 2007 – a má také největší možnosti.

I na telefonech s Mascot Capsule v3 se doporučuje spíše používat M3G. Její implementace na těchto telefonech byla dokonce vytvořena taktéž společností HI CORPORATION a je známa i pod kódovým označením Mascot Capsule v4 (i když s předchozí verzí nemá prakticky nic společného).

4.3.4 M3G 2.0

Pomocí Mascot Capsule v3 i M3G 1.1 bylo vytvořeno mnoho 3D her od 3D šachů přes závodní simulátory, RPG, až k jednoduchým *first person shooters*. Žádná z těchto her však nebyla implementována na široké spektrum telefonů – buď kvůli slabé podpoře jednotlivých 3D knihoven, nebo kvůli nízké rychlosti zobrazování na mnoha zařízeních. Kromě toho i v M3G stále chybí prvky, které by grafiku vylepšily natolik, aby přitáhly hráče zvyklé z PC her na mnohem vyšší úroveň detailů a snímkovou rychlost.

Proto bylo rozhodnuto, že specifikace M3G se výrazně rozšíří a do specifikačního procesu se zapojila také společnost AMR, která vyrábí čipy pro mobilní zařízení. Výsledkem snažení specifikační skupiny je M3G verze 2.0 (JSR-297, [20]), jejíž první draft byl navržen na konci července 2007 a která je aktuálně (polovina roku 2010) ve fázi schvalování odbornou veřejností (*public review*).

Nová verze je zpětně kompatibilní s verzí 1.1, jedním z jejích klíčových prvků je však rozdělení celé specifikace na dva stupně. Prvním z nich je tzv. jádro (*M3G 2.0 Core*), které je složeno hlavně z původní funkcionality verze 1.1, a druhou jsou pokročilé funkce (*M3G 2.0 Advanced Block*). Mobilní zařízení, která budou implementovat druhou verzi specifikace, musí bezpodmínečně obsahovat všechny funkce z jádra a funkce z pokročilého bloku buď všechny, nebo žádné. Tím by se mělo předejít další fragmentaci API (kterou však měla vyřešit už první verze). Pokročilé funkce mají společnou zejména skutečnost, že je nelze efektivně implementovat bez hardwarové podpory – tj. bez grafického čipu přímo v mobilním zařízení.

Jednotlivé funkce probereme pouze přehledově. Specifikace nebyla ještě finálně schválena a není k dispozici ani referenční implementace, takže není ještě možné zjistit, jak rychlé a použitelné budou jednotlivé funkce přímo na mobilním zařízení. Máme také určité pochybnosti o tom, zda bude M3G 2.0 vůbec někdy na mobilních zařízeních skutečně využita.

Verze 2.0 zjednodušuje hierarchii tříd. Funkčnost z *TriangleStripArray* je sloučena s abstraktní nadtřídou *IndexBuffer* (už se nepředpokládá jiná definice geometrií), *MorphingMesh* je sloučen s *Mesh*. Díky tomu lze morfovat i *SkinnedMesh*. Do animačních sekvencí jsou kromě klíčových snímků zavedeny i události, které se spouštějí ve chvíli, kdy animace dosáhne určitého času. Takže lze programově spouštět chování synchronizované s animací. Před třídou *Appearance* je vložena nadtřída *AppearanceBase*, která umožňuje definovat jiné pořadí zobrazování než jen podle vzdálenosti objektů od kamery – zavádí seskupování vzhledů do vrstev, které se zobrazují postupně. Nadtřída je definována hlavně kvůli nové podtřídě vzhledu, kterou je *ShaderAppearance*.

Možnost použití programovatelných shaderů je zřejmě nejdůležitějším prvkem, který zavádí nová specifika v *advanced block*. Jde o celou skupinu tříd, které jsou navrženy přímo v kompatibilitě s OpenGL ES.

V základním bloku se výrazně rozšířily možnosti obrázků. Byla přidána nová nadtřída *ImageBase*, která definuje vlastnost automatického vytváření úrovní mipmappingu. V samotné třídě *Image2D* pak byly přidány metody pro nastavení dat buď přímo z pole ARGB barev pixelů nebo z nativního GUI obrázku. Navíc lze těmito metodami nastavovat data pro každou úroveň mipmappingu zvlášť. *Image2D* může být také cílem samotného kreslení části scény (na textuře lze tedy zobrazit část scény). Další novou podtřídou je *DynamicImage2D*, kterému se nastaví zdroj (typicky opět nativní obrázek) a grafická data jsou z něj automaticky obnovovány před každým předkreslením. Instance této podtřídy je možné použít pouze pro vytváření textur (tedy ne třeba na pozadí scény). Dalším typem textury v *advanced block* je *TextureCube*, která ve spojení s obrázkem *ImageCube* slouží pro vytváření *cube maps* (viz [19]).

Největší změny zaznamenala třída *Node*. Nejdříve popíšeme nově přidaný mechanismus *level of detail (LOD)*. Každému uzlu lze nastavit úroveň detailů (jednotkou jsou *features per unit*) a po zavolání metody *selectLOD* na některém uzlu se na všech instancích *Group* v jeho podstromě, které mají aktivovaný LOD mechanismus, hledá uzel zobrazující se s nejvhodnější úrovní detailů. Implementace se snaží najít takový, který by zobrazil optimálně jednu *feature* na každý pixel aktivní kamery. Mechanismus LOD není vůbec propojen se samotným mechanismem zobrazení, pouze uzlům nastavuje příznak, jestli mají být vykreslovacím cyklem zobrazeny nebo ne.

Každý uzel může volitelně určit svoje trojrozměrné hranice pomocí *bounding box* nebo *bounding sphere*. Tyto hranice pak může implementace volitelně použít pro zrychlení zobrazování nebo *picking*.

Po málo použitelné implementaci kolizí pomocí vrhání paprsků ve verzi 1.x specifikace (viz později), zavádí nová verze skutečnou detekci kolizí pomocí *k-discrete orientation polytops*. Je vhodné mít polytopy buď předpřipravené přímo v souboru se scénou, nebo je nechat vygenerovat automaticky. Ruční nastavování lze provést také, ale podle specifikace to bude velice pracné. Pro každý uzel je pak možné vyhledat všechny kolize s jiným uzlem.

Pro bližší detaily odkazujeme na návrh specifikace [20], resp. na specifikaci OpenGL [21], podle které je M3G navrženo.

4.3.5 Java binding for OpenGL ES

Při přehledu 3D enginů na mobilních telefonech je třeba zmínit také JSR-239 [22]. V podstatě ale nejde o samostatné API. Na zařízeních, které obsahují podporu OpenGL ES, lze pomocí tohoto API volat nativní OpenGL funkce. Jde tedy pouze o *binding* nativní knihovny do javového kódu. Ačkoliv podpora tohoto API by se dala čekat nejvíce na nových symbianových telefonech od Nokie s grafickým čipem, které OpenGL podporují pro nativní kód, bohužel je dostupná jen na několika novějších telefonech Sony-Ericsson (14 telefonů v polovině roku 2009, což je tragicky málo).

Je však velmi pravděpodobné, že se dalších zařízení dočkáme velmi brzy. Stojí také za zmínku, že tato knihovna je podporována platformou Android. Podle našeho názoru má toto API na zařízeních s grafickým čipem větší budoucnost než M3G 2.0, jelikož OpenGL má více možností a přitom ho programátoři na rozdíl od M3G dobře znají a nemusí se tedy učit novou technologii.

4.4 Využití 3D API v 2D hrách

4.4.1 Motivace

I když 3D knihovny jsou v principu určeny pro vytváření 3D her, jejich široké možnosti se dají využít i pro vylepšení běžnějších 2D her. Zde budeme mluvit výhradně o M3G, i když Java binding pro OpenGL sem přirozeně patří také, protože velká část jeho funkčnosti se týká pouze 2D grafiky.

4.4.2 Vykreslení popředí 2D her pomocí 3D API

Běžné 2D hry se vyznačují tím, že na pozadí se kreslí obrázky, typicky složené z dlaždic v několika vrstvách, a přes ně se kreslí animované nebo statické sprity znázorňující herní postavy a objekty. Animace spritů se dříve vytvářela jako série po sobě jdoucích snímků, kde každý snímek byl tvořen samostatným obrázkem. Později se přešlo na úspornější způsob. Každý snímek spritu je složen z malých obrázků (fragmentů) a definice snímku určuje, které fragmenty a kde se mají zobrazit. Tím se výrazně zlepšily animační možnosti, ačkoliv nároky na paměť zůstaly pořád poměrně malé.

Pro některé hry to však stále není dostatečné – potřebují používat i animace, které by se v optimálním případě měly generovat přímo za běhu. Referenční implementace M3G vytvořená společností Sun obsahovala dříve demo aplikaci využívající pro animaci spritu na popředí právě knihovnu M3G (součástí starších verzí Wireless Toolkit [13]). Aplikace zobrazuje 2D pozadí jako náhled na terén z leteckého pohledu a přes pozadí je pak kreslen sprite malého letadla ovládaného hráčem. Pohyb letadla není kreslen jako animace snímek po snímku, ale je to 3D model kreslený pomocí M3G *immediate mode*. Díky tomu lze velmi snadno animovat natáčení letadla při pohybu do stran. Šlo pouze o jednoduchou ukázkovou implementaci, takže pohyb letadla nebyl detailně propracován, ale dokázala, jak jsou možnosti této techniky široké. Lze jimi nejen zlepšit vzhled animací, ale také podstatně zmenšit velikost zdrojových dat, protože uložení 3D modelu je několikanásobně úspornější.



Obrázek 10: M3G ve 2D hrách - nalevo náhled verze bez 3D, napravo verze využívající 3D (*Stranded 2: Mysteries of Time*)

Kreslení spritů pomocí 3D modelů lze využít v podstatě ve všech typech klasických dvou-dimenzionálních her. Jde převážně o sprity postav, ale lze je použít i na pohyblivé prvky v pozadí. Transformační matice totiž umožňuje model libovolně natočit, případně zvětšit nebo zmenšit, což sice lze v mobilní Javě udělat i bez použití 3D, ale je to pracné, náročné na paměť (nutné použití pixelového bufferu) a na většině zařízeních bohužel i příliš pomalé. Typickým efektem je vytvoření dojmu pohybu ve třech rozměrech změnou velikosti postavy hráče. U prvků pozadí pak zmenšení evokuje větší vzdálenost a dojem prostoru. Rotaci obrázku lze zase využít například pro animace padajícího kamení.



Obrázek 11: Rotace herních prvků pomocí M3G (*Gish Mobile*)

Společným rysem všech těchto her je nutnost implementovat také verzi bez použití M3G pro zařízení, která buď M3G vůbec nemají, mají ho pomalé, nebo je jeho použití nemožné kvůli chybám v implementaci.

4.4.3 Vykreslení pozadí 2D her pomocí 3D API

Naprostě specifickým použitím M3G v 3D hrách je případ hry *Crazy Cats* (vydána v polovině roku 2009). Zde jsou sprity na popředí vykresleny klasickým způsobem a M3G je využito pro vykreslení pozadí. Opět je použito převážně kvůli efektu rotace obrázku, ale jsou pomocí něj vytvořeny i jednoduché animace celé herní plochy.



