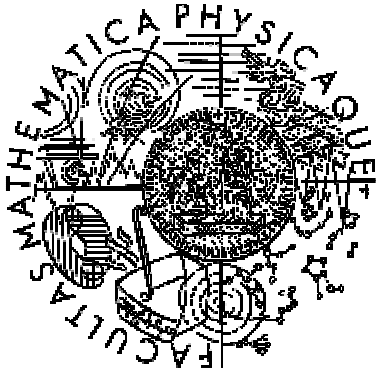
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Jakub Míšek

Paintball 3D pro platformu Symbian

Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. Filip Zavoral, Ph.D.

Studijní program: informatika, programování

2007

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

V Praze dne: Jakub Míšek

1. Obsah

Název práce: Paintball 3D pro platformu Symbian

Autor: Jakub Míšek

Katedra: Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. Filip Zavoral, Ph.D.

e-mail vedoucího: Filip.Zavoral@mff.cuni.cz

Abstrakt: V bakalářské práci se zabývám konstrukcí počítačové hry vyžadující prostorové vykreslování v reálném čase, s předpokládaným nasazením na mobilních telefonech s operačním systémem Symbian. Jsou zde popisovány techniky softwarového vykreslování prostorové scény, vektorové animace a správy a reprezentace příslušných dat. Popsána je i realizace jednoduché umělé inteligence pro protihráče nebo možnost hraní ve více hráčích na více zařízeních. Postupy popsané v práci jsou implementované v přiložené aplikaci.

Klíčová slova: grafika, 3D, hra, mobil, symbian

Title: Paintball 3D for Symbian platform

Author: Jakub Míšek

Department: Department of Software Engineering

Supervisor: RNDr. Filip Zavoral, Ph.D.

Supervisor's e-mail: Filip.Zavoral@mff.cuni.cz

Abstract: In the present work I'm describing construction of the computer game which is featured by using real time rendering. The work is targeted for implementation on cell phones with Symbian® OS. The work describes techniques of software rendering of three-dimensional scene, vector animation and managing and representation of used data. Realization of simple artificial intelligence or multiplayer support is described too. All these features are implemented in attached application.

Keywords: graphics, software rendering, game, mobile, Symbian

2. Úvod

Počty mobilních telefonů a chytrých zařízení mezi lidmi se dlouhou dobu nezadržitelně zvětšují a s tím i odvětví zabývající se mobilní zábavou. Většina uživatelů chce používat své zařízení, které je vždy po ruce, i jako hračku. Přestože obětují každé nové aplikaci průměrně 28 minut, než ji zcela zavrhnou, každá si nakonec najde své místo.

Tato práce popisuje, co se skrývá za vytvořením jedné takové hry. Konkrétně jde o 3D simulaci známé *out-door*[1] akce *Paintball*. Uživatel se stane jedním z hráčů, kde podle pravidel a možností hry bude interaktivně ovlivňovat prostředí skrze postavu. Vše uvidí v reálném čase v realistickém prostorovém zobrazení z pohledu hráče, kterého ovládá.

Práce je rozdělena na části, které je možno řešit odděleně. Samozřejmostí je snaha vypracovat je obecně, což ušetří práci v budoucnu na jiných programech, zpřehlední návrh a zjednoduší realizaci. Základem je tzv. *framework* (4.1), který obstarává funkce závislé na cílovém systému vyšším vrstvám. Zajišťuje práci se souborovým systémem, spuštění aplikace, zpřístupňuje nízkoúrovňový grafický výstup, obstarává přeposílání systémových událostí vyšším vrstvám, stará se i o ukončení programu. Pouhým upravením *frameworku* je tak i celý program upraven pro zcela jiné zařízení. Výše je postaven *Herní engine* (4.2), ten již bývá sestaven velmi obecně a je takovým nástrojem pro samotnou tvorbu hry. Definuje matematické struktury, datové struktury, řesí prostorové vykreslování, animace, správu paměti, načítání a zpracovávání dat a další často používané funkce. *Engine* se v některých místech s *frameworkem* prolíná, jelikož některé jeho části jsou závislé na zařízení, zvláště v oblasti grafiky, kvůli zvýšení výkonu. Poslední část, samotná hra (11), je nad těmito dvěma vrstvami, jde o logické poskládání prostředků nabízených *enginem*, doplněných o další kód, které spolu s vstupními daty tvoří tížený výsledek. Umělá inteligence, správa herních pravidel, pohyb objektů a ovládání je asi to nejzajímavější, co herní část definuje. Ostatní je již zajištěno enginem.