Obrázek 12: *Crazy Cats* - vlevo herní prostředí ve 2D, vpravo rotace herní plochy pomocí M3G

Java na mobilní telefonech umožňuje rotace obrázku jen do pravých úhlů, takže M3G je použito prakticky jen pro samotné otočení plochy a animace. Před každým otočením jsou pak objekty na popředí předkresleny do herní plochy, z plochy je vytvořena textura a 3D knihovna je použita pro zobrazení přechodových fází. Poté je opět hra zobrazena běžným způsobem.

Jak již bylo zmíněno výše, M3G ve verzi 1.x neumožňuje změnu textury za běhu a je tedy nutné vždy před každým otočením alokovat paměť pro novou texturu a po otočení opět paměť uvolnit. Tato metoda je velice paměťově náročná a na mnoha telefonech způsobuje vytvoření textury i uvolnění paměti mírné sekání kvůli nutnosti volání garbage collectoru.

Pro některá zařízení proto bylo nutné vytvořit vlastní jednoduchý 3D engine, který úplně nahrazuje funkce M3G. Jde skutečně o primitivní engine – pouze pro vykreslení dvou otexturovaných trojúhelníků bez jakýchkoliv výpočtu viditelnosti. Tento způsob sice vyžaduje ještě o něco více paměti, ale tuto paměť lze znovu používat, takže nevzniká fragmentace. Pokud však zařízení neumožňuje ani použití M3G, ani rychlé vykreslování bufferu pixelů na displej, je nutné otáčení simulovat jednoduššími způsoby.



Obrázek 13: Crazy Cats - jednoduchá rotace herní plochy pro zařízení bez podpory M3G

4.4.4 Jiné kombinace 2D a 3D

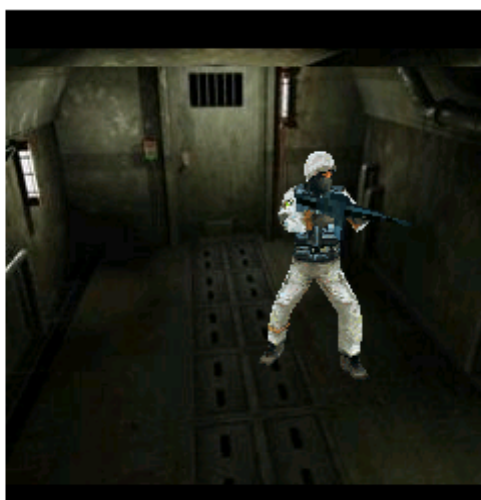
Možnosti kombinace dvourozměrných her a 3D grafiky jsou víceméně neomezené, ale zmíníme se ještě o dvou, které považujeme za zajímavé. Ani jedna z nich zatím nebyla v žádné mobilní hře použita, ačkoliv jsou známé z PC her. Zmiňujeme se o nich proto, že již existují technické prototypy pro mobilní telefony, ze kterých je zřejmé, že lze stejnou funkčnost jako na PC implementovat i v J2ME.



Obrázek 14: 2D plošinová hra s 3D zobrazením (technický prototyp)

První z nich je klasická 2D plošinová hra, avšak se světem vytvořeným kompletně ve 3D a nestandardním pohledem kamery. Fyzikálně je to stále plošinová hra, ale grafické zpracování ji kompletně odlišuje.

Druhým zajímavým případem je hra ve stylu her *Resident Evil* na PC. Svět se skládá ze vzájemně propojených místností. Podle toho, jak se hráč ve světě pohybuje, se pohled přepíná mezi předdefinovanými kamerami. Nejdůležitější však je, že ačkoliv grafika vytváří 3D dojem, tři-dimenzionální jsou jen postavy v popředí. Pozadí je složeno z několika dvojrozměrných obrázků, na kterých bylo grafikem renderováno 3D prostředí.



Obrázek 15: Hra s před-renderovaným pozadím (technický prototyp)

Kapitola 5

Multiplayer

5.1 Spojení dvou mobilních zařízení

Když mluvíme o moderních hrách na mobilních telefonech, musíme se zmínit také o možnosti multiplayeru. Pokud se podíváme na stránky jakéhokoliv distributora mobilních her, je zřejmé, že her podporujících multiplayer mód je jako šafránu. To má několik důvodů, které vycházejí z omezení pro jednotlivé typy spojovacích technologií.

5.1.1 Internet

Realtime multiplayer na mobilních telefonech nelze zatím uskutečnit pomocí internetového spojení. Internet na mobilních telefonech už je dnes sice dostatečně rychlý, ale problematická je jeho doba odezvy. Při testech cyklické doby odezvy (poslání jednoduchých dat na server a jejich vrácení beze změny zpět na mobilní zařízení) lze zjistit, že v průměru se doba odezvy pohybuje kolem 2 sekund. Není ale stabilní a jednotlivé časy se často vyhoupnou až na 8 sekund. Návrh internetového API v mobilní Javě je navíc v principu určen pro vytváření diskrétních HTTP dotazů.

Na novějších telefonech lze sice vytvořit přímo i socket a udržovat stále spojení se serverem, ale když jsme před několika lety toto spojení testovali, ukázalo se, že mobilní síť (mluvíme zejména o dnes nejpoužívanějším GPRS spojení) nedokáže socket udržet otevřený příliš dlouho a spojení se může kdykoliv ukončit. Průměrná doba života socketu se pohybuje kolem deseti minut. Detekce přerušení socketu navíc není spolehlivá a může se projevit s poměrně značným zpožděním. Navázání nového spojení přitom trvá 20-30 vteřin. Některé novější mobilní telefony sice umožňují připojení i přes wifi, kde většina těchto problémů mizí, ale při vytváření mobilní hry nelze nikdy počítat s tím, který typ připojení uživatel použije.

Internetové připojení lze proto použít pouze pro tahové hry, nahrávání dosaženého skóre na webový server nebo přímo o zaznamenání průběhu hry a její nahrání na server, takže hráči mohou záznam stáhnout a soutěžit s virtuálním protivníkem (tento způsob uplatňovala například firma Geewa ve svých závodních hrách). Vždy však chybí realtime interakce s druhým hráčem.

5.1.2 Infraport

Dřívější telefony byly vybavené infraportem, po nástupu Bluetooth se už však od této technologie ustoupilo a už několik let se do nových telefonů infraporty vůbec nemontují. Na telefonech ze začátku tisíciletí, ještě před nástupem Javy, existovaly nativní hry, které umožňovaly IR multiplayer (například *Snake* na Nokiích), ale na novějších zařízeních už nic takového není možné. I když bylo definováno javové API pro komunikaci pomocí IR, v celé historii mobilních telefonů ho implementovaly pouze jeden telefon Nokia a tři smartphony Sony-Ericsson řady Pxx. I kdyby ho však mobilní zařízení podporovaly více, kvůli svým vlastnostem nebyl infraport pro multiplayer hry nikdy příliš vhodný. Nejen že telefony musí být poměrně blízko sebe a spojení se může ztratit při každém prudkém pohybu, ale oba senzory musí být nasměrovány proti sobě. A vzhledem k tomu, že senzor infraportu nebyval

vždy na vrcholu telefonu, nasměrování senzorů proti sobě by fyzicky znemožnilo ovládnutí hry.

5.1.3 Kabelové spojení

Spojení telefonů kabelem by bylo pro multiplayer ideální, ale bohužel podpora ať už hardwarově nebo v API existuje jen na telefonech některých asijských výrobců a podíl takových telefonů na trhu v Evropě a Severní Americe je menší než 1%.

Podpora kabelového spojení mezi dvěma telefony prakticky neexistuje ani pro zařízení s operačním systémem.

5.1.4 Bluetooth

Jediný použitelný způsob je propojení telefonů pomocí technologie Bluetooth, kterou dnes podporuje drtivá většina telefonů. Masivní rozšíření technologie Bluetooth na mobilních zařízeních bylo hlavním důvodem, proč ostatní způsoby přímého spojení vymizely.

5.2 Tvorba Bluetooth her

Ačkoliv je Bluetooth API (JSR-82, [23]) dnes dostupné na každém telefonu, neznamená to, že lze jakékoliv dva telefony propojit. Tato knihovna je totiž jedna z těch, jejichž specifikace byla sepsána natolik volně, že se od sebe jednotlivé implementace poměrně značně liší. Kvůli tomu je obtížné propojit mezi sebou zejména zařízení od různých výrobců. Při testování však není prakticky možné zkoušet, zda spojení mezi libovolnými dvěma telefony probíhá v pořádku, takže hry se běžně distribuují s tím, že některé kombinace mobilních zařízení nebude možné pro multiplayer použít.

U starších her, kdy ještě Bluetooth API nebylo příliš rozšířené a počet cílových zařízení byl tedy malý, bývalo běžné provést úplné testování přes všechny dvojice podporovaných telefonů. Díky tomu bylo dokonce možné rozdělit hru na dva nezávislé díly – zvlášť single player a zvlášť multiplayer – a distribuovat je jednotlivě. Dnes kvůli nárokům na testování multiplayer částí už takový postup není možný. Na první pohled se sice zdá, že rozdělení na dvě části by přineslo dvojnásobné zisky, ale bohužel to tak není. Není to příznivě přijímáno distributory a ve výsledku je to nerentabilní.

Dříve se ještě rozdělení hry dalo zdůvodnit nutností zmenšit celkovou velikost aplikace, ale to už dnes není natolik výrazný problém. Rozdělení aplikace pak sebou nese také požadavky na dvojí testování, dvojí ověřování (často další nezávislé testování) u distributora (případně distributorů) a dvojí marketingovou kampaň.

Nejvíce však nasazení multiplayer Bluetooth her (ať už součástí hlavní hry nebo samostatně) omezuje skutečnost, že nejsou pro hráče dostatečně lákavé. Přidání multiplayer módu do hry prakticky nezmění celkový počet zákazníků. To je způsobeno převážně způsobem nákupu mobilních her – složitostí hry a hodnocením recenzentů se orientují pouze *hard-core* hráči, ale ti stále tvoří minoritní podíl na trhu a naprostá většina uživatelů hru kupuje jen podle několika náhledů ze hry a podle jejího umístění na žebříčku oblíbenosti. Proto si ani nekoupí samostatnou Bluetooth verzi hry, pokud už mají single player verzi. Náhledy ze hry už jim nepřinášejí nic nového. Naproti tomu nelze vytvořit pouze multiplayer hru, bez možnosti single player módu, protože je těžké prodat aplikaci s tím, že si ji musí zároveň koupit i někdo jiný, aby vůbec byla hratelná. Je to opět už zmíněný trend *casual* her.

Bluetooth hry vznikají proto zejména v poloamatérském prostředí, kde se autoři nebojí do implementace multiplayer módu investovat hodně času, protože jim jde více o dobře vytvořenou hru než o finanční rentabilitu (viz např. hra *Gish Mobile*).

5.3 Vlastnosti Bluetooth API

Bluetooth knihovna (JSR-82, s volitelnou podporou OBEX) je velmi jednoduchá a skládá se z pouhých několika tříd. Většina funkcionality je určena pro vyhledání dostupných zařízení v okolí a prozkoumání služeb, které poskytují. Některá zařízení různých výrobců se navzájem vůbec nedokáží vyhledat, s tím pak programátor nic nezmuže.

Nejprve je nutné upřesnit Bluetooth klient-server model, protože má jednu odlišnost, kvůli které i mezi programátory dochází k mnoha nedorozuměním v názvosloví. Bluetooth spojení funguje tak, že jedno zařízení vytvoří službu s předdefinovaným UUID (jednoznačný identifikátor typu služby) a začne tedy fungovat jako Bluetooth server. K této službě se pak může připojit právě jedno mobilní zařízení – Bluetooth klient. Pokud je třeba propojit pouze dvě zařízení, je jedno, které z nich potom převezme funkci aplikačního serveru, který bude řídit vzájemnou komunikaci. Je-li však nutné propojit více zařízení najednou, musí být nejprve spuštěné všechny Bluetooth servery a jeden Bluetooth klient se musí ke všem připojit. Bluetooth klient tak musí plnit funkci aplikačního serveru a přeposílat data mezi jednotlivými aplikačními klienty (Bluetooth servery).

Průběh komunikace pak vypadá tak, že jedno zařízení musí být spuštěno jako Bluetooth server, tj. začít poskytovat službu s určitým UUID. Druhé zařízení pak spustí vyhledávání zařízení a vybírá z nich ty, které poskytují hledanou službu. Z výsledné skupiny pak nechá uživatele vybrat zařízení, ke kterému se chce připojit. Pro uživatele to znamená, že nejprve musí spustit všechny klienty a teprve jako poslední spustit server. Server spustí vyhledávání zařízení ve svém okolí (standardně se nastaví filtrování podle typu zařízení pouze na mobilní telefony) a na každém zařízení poté vyhledá poskytované Bluetooth služby. Pokud nalezne zařízení, která poskytují službu s hledaným identifikátorem, nabídne uživateli jejich seznam, aby vybral, ke kterým se chce připojit.

Podle specifikace se jeden Bluetooth klient může připojit až k sedmi serverům najednou, ale prakticky je toto číslo mnohem nižší – na většině implementací se klient může připojit k pouze jedné službě. Aplikačním serverem u her pro více hráčů se tedy musí stát nejlepší možné dostupné zařízení. I tak je nutné multiplayer módy redukovat tak, že hratelnost je plně zachována i pro dva hráče, a více hráčů může být podporováno pouze jako vylepšení.

API pro vyhledání zařízení je navrženo jako série asynchronních callback funkcí, které se volají při nalezení každého dalšího zařízení a další služby. Hledání je možné kdykoliv přerušit, takže je možné snadno vytvořit grafické menu, ve kterém se postupně objevují nalezená zařízení a uživatel může vyhledávání ukončit hned, jakmile se objeví zařízení, které hledá.

Z vyhledaných informací o zařízení lze poté sestavit *locator*, tedy protokolovou adresu, která se předává obecnému API pro vytváření připojení (*javax.microedition.io.Connector*). Spojení může být klasicky dvou typů – buď streamované spojení protokolem RFCOMM nebo paketové L2CAP. Práce s nimi je obdobná jako pro jakékoliv jiné synchronní a asynchronní spojení. RFCOMM se hodí zejména pro připojení telefonu k externím zařízením, která poskytují trvalý proud dat (např. externí GPS modul), avšak pro použití v mobilních hrách se příliš nehodí,

protože je pomalé, má vysokou latenci a blokovací volání se obtížně vkládají do herních mechanismů. V mobilních hrách se totiž snažíme co nejvíce omezit počet běžících vláken a jejich vzájemné synchronizování. Pro každé synchronní připojení by bylo nutné vytvořit speciální vlákno čekající na data, což by pro aplikační server způsobilo přílišnou zátěž.

Asynchronní L2CAP slouží pro zasílání malých paketů. Jejich velikost je omezena hodnotou MTU (*maximum transfer unit*) a na mobilních telefonech zpravidla nepřesahuje 512 bytů. Systém komunikace je triviální, není však snadné ho implementovat bez chyb. Na některých telefonech například musí být velikost paketu rovna vždy MTU, na jiných je nutné posílat pravidelně neprázdné dummy pakety, aby nedošlo k vyprázdnění bufferu příjemce, protože pak by se spojení přerušilo. Naštěstí tyto chyby jsou více doménou starších telefonů a dnes už se tolik neobjevují, ale přesto je vždy nutné Bluetooth hry pečlivě testovat, zejména na nestandardních kombinacích zařízení a na zařízeních různých výrobců.

Portování Bluetooth hry je časově značně obtížné, zejména u her, které jsou náročné na výpočty, nebo mají náročnou grafickou stránku. Pokud je totiž většina procesorové síly zařízení věnována náročným výpočtům, nativní vlákno zajišťující Bluetooth komunikaci nedostává dost procesorového času pro posílání a příjem paketů a dochází ke značným latencím. Stejně jako u jakékoliv jiné asynchronní komunikace je nutné řešit případy, kdy pakety dorazí k příjemci v odlišném pořadí, než byly odeslány, případně nedojdou vůbec. Oproti PC hrám nelze zajistit zařízení, které by bylo dedikované jako herní server a pouze řídilo komunikaci mezi klienty. To znamená, že implementace serveru musí být co nejjednodušší a všechny výpočty je nutné decentralizovat na jednotlivé klienty – herní server je nutné omezit právě jen na přeposílání dat a hratelnost na něm musí být plně zachována.

Existují však problémy, které nelze navzdory všem snahám uspokojivě vyřešit ani na moderních telefonech. Jedním z nich je například náhle ukončení komunikace s jedním z klientů. Taková chyba může nastat poměrně snadno, například při slabé baterii, příchozím volání, doručení textové zprávy, nebo po tvrdém ukončení jednoho z klientů červeným tlačítkem apod. Reakci herního serveru na takovou chybu je sice možné programově vyřešit (v Javě prostřednictvím mechanismu výjimek), ale samotná implementace Javy tuto situaci často nezvládá a dochází k ukončení nebo zamrznutí herního serveru. I když je jako server použito zařízení, které tuto situaci zvládne, nikdy nelze odpadlého klienta znovu připojit už k probíhající hře. To je způsobeno právě Bluetooth klient-server modelem, protože vyhledání zařízení musí iniciovat právě herní server (Bluetooth klient) a to nelze zajistit, pokud už probíhá hra a tedy komunikace s dalšími klienty.

5.4 Jiná využití Bluetooth komunikace ve hrách

Kromě multiplayeru lze Bluetooth ve hrách využít i jinak. V nedávné době se například objevil speciální joystick určený právě pro mobilní hry (zařízení *Zeemote*, [24]), který se připojuje právě pomocí Bluetooth a umožňuje snadnější ovládání hry, zejména na mobilních zařízeních s nestandardním rozložením klávesnice nebo s dotykovým ovládáním. Implementace ovládání pomocí *Zeemote* má pak vliv na prodejnost her, protože pokud už hráči toto zařízení vlastní, sami se snaží vyhledávat hry, které ho podporují.

Kapitola 6

Implementace engine

6.1 Přístup – požadavky na 3D engine

Při navrhování engine pro FPS 3D hry jsme nejprve narazili na otázku, jak daný problém uchopit. Důležitým rozhodnutím bylo hlavně to, co má daný engine podporovat a co ne. Pokud bychom mluvili o PC hře, bylo by jasné, že engine musí být co nejobecnější a přitom podporovat co nejvíce vlastností. U mobilních her je však nutné najít nutné minimum a ostatní vlastnosti nechat na programátorech dané hry. To je dáno tím, že každé obecné řešení vyžaduje více výpočetní síly a paměti než řešení psané přímo na míru danému použití. Proto je samozřejmé, že mobilní engine nebude obsahovat například žádný skriptovací jazyk, protože interpretace skriptu je náročnější než implementovat reakce na jednotlivé herní události přímo v kódu dané hry, nebo vytvoření jednoduchého skriptu přímo na míru zamýšleným situacím. Mobilní 3D světy budou navíc vždy poměrně malé, takže napsání skriptů na míru nebude nikdy příliš obtížné. Naproti tomu je však nutné engine pro skriptování připravit, tedy zajistit aby programátor mohl jednotlivým objektům přiřazovat identifikátory a ve svém kódu nebo skriptech se na ně odkazovat.

Jednotlivé požadavky na engine a jejich řešení jsme shrnul do následujících podkapitol.

6.2 Vytvoření 3D světa

Každá komplikovanější mobilní hra si žádá vytvoření editoru, pomocí kterého lze nadefinovat herní svět, jeho vlastnosti a jednotlivé levely či mapy. Ve většině případů lze použít stejný editor na více her (například dlaždicový editor pro sestavení 2D pozadí) nebo použít už existující editor pro PC hry a vytvořit převodník do formátu použitelného na mobilním zařízení. Pro 3D hry je vytvoření herního světa jedna z nejsložitějších věcí a odpovídající editor je tedy nepostrádatelný.

6.2.1 Editor

Nejdříve jsme plánovali vytvořit grafický editor, tj. program, se kterým by pracoval přímo designer grafiky a modeloval v něm 3D objekty. To se však ukázalo jako naprosto nevhodný přístup. Vytvořit dobrý grafický editor je velmi obtížný úkol a jde v podstatě o práci, která je zbytečná, protože již existují editory, které podobné věci dělají. Navíc aby bylo možné použít knihovny objektů na internetu, bylo by potřeba napsat i importovací algoritmy pro soubory z nejpoužívanějších grafických editorů (Blender, 3D Studio Max, Maya). Překonat tyto profesionální editory by i tak bylo velmi těžké a uživatelé (designeré grafiky) by s takovým programem odmítali pracovat a preferovali by lepší editor, na který jsou zvyklí. Proto jsme návrh editoru výrazně zredukovali.

Náš editor není určen pro přímé upravování 3D modelů. Pouze umí načíst modely v mobilním formátu M3G (exportery pro nejrozšířenější editory již existují) a pak z nich sestavovat složitější scény. Grafici tak mohou dále používat svůj oblíbený editor a náš program je uživatelsky zaměřen spíše na level design.

Vstupními soubory jsou tedy už vytvořené 3D objekty v M3G souborech, z nichž editor vytváří jednotlivé knihovny objektů. Ke každému objektu je pak možné přidat další dodatečné informace. Jelikož importované objekty mohou pocházet z různých zdrojů, mohou mít například nastavené odlišné měřítko. Pro použití v editoru jim lze proto změnit měřítko na danou jednotkovou velikost, změnit natočení (nutné například pro výstupy grafických editorů, které mají souřadnicové osy nastavené jinak než M3G) a nastavit posunutí, aby byly na správné pozici na jednotkové mřížce.

Objekty jsou v editoru zobrazeny pomocí knihovny Java3D. Využití Java3D se nám zdálo velice výhodné, protože její strom scény je velmi podobný M3G. Převod z M3G objektů do Java3D objektů lze elegantně provést rekurzivně na celém M3G stromě postupně od listů až ke kořeni.

6.2.2 3D svět

U 3D světů je obvyklé, že svět je složen z jednotlivých buněk (*cells*), které se načítají do paměti podle potřeby a při přechodu mezi nimi je nejprve jedna buňka smazána z paměti a poté druhá načtena. U PC her bývá samozřejmě načteno najednou více buněk, aby z nich bylo možné vytvořit rozlehlé světy s plynulým přechodem mezi buňkami. Na mobilních telefonech s malou pamětí by taková implementace byla obtížná, ale pokud by byly jednotlivé buňky dostatečně jednoduché a sdílely mezi sebou objekty, ze kterých jsou vytvořeny, je možné to udělat také. Výstup editoru musí být tedy takový, aby jednotlivé buňky mohly mezi sebou objekty snadno sdílet. Mobilní engine musí být také schopen jednoduše zjistit, které objekty jsou již načtené a nenačítat je znovu. V zájmu jednoduchosti jsme engine a vzorovou hru implementovali tak, že předpokládá vždy pouze jednu načtenou buňku, ale udržuje si informace o načtených objektech a lze jej tedy snadno rozšířit na podporu více načtených buněk najednou.

Každá buňka má danou velikost v jednotkách a umožňuje vložení libovolného množství 3D objektů. Každý objekt je možné libovolně a fině transformovat bez zbytečné duplikace dat vrcholů. Je však zřejmé, že čím více typů objektů bude každá buňka obsahovat, tím náročnější bude načtení a zobrazení buňky na mobilním zařízení. Editor nevytváří na objekty v buňce žádná omezení. Udržení složitosti na dostatečně nízké úrovni je ponecháno na level designerovi, který bude mít přehled o tom, jak jsou jednotlivé objekty složité.

Každému jednotlivému objektu je možné nastavit textový identifikátor, který lze pak v enginu použít pro nalezení daného objektu a práci s ním (např. ve skriptech). K implementaci identifikátorů a několika dalším vlastnostem objektů v buňkách se ještě vrátíme.

Kromě objektů, které jsou přímo součástí jednotlivých buněk, musí 3D svět obsahovat i jiné objekty. Prvním typem jsou globální objekty, které musí být načteny neustále bez ohledu na to, která buňka je aktivní. Mezi takové objekty patří zejména 3D objekty pro zobrazení hráče, nepřátele a jiné opakující se prvky. Druhým typem jsou lokální objekty, které buď nejsou přímo součástí světa, ale musí být načtené ve chvíli, kdy je načtená určitá buňka, aby mohly být do světa dynamicky přidávány. Jako příklad je možné jmenovat třeba několik objektů pozadí světa (*Background* v M3G), které mohou být v buňce přepínány podle herního času a vytvářet iluzi dne a noci.

Pro implementaci podobného typu chování a zjednodušení skriptování musí editor podporovat ještě vytváření značek. Značka je vlastně pojmenovaný vektor v buňce – udává počáteční

a cílovou pozici. Na rozdíl od identifikátoru objektů však typ značky nemusí být v buňce jednoznačný, protože typicky je potřeba mít více značek stejného typu (například pro označení *spawn points* pro NPC, z nichž jeden je náhodně vybírán).

Poslední vlastností každé buňky je pozadí. Jelikož pozadí je jeden z M3G objektů, který se v běžných 3D editorech nastavuje obtížně, editor světa obsahuje vlastní editor pozadí.

Více vlastností 3D svět nepotřebuje.

6.2.3 Exportování 3D světa

Vytvoření a editace 3D světa tvoří jen jednu polovinu editoru. Druhou, neméně důležitou částí, je exportování světa pro mobilní engine.

U každé mobilní hry je příprava dat velmi důležitá, a to ze dvou důvodů. Prvním je skutečnost, že data musí být co nejmenší, aby celková velikost aplikace nepřesáhla maximální limity. Druhým důvodem je omezená výpočetní síla mobilních zařízení, která nutí programátora zmenšit složitost načítání a tím zkrátit neúměrně dlouhé čekací doby. Zdrojová data aplikace musí být tedy uloženy v co nejúspornějším binárním formátu. Vzhledem k tomu, že mobilní telefony neumožňují *random-access* přístup ke zdrojům, jen zřídkakdy umožňují přístup k více zdrojům zároveň a kvůli zlepšení komprese, je vhodné data umístit do co nejmenšího počtu souborů.

Data, která je nutná exportovat, se dají rozdělit do tří skupin. Jsou to jednotlivé M3G soubory, obrázky použité v texturách a v pozadích scén a data jednotlivých buněk. Obrázky mohou být exportované také jako speciální M3G objekty, ale obvykle to není nejlepší řešení, protože v M3G souboru jsou obrázky uloženy velmi primitivním a neúsporným způsobem, zatímco uložení obrázku jako samostatného souboru (odkazovaný z M3G jako *External Link*), umožňuje velikost souboru (PNG formát) optimalizovat na co nejmenší velikost, případně ho i měnit bez zásahu do ostatních dat.

Běžný M3G soubor má v sobě uložená data jednoho nebo více podstromů stromu scény. *Loader* v mobilním API umí každý soubor načíst pouze jako celek a programátorovi vrátí objekty, které tvoří kořeny těchto podstromů. Abychom mohli jednotlivé objekty efektivně sdílet mezi buňkami a vyhnuli se načítání nepotřebných objektů, je při exportování nejdříve nutné objekty rozdělit do skupin podle jejich vzájemné závislosti a podle toho, v kterých buňkách jsou použity.

Globální objekty (viz výše) a všechny objekty, na kterých globální objekty závisí, jsou přidány do první skupiny. Každý další objekt pak na začátku tvoří jednu vlastní skupinu a tyto skupiny jsou slučovány podle vzájemných závislostí. Závislost uvažujeme pouze na úrovni nejjednodušších objektů. Například souřadnice vrcholů, normál a souřadnice textury, které jsou používány nějakým 3D objektem, jsou vždy sloučeny do jedné skupiny, protože by měly být v naprosté většině případů načítány najednou. Dalším krokem je zjišťování, které skupiny jsou využívány kterou buňkou. Skupiny, které jsou používány pouze jedinou buňkou, mohou být sloučeny se skupinou, která je používá.

Problémem při slučování jsou složené objekty, které mají ve svých podstromech objekty z různých skupin a mohly by způsobit jejich nadměrné slučování. Takovými objekty jsou typicky *Mesh* a *Group*. Ty proto nejsou slučovány s ostatními. Nemohou však být uloženy v samostatné skupině, protože se z jedné skupiny nelze odkazovat na objekty v jiných

skupinách. Nejsou proto uloženy jako M3G objekty, ale jejich data jsou uložena přímo – ve formátu obdobném, v jakém by byly uloženy pomocí M3G, avšak se speciálním identifikátorem pro každý M3G objekt, na který se odkazují. Při načítání jsou pak tyto objekty vytvořeny pomocí konstruktorů a propojeny s již načtenými M3G objekty.

Data jednotlivých buněk jsou jednoduchá – každá buňka se skládá pouze z několika vlastností (rozměry, pozadí) a z jednotlivých objektů. Objekt je dán jako odkaz na exportovaný M3G objekt (viz dále) a transformační matice udávající jeho polohu v buňce. Soubor s daty buňky má pak na začátku index, obsahující pro každou buňku seznam skupin, jejichž objekty používá. Při spuštění aplikace je tento index načten a neustále je držen v paměti.

Když je požadováno načtení nějaké buňky, prvním krokem je načtení všech skupin objektů, které buňka používá a ještě nejsou načtené v paměti (tj. používané jinou načtenou buňkou). Data všech ostatních skupin jsou přeskočena. Ve stejném souboru jsou pak data objektů uložených přímo (objekty typu *Mesh* a *Group*, viz výše). Jelikož všechny objekty, na které se mohou odkazovat jsou již načteny, mohou být načteny také.

Dalším krokem je sestavení světa určeného buňkou, tj. nejprve je vytvořena instance *World* a pak jsou ze souboru s daty buňky načteny odkazy na všechny objekty. Ty jsou nalezeny a jsou přidány do světa se svou transformační maticí. Jde tedy o velmi jednoduchý postup.

Identifikátorem, který je používán pro rozpoznání M3G objektů je jejich *userId*. Jde o prostý číselný identifikátor, ale jeho výhodou je to, že M3G API podle něj umožňuje vyhledávat v podstromech scény. Protože však toto vyhledávání zabere nějaký čas, již nalezené objekty jsou tímto identifikátorem kešovány v hašovací tabulce. Identifikátory jsou generovány během exportování pouze pro objekty, na které se odkazuje nějaký jiný objekt.

Všechny textové identifikátory vložené uživatelem (jména buněk, objektů v nich a typy značek) jsou při exportování změněny také na celočíselné konstanty. To je výhodné jak kvůli menší velikosti výsledného souboru, tak i kvůli snadnější práci s nimi. Aby však programátor věděl, jaké konstanty použít pro nalezení daného objektu nebo značky, je také vygenerován soubor *CellConstants.java*, který obsahuje definici javového rozhraní. Toto rozhraní pro každý identifikátor definuje celočíselnou konstantu, jejíž název je jednoznačně vygenerován z typu a jména původního textového identifikátoru a hodnota je rovna exportované celočíselné konstantě pro daný identifikátor. Programátor tak může dále používat identifikátory podle jména. Pokud by hra obsahovala externí skripty používající původní textové identifikátory, při jejich překladu do binárního tvaru lze identifikátory nahradit vygenerovanými indexy. Nepřeložené skripty v enginu neuvažujeme, protože jsou pro mobilní engine nevhodné, nicméně změna exportování a načítání, aby indexy zůstaly textové, je triviální.

Poslední důležitou vlastností exportovací části je zjednodušování objektů. Tím není myšleno zjednodušování grafických dat, protože to nelze nijak rozumně udělat bez supervize člověka. Při používání M3G objektů, které jsou vygenerovány nějakým grafickým editorem, se však snadno může stát, že mnohá data jsou duplicitní. Například objekty pro NPC postavy se od sebe mohou lišit pouze texturou, ale přitom mohou být ve zdrojových souborech duplicitní i jejich geometrická data. Exportované objekty stejného typu jsou proto mezi sebou porovnávány na shodu a pokud je mezi nimi shoda nalezena, všechny odkazy na jeden z nich jsou nahrazeny odkazem na druhý.

Druhou možností zjednodušování je omezení některých vlastností objektů, které ovlivňují rychlost zobrazování. Mezi ně patří například *blending* v instancích *CompositingMode* a *Texture* a vlastnost *shading* v *PolygonMode*. Volitelně je možné pro exportování nastavit, že tyto vlastnosti jsou nastaveny na nejhorší možnou kvalitu. Ve výsledku jsou pak takto zjednodušená data zobrazována rychleji. V závislosti na zařízení se zvětšení rychlosti podstatně liší – čím je zařízení horší, tím je zrychlení výraznější (pokud ovšem je daná vlastnost zařízením vůbec podporována).

6.3 Kolizní systém

Každá FPS 3D hra musí samozřejmě disponovat vlastním kolizním systémem, který ve světě omezuje pohyb hráče nebo případně i dalších objektů. Pro hry s jednoduchým světem (viz historie 3D her) lze snadno vytvořit dobře fungující kolizní systém, který bude zároveň výpočetně nenáročný. U hry, ve které by měl hráč kolidovat s objekty libovolných tvarů a natočení, to však není tak jednoduché. Pomocí kolizního systému musí jít také implementovat fyzikální model s volitelnou rychlostí pohybu hráče, padáním se zrychlováním a překonáváním nerovností povrchu (schody, kopce).

Jelikož musíme kolizní systém udržet co nejjednodušší, začneme nejdříve vylučovat vlastnosti, které jednoduchá FPS hra potřebovat nebude. Jako první vyloučíme kolize s dynamickými objekty. To znamená, že celý svět považujeme za statický a veškerá kolizní data se v průběhu hry nebudou měnit. I když se část světa bude měnit (např. animace nějakého objektu), musí s hráčem kolidovat pořád stejně. Tím se ušetří čas nutný na spočítání nových kolizních informací. Za změnu považujeme změnu kolizních dat pro nějaký objekt, nikoliv úplné vypnutí či zapnutí kolizí (např. otevřené a zavřené dveře, pokud dveře tvoří samostatný objekt).

Speciálním důsledkem tohoto předpokladu je nemožnost řešení kolizí mezi různými (ne)hráčskými postavami, jelikož ty jsou zpravidla pohyblivé. Tato vlastnost je však z hlediska hrátelnosti obvykle nevýznamná – není problémem, pokud hráč bude moci procházet skrz jiné hráčské nebo nehráčské postavy. Pokud by nastala situace, kdy takové případy bude nutné řešit, vždy je možné nad kolizní systém implementovat specializovaný kód přímo určený pro danou funkcionalitu (např. vynucení minimální vzdálenosti mezi postavami).

Druhým hlavním omezením, které koliznímu systému vnutíme, bude omezení funkčnosti pouze na avatary ovládané hráči. Náročnost výpočtů totiž na většině zařízení znemožní volání kolizní procedury více než jednou za jeden herní cyklus (kolizní procedura je navíc rekurzivní, viz dále). Pokud ve světě má být více pohybujících se objektů nebo nehráčských postav s umělou inteligencí, bude snadnější omezit jejich pohyb jednoduššími funkcemi - např. vygenerováním čtvercové mřížky pro pohyb NPC, kde každý roh obsahuje nadmořskou výšku (ypsilonovou souřadnici) a informaci o průchodnosti stěny mřížky. Když budeme mluvit o kolizním systému, budeme tedy mluvit o procedurách omezujících pohyb hráče ve statickém prostředí. Pokud se část světa bude měnit, programátor používající engine musí sám vyřešit případy, kdy po změně není poloha avatara korektní vůči kolizním plochám.

Při implementaci systému jsme vyzkoušeli několik přístupů. Jejich společným znakem je, že všechny hledají stěnu, která je ve 3D světě nejbližší k hráčovi. Po nalezení stěny se všechny systémy chovají obdobně. Pokud je stěna dostatečně daleko, krok pohybu může být proveden bez omezení. V opačném případě je avatar posunut co nejbližší ke stěně a novým vektorem

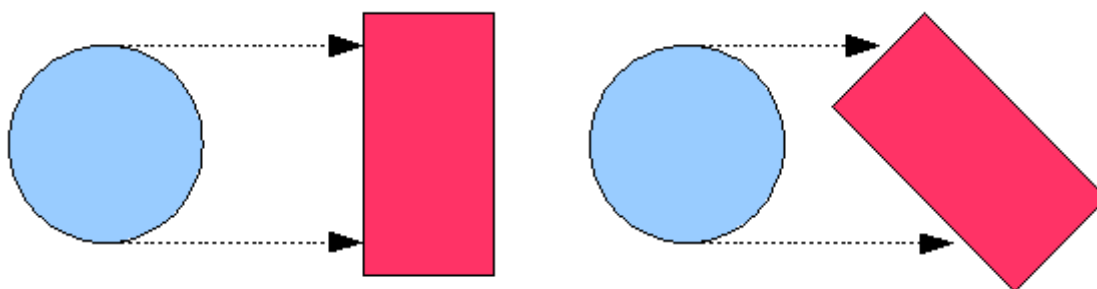
rychlosti je projekce starého vektoru na rovinu rovnoběžnou s danou stěnou. Poté je kolizní procedura rekurzivně volána znovu. Tímto postupem automaticky funguje gravitace a vzniká efekt klouzání (*sliding*) podél stěn, což umožňuje vystoupat na schody, chodit po kopcích apod.

6.3.1 Vysílání paprsků – naivní přístup

V dřívějším popisu knihovny M3G jsme se nezmínili o jedné vlastnosti, která má velký význam při implementaci kolizního systému. Je jí nemožnost z načtených dat efektivně získat informace o jednotlivých polygonech. Ve verzi 1.0 to nejde vůbec, ve verzi 1.1 jen postupným procházením podstromu scény, kdy z každého nalezeného objektu *Mesh* je třeba získat data vrcholů, transformovat je pomocí *scale* a *bias* a transformačními maticemi všech jejich rodičů ve scéně, a následně získat indexy vrcholů jednotlivých trojúhelníků a postupně je procházet.

Protože provádět takové operace přímo v interpretovaném javovém kódu je výpočetně náročné, prvním nápadem je samozřejmě snaha využít některou knihovní funkci, která by přenesla výpočty do nativního prostředí. Jedinou možností, kterou při detailním prozkoumání API najdeme, je vysílání *picking* paprsků. Využití paprsků pro ošetření kolizí je dokonce přímo zmíněno i v několika M3G tutoriálech. Proto jsme se rozhodli vyzkoušet jejich skutečné možnosti.

Nejprve se podíváme na jednoduchý případ, kdy se hráč pohybuje pouze v horizontální rovině (žádný pohyb po ose y). Na první pohled se zdá, že pro zjištění kolize (nejbližší plochy) je potřeba použít pouhé dva paprsky ve směru pohybu, vzdálené od sebe na šířku avatara. Co když je ovšem před avatarem objekt užší než je on sám, např. sloup? Pak by ho paprsky nezachytily a avatar by jím prošel. Je zde sice možné vyslat více horizontálních paprsků a objektům ve světě dát nějakou minimální velikost, aby je paprsky nemohly minout, ale i tak se zdaleka neblížíme řešení, neboť hrany objektů způsobí, že kolizi nezachytíme včas a avatar se dostane dovnitř objektu.



Obrázek 16: Využití paprsků pro detekci kolizí. Vpravo chybový případ, kdy nejbližší bod není zaznamenán.

Principem problému je to, že sice získáme informace o vzdálenosti polygonů v bodech dopadu paprsků, ale nikoliv v bodě, kde je kolize. To by ještě nemuselo znamenat takový problém, protože jsme alespoň zjistili, které polygony jsou k avatarovi nejbližší. Bohužel autoři API neimplementovali možnost získat z paprsku *RayIntersection* data o rozích trojúhelníku, na který paprsek dopadl, ale pouze jeho normálu a souřadnice textury v bodě dopadu. Tím pádem nelze spočítat přesný bod kolize s avatarem.

Nyní zahodíme jednoduchý model horizontálních paprsků a pokusíme se nastíněný problém vyřešit na pohybu v libovolném směru. Uvažujme avatara jako kouli (obvykle by to byl spíše elipsoid). Pak při pohybu libovolným směrem bude jeho průřez ve směru pohybu kruh. Chceme-li zjistit kolizní body, je třeba vyslat paprsky po obvodu kruhového průřezu a několik paprsků uvnitř kruhu (mluvíme o průřezu, ve skutečnosti jsou však vysílány z míst na povrchu koule). Narazíme samozřejmě na stejný problém jako na zjednodušeném modelu. Nicméně zde jsme se ho pokusili vyřešit. Už víme, že dokážeme najít nejbližší trojúhelníky, ale nedokážeme najít nejbližší bod. Zkusili jsme proto implementovat algoritmus, který vzal zjištěné trojúhelníky, našel mezi nimi nejbližší kolizní kandidáty a vyslal další paprsek uprostřed mezi paprsky, které tyto kandidáty našly. Tento postup lze několikrát opakovat a v optimálním případě získat interpolovanou aproximaci kolizního bodu.

Bohužel i toto řešení je v podstatě naivní, a i když v mnoha případech funguje, neřeší samotnou podstatu problému a při troše snahy lze najít pozici, při které avatar projde zdí. Chyba je totiž v tom, že sice známe skupinu nejbližších trojúhelníků, ale nevíme a ani nemůžeme vědět, který z nich obsahuje nejbližší kolizní bod. Část každého trojúhelníku je totiž našim paprskům skryta a nevíme, zda body, které paprsky minuly, jsou blíže než body, které jsme zatím našli jako nejbližší. Vybrání paprsků, mezi kterými se bude interpolovat, je tedy možné pouze na základě heuristiky, resp. hádání bez záruky. A i když nám z předchozího kroku interpolace vyjdou paprsky, z nichž nejbližší kandidáti jsou „hezky“ blízko sebe, nový paprsek nemusí vůbec nic zasáhnout, protože projde mezi trojúhelníky nalezenými předchozím krokem.

Je zřejmé, že paprsky se pro složitější kolize naprosto nehodí. Pro úplnost se ale ještě zmíníme o jejich výpočetní rychlosti, jejíž měření jsme ve skutečnosti provedli ještě před prvními pokusy o implementaci.

Doba nutná pro výpočet paprsku je úměrná tomu, kolik objektů musí API s paprskem porovnat. V nejhorším případě jsou to tedy všechny objekty ve scéně. Před vysláním paprsku je tedy nutné co nejvíce omezit počet objektů, nad kterými se *picking* volá. Vyzkoušeli jsme dvě různá řešení a obě dokázaly vysílání paprsků velmi úspěšně zrychlit, přičemž každé se hodí pro jiné situace.

Prvním řešením je použití oktanového stromu (*octree*, [25]). V naší implementaci každý list stromu obsahoval seznam instancí *Group*, které zasahují do daného místa v prostoru. Když pomineme, že implementace *octree* nejsou v Javě paměťově příliš dobré, je tato struktura výhodná tím, že v ní lze snadno simulovat právě vyslání paprsku a tedy najít přesně ty objekty, přes které by M3G paprsek mohl procházet a *picking* provést pouze na nich.

Druhým řešením je prakticky libovolná jiná struktura, která nalezne objekty ne v dráze paprsku, ale v blízkosti hráče (například *bounding boxes*). Paprsky totiž obvykle stačí vyslat do omezené vzdálenosti a není třeba prohledávat celou dráhu paprsku. Už jednou nalezené objekty lze navíc použít, na rozdíl od *octree* řešení, pro více paprsků. V implementaci také není nutné vyslat paprsek pro každý objekt zvlášť. Stačí mít pro všechny uzly ve scéně vypnutý příznak *picking enabled*, při nalezení vhodných kandidátů jim tento příznak zapnout, vyslat paprsek do celé scény (nebo vhodného společného nadstromu) a poté ho opět vypnout.

Uvažovali jsme i využití vlastnosti *scope*, ale dynamické přepínání *scope* nepřináší žádnou výhodu oproti pouhému přepínání *picking enabled*. Sice tím lze omezit počet objektů, které paprsek najde, ale pro účely kolizí to nemá význam.

První verze enginu jsme testovali na zařízení SonyEricsson W800i, což byl jeden z prvních telefonů s M3G API. S použitím výše zmíněných urychlovacích metod jsme testováním

ověřili, že v jednoduché scéně na něm bylo možné vyslat pouze kolem 30 paprsků za jeden krok herního cyklu, jinak už docházelo k viditelným zpomalením (při nastavení 16 herních cyklů za sekundu). Naproti tomu telefon Nokia 5610 XpressMusic (cca o 2,5 roku novější) už začal vykazovat zpomalení až v hodnotách kolem 500 paprsků za herní cyklus. U novějších zařízeních lze tedy vysílat *picking* paprsky prakticky bez omezení, ale jejich využití není ve sféře detekce kolizí.

6.3.2 Kolize s trojúhelníky

Nefunkčnost kolizního systému založeného na posílání paprsků znamená, že nemůžeme využít stávajících M3G dat a musíme mít navíc předpřipravená speciální data určená pouze pro kolize. Tato data se na nejnižší úrovni musí skládat z informací o jednotlivých kolizních plochách – tj. musíme mít k dispozici data jednotlivých trojúhelníků pro každý M3G objekt. Je to mnoho dat navíc, ale pokud se jednotlivé objekty neskládají z příliš velkého množství trojúhelníků (což by neměly už kvůli složitosti vykreslování), tak se s tím mobilní telefony dokáží paměťově vypořádat. Je nutné si uvědomit, že stejně jako data M3G objektů jsou sdílená, pokud je stejný objekt v jedné buňce použit vícekrát, tak stejně budou sdílena i kolizní data. Budeme preferovat vytvoření těchto dat už během exportování, ale na rychlejších telefonech s M3G API verze 1.1 je možné kolizní data generovat dynamicky, během načítání buňky. Tím můžeme zmenšit celkovou velikost aplikace, ale paradoxně jen na novějších zařízeních, kde už velikost aplikace není natolik limitující. Způsob uložení dat ponecháme stejný, jaký používá M3G, to znamená pole vrcholů a pole *triangle strips*, které je indexují. V průměrných případech je tento formát málo náročný na paměť a umožňuje snadné procházení jednotlivých trojúhelníků.

Na vyšší úrovni je třeba navíc přidat jednodušší kolizní data pro rychlé vyloučení objektů, které jsou od avatara příliš vzdálené. Na to jsme se rozhodli použít *axis aligned bounding boxes (AABB)* – každý objekt v buňce, který se může účastnit kolizí, má v sobě uloženo šest desetinných čísel (dva vrcholy), které udávají pozici nejmenšího kvádra, do kterého se objekt vejde. Kvádr je navíc rovnoběžný se všemi souřadnicovými osami. Využití *AABB* je nenáročné na paměť a velmi rychlé, i kdybychom museli prozkoumat všechny kvádry v každé buňce. Na rozdíl od trojúhelníků nejsou kvádry sdílené mezi stejnými objekty, ale každá instance objektu v buňce má svůj vlastní, aby se nemusely pozice dopočítávat dynamicky podle transformační matice dané instance a výpočet zůstal co nejjednodušší. Tady uplatňujeme předpoklad statického světa.

Existují možnosti jak kvádry udržovat v pomocných stromovitých strukturách a prohledávání tak ještě více zrychlit. Jelikož se očekává, že počet objektů v každé buňce bude poměrně malý, zdálo se nám implementování struktury pro hledání nejbližších kvádrů jako zbytečné komplikování kódu. Nemožnost využívat přímé paměťové ukazatele totiž v Javě způsobuje, že mnoho datových struktur je paměťově několikanásobně náročnějších než například v C++. Posléze jsme se však rozhodli implementovat strukturu, která je v i v Javě nenáročná a přitom poměrně rychlá, a tou je *uniform implicit grid*.

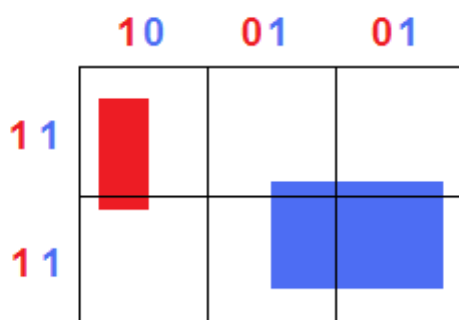
Jde o tří-rozměrnou mřížku složenou z buněk o jednotné velikosti. Princip této struktury vysvětlíme na dvou-rozměrné verzi. Mřížka pro každý řádek a každý sloupec ukládá bitovou masku, která obsahuje jeden bit pro každý objekt ve světě – nastavený na jedničku nebo nulu podle toho, zda se objekt vyskytuje v daném řádku nebo sloupci. Je zřejmé, že se nehodí pro velký počet objektů, protože její paměťová složitost je lineárně závislá na velikosti každé dimenze světa a na počtu objektů ve světě. Když chceme nalézt objekty v určité oblasti mřížky, tak pro každý čtverec, který do oblasti zasahuje, spočítáme bitový součin masky jeho řádku s maskou jeho sloupce a výsledné masky pro všechny čtverce spojíme bitovým

součtem. Bitové pozice nastavené na jedničky ve výsledné masce nám udávají, které objekty zasahují do zkoumané oblasti. Důležité je, že zjištění, které bity jsou nastavené na jedničku, je možné provést s výpočetní složitostí lineárně závislé na počtu jedniček, nikoliv na celkovém počtu bitů.

Optimálních výsledků s *uniform implicit grid* lze dosáhnout, pokud se velikost buňky mřížky blíží velikosti objektů. Mřížka se v naší implementaci generuje během načítání světa a velikost jejích buněk je dána v každém rozměru průměrnou velikostí objektů. Tím zajistíme přiblížení k optimálnímu vyhledávání v mřížce. Abychom se vyhnuli kritickým případům, do implementace jsou zavedeny dva limity, které se kontrolují před generováním mřížky a mohou ji úplně vypnout. Jedním limitem je minimální počet objektů v buňce (malý počet objektů lze rychle prohledat sekvenčně) a druhým limitem je maximální velikost mřížky v bytech (příliš velké nároky na paměť).

Během implementace mřížky jsme museli také vyřešit algoritmus nalezení jediného jedničkového bitu, který je nastaven v celočíselné masce. Lze to ovšem udělat v konstantním čase použitím logaritmu, avšak logaritmické funkce nejsou v J2ME obsaženy. Využili jsme proto konstrukci *switch-case*. Řešení není ideální, protože může docházet k sekvenčnímu hledání správné hodnoty. Předpokládáme ovšem překlad do instrukce bytekódu *lookupswitch*, která sekvenční vyhledávání zpravidla nepoužívá. Lepší řešení se nám nepodařilo nalézt. I přes tento drobný nedostatek totiž mřížka bezesporu zvyšuje rychlost vyhledávání, protože mnoho objektů dokáže z vyhledávání okamžitě vyloučit a věnuje se pouze objektům v prohledávaném prostoru.

Pro detaily implementace doporučujeme [26], pro porovnání rychlosti s jinými strukturami v J2ME [27].



Obrázek 17: Implicit uniform grid ve 2D

Samotný algoritmus detekce kolizí není ničím objevený, neboť je používán většinou dnešních 3D her. Popíšeme zde proto jen jeho základní princip.

Každý krok kolizního algoritmu má za vstup aktuální pozici avatara a cílovou pozici (tj. aktuální pozici navýšenou o zamýšlené posunutí). Avatar je uvažován jako koule o předdefinovaném poloměru – rozšíření modelu na elipsoid je prosté a výpočetně nenáročné. Výstupem je nová pozice.

Nejprve se využije *uniform implicit grid* a *AABB* pro nalezení objektů, které mohou s avatarem kolidovat. Naleznou se tedy všechny objekty, jejichž obklopující kvádry zasahují do kvádrů, který obklopuje avatara v počáteční a v cílové pozici. Získáme tak objekty, s kterými může avatar kolidovat na cestě od počáteční do cílové pozice (*broad phase*).

Poté se procházejí všechny trojúhelníky v každém z nalezených objektů a hledá se trojúhelník, který se dostane do kolize s koulí avatara nejdříve (*narrow phase*). Pro každý trojúhelník je postup následující:

1. Vypočítej *planeDistance* - vzdálenost od pozice avatara k rovině, ve které leží trojúhelník.
- 2a. Pokud je *planeDistance* menší než poloměr, koule protíná rovinu trojúhelníka. Tento případ nazýváme *embedded plane* (vnořená rovina). Nalezneme *planeIntersectionPoint* - nejbližší bod k rovině (k pozici avatara přičteme vektor ve směru opačném k normále roviny a s délkou *planeDistance*).
- 2b. V opačném případě spočítáme nejbližší bod koule k rovině a *planeIntersectionPoint* jako průsečík roviny a přímky zadané nejbližším bodem a vektorem zamýšleného posunutí.
3. Nalezneme *triangleIntersectionPoint*. Pokud *planeIntersectionPoint* leží uvnitř trojúhelníku nemusíme nic počítat, jinak je jím bod na obvodu trojúhelníku nejbližší k *planeIntersectionPoint*. Jde o bod, kde se nejdříve střetne koule se zkoumaným trojúhelníkem.
4. Nalezneme *sphereIntersectionPoint*, tj. bod na kouli, který bude nejdříve kolidovat s trojúhelníkem. Jde o průsečík koule a přímky zadané vektorem zamýšleného posunutí a bodem *triangleIntersectionPoint*. Pokud žádný takový bod neexistuje, pak trojúhelník neleží avatarovi v cestě a končíme. Jinak jsou obvykle nalezeny dva průsečíky, ale není problém rozlišit, který je bližší k rovině.
5. Spočítáme vzdálenost od *sphereIntersectionPoint* k *triangleIntersectionPoint*. Při procházení si ukládáme nejmenší z těchto vzdáleností.

Po průchodu všech trojúhelníků posuneme avatara po přímce posunutí o nejkratší nalezenou vzdálenost. Tím máme zajištěno, že nenastane kolize s žádným trojúhelníkem, protože se k žádnému nedostaneme příliš blízko. Protože ale chceme, aby se avatar posouval podél stěny, vektor posunutí projektujeme na rovinu nejbližšího trojúhelníku a pokud jeho nová délka není příliš malá, voláme kolizní proceduru znovu.

Pro podrobnější vysvětlení algoritmu doporučuji například [28]

Z implementačních detailů bych zmínil pouze dvě věci, které jsou specifické pro mobilní engine. Vzhledem k tomu, že trojúhelníky vytváříme z pole indexů *triangle strips*, musíme nejdříve zjistit, jestli trojúhelník není degenerovaný (s nulovým obsahem). Takový případ nastane, pokud jsou dva vrcholy trojúhelníku shodné, a tedy stačí jen porovnat jejich hodnoty. Druhý aspekt se týká směru normály. V běžných implementacích algoritmu se přeskakují trojúhelníky, ke kterým avatar přichází zezadu (zjistí se podle skalárního produktu vektoru posunutí a normály roviny trojúhelníku), ale indexy vrcholů v *triangle strips* M3G objektů, které jsou generované většinou editory, nebývají uloženy ve správném pořadí, takže zpravidla nelze zjistit, jestli jsou k avatarovi otočené přední nebo zadní stěnou. Při vykreslování je to řešeno jen nastavením *PolygonMode* tak, aby se zobrazovaly obě strany trojúhelníku. Při kolizích pak musíme zjistit, jestli spočtená normála nemíří směrem od avatara, a v případě potřeby jí otočit.

Tento kolizní systém je výpočetně náročnější než kolize pomocí paprsků (které ovšem dobře nefungují), ale při testování jsme ověřili, že mobilní zařízení jsou schopny ho zvládnout. Kritická je hlavně redukce počtu objektů pro kolize v prvním kroku. Pokud jsou objekty v buňce objemově velké, pak se v jednom kroku zpravidla zpracovávají pouze jeden až dva objekty. Je také důležité, aby vektorové výpočty nebyly prováděny pomocnou třídou, ale byly přímo inlinovány v kódu a nedocházelo ke zpomalování voláním metod.

Jednoduchý fyzikální engine se ke kolizím přidá už velmi snadno. Stačí v každém kroku upravit rychlost avatara podle stisknutých kláves a nastaveného zrychlení a přidat gravitační zrychlení. Triviální jsou také drobné úpravy posouvání podél stěny – nechceme-li například, aby avatar sjížděl dolů na mírných svazích, stačí zjistit úhel roviny, podél které k posouvání dochází, a v případě potřeby ho zakázat.

6.4 Viditelné objekty, míření, používání objektů

Klíčovou vlastností pro hry je zjišťování, který objekt v buňce se nalézá přímo před hráčem, respektive v jeho zaměřovači. Pokud to víme, můžeme implementovat interakci s daným objektem – například otevírání dveří, přechody mezi buňkami, animace, nebo míření na jiné hráče.

Tady se nám velice hodí *picking* paprsky. Stačí jednou za cyklus vyslat paprsek z daného bodu na obrazovce (střed nebo pozice zaměřovače) a získáme nejbližší objekt před hráčem. V engineu jsme *pickable* objekty omezili pouze na ty, kterým byl v editoru přidělen identifikátor. Tedy takové, které pak programátor dokáže rozpoznat a namapovat na ně nějaké chování.

Jelikož paprsek umožňuje získat z nejbližšího objektu i souřadnice textury v bodě dopadu paprsku, přidali jsme do editoru také možnost, že pro každý objekt je možné nastavit rozmezí texturových souřadnic, pro které je míření na objekt aktivní. Typickým použitím je například objekt reprezentující budovu, na jehož textuře jsou nakresleny dveře. Pokud jako aktivní část textury nastavíme souřadnice dveří, bude programátor moci rozeznat, kdy avatar může do dveří vstoupit.

6.5 Animace světa

Posledním požadavkem na vlastnosti engineu je možnost začlenit do 3D světa animace a pohyblivé prvky. Nejde o vlastnost podstatnou pro hratelnost, ale z grafického hlediska dokáže každý pohyblivý prvek výrazně zvednout vizuální úroveň scény, takže je dobré mít je k dispozici.

M3G objekty mají možnosti animace přímo zaneseny v sobě. Samozřejmě si programátor vždy může implementovat vlastní animace, kdy přímo mění vlastnosti (transformace, barvy, viditelnost, průhlednost apod.) objektů, ale je výhodnější, když animace vytvoří přímo grafik při vytváření objektů. Programátor pak jen musí definovat situace, při kterých se animace spustí nebo vypne. Některé mohou být spuštěny neustále, některé mohou reagovat na přítomnost hráče (např. houpající se světlo, pokud kolem projde avatar) nebo na herní akci (aktivace tlačítka).

Díky nativní podpoře animací v M3G znamená podpora animací v engineu jen pravidelné volání animační metody s aktuálním časem. Programátor může podle identifikátorů hledat objekty a přepínat jejich animace, případně nastavit skripty, které to udělají automaticky podle definovaných značek v buňce. To však mluvíme pouze o statických objektech v buňce. Animaci objektů, které jsou do buňky přidány dynamicky (např. NPC), musí programátor implementovat sám, ale je to pořád stejně snadné.

Speciálním případem animovaných objektů jsou grafické efekty (výbuchy, silová pole, výstřely laserové pušky apod.). I s těmi se však pracuje stejně. Např. výbuch lze reprezentovat

jako žlutočervený poloprůhledný objekt, který se rychle zvětšuje. Celá animace může být definovaná pomocí M3G a programátor musí před zobrazením efektu jen umístit objekt do světa, spustit animaci a po jejím skončení objekt odstranit.

6.6 Multiplayer

Ačkoliv multiplayer mód není obvyklou součástí mobilních her a není tedy důvod, proč by ho engine měl obsahovat, rozhodli jsme se ho přidat. Na rozdíl od ostatních částí je totiž kód multiplayeru relativně nezávislý na zbytku enginu a v případě, kdy není potřeba, ho lze velmi snadno kompletně odstranit. Pro tyto případy jsme definovali přepínač preprocessoru.

Centrální zde není komunikace mezi jednotlivými telefony, protože to je velmi krátký a jednoduchý kód. Podstatná je zejména implementace vyhledání spojení - spuštění Bluetooth služby na jednom nebo více zařízeních a jejich nalezení Bluetooth klientem. Tato část je netriviální a vyžaduje již hlubší pochopení fungování Bluetooth API. Vyhledání zařízení jsme implementovali obecněji než je zvykem – kód podporuje více připojených aplikačních klientů k serveru, ale lze ho použít i jen na propojení pouhých dvou zařízení, což je v dnešních hrách nejobvyklejší.

Nicméně jsme neměli možnost komunikaci testovat na širokém spektru telefonů, takže je velká pravděpodobnost, že při portování bude nutné udělat drobné úpravy kvůli chybám v API. Přesto jsme se snažili kód napsat co nejrobustněji, se znalostí toho, co obvykle na mobilních zařízeních způsobuje problémy.

Telefony si mezi sebou vyměňují krátké zprávy o velikosti v řádu jednotek bajtů. Po zahájení komunikace všechna zařízení přejdou do fáze čekání. Nejprve server pošle všem klientům informace o nastavení hry, ti potom načtou potřebná data a zašlou serveru notifikaci, že jsou připraveni ke hře. Poté server pošle hráčům počáteční pozice a hra začíná. Během herního cyklu klientská zařízení vypočítají novou pozici a zašlou ji serveru. Server na konci cyklu k pozicím přidá svou pozici a všechny známe pozice zašle klientům. Základní komunikace je tedy velmi jednoduchá. Důležité je, že server ani klienti se nesmí v žádném momentu blokovat čekáním na příchozí data, jinak velmi klesá snímková rychlost. Každé čekání na data musí být proto prováděno v samostatném vlákne. Zpoždění dat se pohybuje kolem 1-3 herních cyklů. Nová nastavení grafu scény podle došlých dat se kvůli synchronizačním problémům nikdy neprovádí ihned po příchodu nových dat, ale vždy až na začátku následujícího cyklu.

6.7 GUI

Poslední částí enginu je infrastruktura grafického uživatelského rozhraní. Na prvním místě je třeba říct, že tato část není nepostradatelná a ani se nepředpokládá, že bude využívána v každé hře založené na tomto enginu. Programátoři už totiž budou mít pravděpodobně vytvořený vlastní základ hry, navíc propojený s buildovacími skripty.

Grafické uživatelské rozhraní enginu má však za úkol demonstrovat, jakým způsobem by měly být ostatní části zapojeny do kódu hry. Je ale psáno dostatečně robustně, aby ho v případě potřeby bylo možné použít v profesionální hře. Skládá se z pěti tříd. Hlavní třída (*Main*) je potomkem třídy *MIDlet* a stará se pouze o spuštění aplikace a poté předává řízení třídě *Controller*.

Třída *Controller* se stará o řízení běhu aplikace a vykreslování. Je potomkem třídy *Canvas* (třída v MIDP API pro přímé kreslení na displej zařízení), zpracovává uživatelské akce (stisky kláves, případně pohyb stylusu) a spouští nové vlákno, které se stará o cyklické překreslování a update hry. *Controller* funguje stavově – má několik různých chování na základě stavu aplikace. V běžných programech by se jednotlivé stavy řešily spíše dědičností, ale na mobilních zařízeních počet tříd a složitost zjednodušíme na minimum a proto raději každou hlavní metodu podle stavu větvíme. Jednotlivých stavů může být poměrně značný počet – v profesionálních hrách až kolem patnácti – ale jen několik je nepostradatelných. Je to *načítání*, kdy jsou načítány základní obrázky pro menu, fonty, lokalizované texty, zvuky apod. Během načítání je uživatelský vstup ignorován. Dalšími stavy jsou *menu*, které slouží jako základní GUI rozcestník, *hra* a *pozastaveno*. Poslední stav je vyvoláván ve chvíli, kdy je aplikace poslána na pozadí, např. při přijaté textové zprávě, informaci o nízkém stavu baterie apod. Jelikož různá zařízení řeší aplikaci na pozadí různě (někdy je úplně pozastavena, někdy na pozadí pokračuje v běhu), je obvykle používán přístup, že stav je detekován a aplikace se přepne do speciálního stavu, kdy jsou všechny zvuky vypnuté, hra neběží a tento stav trvá, než uživatel sám stiskne speciální tlačítko pro opětovnou aktivaci hry. Je to nutné, protože nelze spolehlivě detekovat chvíle, kdy se má hra opět automaticky aktivovat (např. konec telefonního hovoru).

Oproti PC hrám se zmíníme o jedné odlišnosti v řízení hlavního vlákna. PC hry se snaží udržovat stabilní počet cyklů podle zadané hodnoty. Pokud je FPS příliš vysoké, zvýší se čekací doby mezi jednotlivými cykly. Pokud je naopak příliš nízké, typicky se to řeší pouhým updatem hry a přeskočením vykreslení [29]. Předpokládá se totiž, že vykreslení trvá podstatně déle než update. Podobná technika na mobilních zařízeních nefunguje ze dvou důvodů. Prvním je skutečnost, že vykreslení může být podstatně rychlejší než výpočetní část hry (kolize) a pouhé vypuštění vykreslovací části nepomůže. Druhým je problém, že maximální FPS je velmi nízké (u náročné 3D hry kolem 10 FPS na rychlých telefonech) a po vynechání vykreslování v některých krocích cyklu by trhání překročilo únosnou míru a hra by se stala nehratelnou. Na mobilních telefonech proto každý krok cyklu obsahuje jak update, tak vykreslování. Jediný způsob, jak ovlivnit FPS, je tedy délka čekací doby mezi cykly. V aplikaci se nastaví hodnota FPS, které chceme v optimálním případě dosáhnout a z ní je vypočítán čas, jak dlouho by měl trvat jeden průchod cyklu. Pokud je výpočet rychlejší, čekací doba je zvýšena stejně jako v PC aplikacích. Pokud je však pomalejší, neexistuje způsob, jak část výpočtů vynechat, aby došlo ke zrychlení. Čekací dobu lze snížit maximálně na nulu. V PC aplikacích by nulová čekací doba způsobovala problémy, ale telefony si zpravidla samy zajišťují, aby systémová vlákna nebyla od procesoru odříznuta. I tak je však lepší nechat aplikační vlákno chvíli čekat, protože tak VM donutíme, aby systémové vlákno spouštěl vždy na stejném místě cyklu. Kdyby ho spouštěl v průběhu kroku cyklu, mohlo by to způsobit rozdíl v čase mezi jednotlivými překresleními a tedy k neplynulým animacím. Zároveň tak dáme VM možnost zpracovat Bluetooth komunikaci.

Pokud nastavíme FPS na hodnotu 18 a výpočty se nebudou stíhat, reálné FPS bude nižší, např. 12. Počet updatů hry je však vždy roven počtu vykreslování a tedy rychlost hry je řízená rychlostí překreslování. V 2D hrách se tak nestává, že u animací jsou některé snímky při pomalém vykreslování vynechány, animace prostě běží pomaleji. Stejně je to i počítáním herního času. Aby na pomalejších telefonech nebyl problém s nastavenými časovými limity v levelech, musí být počítání času zpomaleno stejným způsobem (délka sekundy prodloužena). To se provede jednoduše – herní čas je zvýšen o konstantu danou optimální hodnotou FPS při každém updatu. Pokud herní cyklus trvá déle, i přičítání času se zpomalí.

U her, které jsou závislé na reálném čase – např. v multiplayer hrách, kde musí být časování synchronizované mezi jednotlivými propojenými zařízeními, takový postup použít nelze. Počítání času je pak udáno buď serverem (obvykle nejrychlejší zařízení) nebo je vyvozeno z reálného času telefonu (*System.currentTimeMillis*). Nevýhodou jsou pak neplynulé (trhané) animace, ale tomu se už nelze vyhnout.

Další dvě třídy tvoří dvojice *Form* a *Item*. Jsou to vzájemně propojené třídy sloužící pro vytváření uživatelských formulářů. Většinou jde o sloupcová menu, ale podporují také posuvný text (využitelný pro položky *help* nebo *about*) nebo primitivní přepínače. Jde o velmi jednoduché třídy, ale dají se využít v širokém množství případů, včetně profesionálních her. Snadno lze také změnit jejich grafický vzhled (pozadí, barvy, fonty).

Poslední třídou je *Game*, která se stará o update a vykreslování samotné hry ve chvíli, kdy je stav instance *Controller* nastaven na *hra*. Tento stav do samostatné třídy oddělujeme kvůli tomu, že kód pro herní stav je podstatně větší než kód pro ostatní stavy, a samostatná třída nám pomůže udržet přehlednost. Ovšem je zde i další důvod. Při startu hry se typicky načítá poměrně značné množství dat, které se ukládají do atributů instance *Game* a které chceme po přerušení nebo skončení hry opět uvolnit. Zrušení celé reference na instanci *Game* s následným voláním garbage collectoru nám umožní uvolnění paměti provést snadno a elegantně.

Z hlediska 3D jsme do enginu zavedli jedno specifikum, které se bude pravděpodobně hodit v kterékoliv hře. Během implementace jsme totiž zjistili, že pomocí M3G nelze jednoduše zařídit, aby některé objekty byly vždy vykresleny v popředí. To je potřeba například pro zobrazení těla avatara. Tělo musí být zobrazeno před kamerou, aby bylo vidět, což ovšem znamená, že když je avatar postaven blízko zdí, ty mohou být fyzicky blíže ke kameře než jeho tělo, a tedy dochází k optickému procházení těla avatara skrze zeď.

V podstatě existuje jen jeden jediný způsob jak danou situaci vyřešit. Tělo avatara (a všechny objekty, které by měly být vykresleny v popředí), musí být vykresleny až po zbytku světa. To lze zařídit jednoduše – svět je zobrazen v *retained mode* a popřední objekty (sloučené do jedné instance *Group*), jsou následně vykresleny v *immediate mode*. To však jako řešení nestačí. V M3G totiž nelze vymazat *depth buffer* bez současného smazání vykreslené scény. Po vykreslení objektů do popředí se proto stále nezobrazí pixely schované za blízkou zdí. Nakonec jsme naštěstí objevili metodu, která vyřeší i tento problém. Před zobrazením objektů lze nastavit, do jakého intervalu hodnot (*window coordinates*) v *depth buffer* se budou všechny vzdálenosti objektů mapovat. Pokud tak zobrazovanému světu nastavíme interval souřadnic například $0,1$ až $1,0$ a objektům v popředí $0,0$ až $0,1$, získáme požadované chování.

Menu i hra se umí automaticky přizpůsobit změně rozlišení displeje (přepínání mezi *portrait* a *landscape* módem), ačkoliv to u dnešních her ještě nebývá úplně zvykem. Není ovšem řešeno ovládání s převráceným displejem, protože to už je závislé na dané hře implementující náš engine.

6.8 Základ jednoduché hry

Pro lepší pochopení herního enginu a editoru jsme nad enginem implementovali jednoduchý základ hry. Programátorovi je tak názorně ukázáno, jak by měl svou hru na enginu postavit, a případně ji použít jako vzor, pokud si nebude ani po přečtení komentářů a dokumentace v některých využitích enginu vědět rady.

Kapitola 7

Závěr

Vývojem pro J2ME se zabývám už osm let - prošel jsem fází free her, podílel se na vytvoření několika profesionálních mobilních her a větších aplikací portovaných na široké spektrum telefonů a nakonec zakotvil ve společnosti zaměřené na vývoj pro mobilní zařízení. Implementoval jsem emulátor J2ME pro webové stránky a zajímám se i o jiné mobilní platformy (Android, Bada). Zním tedy mobilní API, specifika mobilních aplikací i procesy vývoje poměrně do hloubky.

V tomto textu jsem se pokusil shrnout některé svoje zkušenosti, které jsem získal dlouhodobou prací s platformou J2ME a doufám, že jsem alespoň trochu rozptýlil často slychané námitky, že programování pro mobilní telefony je snadné.

V jednotlivých kapitolách jsem postupně splnil všechny cíle práce, které jsem si předsevzal v úvodu, s výjimkou posledního. Historické hry byly více než detailně rozebrány ve třetí kapitole. Problematika přenositelnosti byla vysvětlena převážně v druhé kapitole, ale analyzovali jsme ji v podstatě v průběhu celého textu. Mobile 3D API a stejně i ostatní 3D knihovny jsme rozebrali v kapitole čtvrté a na multiplayer hry jsme se zaměřili v páté kapitole. Pouze implementace FPS hry zůstala splněna jen z poloviny.

Vyvinout kvalitní mobilní hru totiž není v silách jednoho člověka. Bylo by nutné navrhnout game design, zaplatit vytvoření velkého množství 3D grafiky, sestavit herní světy a průběžně testovat na několika desítkách různých telefonů. Očekávaný čas vývoje pouze z programátorského byl odhadován na deset měsíců. Z těchto důvodů jsem poslední cíl nakonec výrazně zredukoval pouze na minimum potřebné pro tento text – implementaci 3D enginu a multiplayer komunikace.

Místo toho jsem však mnoho nových věcí přidal, protože text by byl bez nich neúplný – např. buildovací proces mobilní aplikace, popis Mascot Capsule 3D, možnosti využití 3D knihoven v 2D hrách a základní informace o chování hráčů při nákupu mobilních her.

V praktické části jsem se nakonec více než mobilnímu enginu věnoval práci na editoru, který v původních cílech taktéž nefiguroval. To však není ničím překvapivé. Čas na vývoj pomocných programů je dnes často stejný, nebo i větší než vývoj samotné hry a leckdy jsou na něj vyhrazeni samostatní programátoři. Je to důsledek skutečnosti, že co největší část výpočtů a optimalizací se snažíme přenést mimo mobilní zařízení.

Přímo v enginu nejvíce času zabralo ladění kolizí, Bluetooth komunikace a optimalizace na rychlost. Použití grafického API bylo víceméně přímočaré.

Domnívám se, že engine se může stát základem kvalitních her, ale i tak to bude znamenat hodně práce. Problematické bude zejména portování 3D her na slabší zařízení. I ukázková hra je pomalá na rychlých zařízeních – to je však zřejmě způsobeno příliš vysokou složitostí objektů ve scéně a příliš velkými texturami. Cestou ke zrychlení je tedy další zjednodušování objektů.

Mnoho rozšíření enginu se přímo nabízí. Velmi rád bych například přidal možnost exportování, načítání a zobrazování knihovnou Mascot Capsule v3, čímž by se zvýšila rychlost vykreslování na starších zařízeních Sony-Ericsson, ale nejsem si sám jistý, jestli poměr zvýšení rychlosti bude stát za čas a problémy s tím spojené.

Zřejmě zajímavější by byla implementace knihovny M3G nad OpenGL ES, protože tím by se otevřela cesta, aby se mobilní 3D hry pro J2ME snáze portovaly i pro jiné platformy, zejména na slibně se rozvíjející platformu Android. Když ale připočteme i nejavové platformy, OpenGL ES je používán na platformách Symbian, iPhone i Bada.

Trend vývoje celkově směřuje k portování na platformy založené na různých programovacích jazycích, tedy k vytváření automatických převodů zdrojových kódů tak, aby bylo možné kód psát jen pro jednu jedinou platformu. Cílem je, aby počet úprav po automatickém převodu byl minimální. J2ME obsahuje ze všech platforem minimální funkcionalitu, takže je žhavým kandidátem právě na počáteční platformu. Automatický převod volání knihovny M3G do OpenGL je tedy logickým krokem.

Doufám, že tento text bude pro programátory, chystající se vyvíjet 3D hry pro mobilní zařízení, přínosem.

Zdroje

[1] Connected Limited Device Configuration (CLDC), JSR-139
<http://jcp.org/aboutJava/communityprocess/final/jsr139/>

[2] Mobile Information Device Profile (MIDP), JSR-118
<http://jcp.org/aboutJava/communityprocess/final/jsr118/>

[3] Nokia UI API
http://wiki.forum.nokia.com/index.php/Nokia_UI_API

[4] Mobile Media API (MM API), JSR-135
<http://jcp.org/aboutJava/communityprocess/final/jsr135/>

[5] Antenna
<http://antenna.sourceforge.net/>

[6] Apache Ant
<http://ant.apache.org/>

[7] Proguard obfuscator
<http://proguard.sourceforge.net>

[8] Preverifikace
http://en.wikibooks.org/wiki/J2ME_Programming/MIDlet_Preverify#Preverification_Steps

[9] J2ME Polish
<http://www.j2mepolish.org>

[10] Fishlabs
<http://www.fishlabs.net/>

[11] PNG specifikace
<http://www.w3.org/TR/PNG/>

[12] Nástroj pngout
<http://en.wikipedia.org/wiki/PNGOUT>

[13] Sun Java Wireless Toolkit (WTK)
<http://java.sun.com/products/sjwtoolkit/>

[14] Optimalizace javového kódu
<http://developer.android.com/guide/practices/design/performance.html>
http://www.javacommerce.com/displaypage.jsp?name=java_performance.sql&id=18264
<http://ssuet.edu.pk/taimoor/books/1-57521-197-1/ch30.htm>

[15] J2ME Benchmarks
<http://www.jbenchmark.com>
<http://www.futuremark.com/products/spmark/spmarkjavajsr184>

- [16] Mascot Capsule V3
<http://www.mascotcapsule.com/en/products/mcv3.php>
- [17] Abstracting M3G and Mascot Capsule v3
http://developer.sonyericsson.com/site/global/newsandevents/latestnews/newsapr07/p_abstracting_3dprogramming_tutorial.jsp
- [18] Mobile 3D Graphics API (M3G), JSR-184
<http://jcp.org/aboutJava/communityprocess/final/jsr184/>
- [19] OpenGL Cube Maps Tutorial
http://developer.nvidia.com/object/cube_map_ogl_tutorial.html
- [20] Mobile 3D Graphics API 2.0 (M3G), JSR-297
<http://jcp.org/aboutJava/communityprocess/final/jsr297/>
- [21] OpenGL ES
<http://www.khronos.org/opengles/>
- [22] Java Binding for OpenGL ES API, JSR-239
<http://jcp.org/aboutJava/communityprocess/final/jsr239/>
- [23] Java APIs for Bluetooth, JSR-82
<http://jcp.org/aboutJava/communityprocess/final/jsr82/>
- [24] Zeemote
<http://www.zeemote.com/>
- [25] Octree
<http://en.wikipedia.org/wiki/Octree>
- [26] Uniform implicit grids
Christer Ericson: *Real-Time Collision Detection*, Morgan Kaufmann, 2005, str. 291-293
- [27] Ondřej Mocný: Porovnání rychlosti struktur používaných pro detekci kolizí v J2ME
<http://physics.hardwire.cz/mirror/CollTest.zip>
- [28] Detekce kolizí koule/elipsoidu
<http://www.gamedev.net/reference/articles/article1026.asp>
<http://www.peroxide.dk/papers/collision/collision.pdf>
- [29] Animační cyklus
Andrew Davison, *Killer Programming in Java*, O'Reilly Media, 2005, str. 13-37