[1] Z anglického *outdoor*, čili *venkovní,* jsou označovány hry odehrávající se na otevřeném prostranství.

3. Analýza

Her takovéhoto typu je na scéně mobilních zařízení málo a stávají se středem velikého zájmu uživatelů. A to z jistého důvodu, programátorská podpora pro tvorbu 3D realistických zobrazovačů do roviny (*rendering*) se vyskytuje jen pro některá zařízení. Proto jakékoliv jednoduché operace spojené s prostorovou grafikou, které může na osobním počítači řešit kterýkoliv začátečník, musí na mobilním telefonu být vypracovány zcela od základů, se značnou znalostí teorie. To se týká samotných principů 3D grafiky, metod řešení viditelnosti a pokusů o realistické zobrazení scény na displeji mobilního telefonu. Tyto hry dále obnášejí jednoduchou implementaci umělé inteligence pro spoluhráče, definici herní scény jako rozsáhlé struktury zapouzdřující a obstarávající vše od řešení detekce kolize a fyzikální zákony, po samotnou správu a přístup ke všem objektům ve scéně. Pak je zde potřeba pracovat s prostorovými modely včetně jejich plynulé animace. Pro toto vše se musejí nějakým způsobem definovat a vytvořit vstupní data, tedy nakreslit textury, nahrát zvuky, vytvořit prostorové modely, navrhnout scénu. K těmto účelům si většinou společnosti vytvoří vlastní software přesně ušitý pro své potřeby. I v tomto případě byla již vytvořena aplikace pracující s prostorovými modely a animacemi a aplikace pro sestavení herní scény.

Nejprve, po samotném nápadu, je třeba sepsat pravidla, ustanovit cíle a zamyslet se nad předpokládanými problémy. Počáteční idea o vysoké úrovni dokonalosti hry musí být dále upřesněna. Tedy rozhodnout, co bude obsahovat herní scéna, jaké budou možnosti interaktivity se scénou, jak se bude hrát, jak vytvořit data a rozvrhnout grafickou podobu. Zde je vhodné zvolit co nejmenší počet základních stavebních kamenů, bude tak jednodušší a rychlejší samotná tvorba dat a zpřehlední se návrh programu. Proto bude herní scéna složena sice jen z mála základních objektů, ale díky širokým možnostem jejich parametrizace, je možno sestavit scénu ve výsledku velice rozmanitou. Například obyčejná podlaha je dána svým povrchem, pozicí a výškou. Lze tak z ní vytvořit cestičku před domem, schod u dveří, kaluž na zahradě, dřevěnou překážku, patník u silnice, papírovou krabici nebo mravenčí cestičku s pochodujícími mravenečky. Uživatel často ani nepozná, že daný předmět má něco společného s podlahou, ale z hlediska návrhu to podlaha je. Následuje ovládání, kvůli možnostem pohybu a řízení aplikace. Ovládání je zvláště u mobilních zařízení dosti omezeno, proto se musí šetřit různými funkcemi,

upřednostňovat předpokládané potřeby uživatele a brát v úvahu zamýšlená podporovaná zařízení.

Problémem je samozřejmě vykreslovat v reálném čase 3D grafiku v rozumné kvalitě. Nebo plynule pohybovat objekty a animovat je. Jsou to důležité oblasti programu na rozmyšlení, které musí být velmi rychlé. Jak vůbec definovat scénu, pravidla hry, tvar prostorových objektů a animace? Jak uzpůsobit ovládání, aby bylo co nejpřívětivější? Dále jsou ve scéně spoluhráči řízeni umělou inteligencí, jistě nelze implementovat složité počítačové vidění a reakce řešit učícími se neuronovými sítěmi. Je tedy třeba optimalizovat a v mnohých oblastech se uchylovat ke kompromisům. Dále musí být šetřeno pamětí, jelikož cílová zařízení disponují maximálně jen pár volnými megabajty.

Výsledek má být rychlý a zdánlivě realistický. Uživatelé také vyžadují množství grafických efektů a také *„něco nového“*, bez tohoto, přívětivého ovládání a hratelnosti nemívá produkt nárok na oblíbenost. Drobné chyby (hlavně během vykreslování) lze tolerovat a počítá se s nimi, na druhou stranu aplikace musí očekávat téměř jakékoliv chování zákeřných uživatelů a nesmí se chovat nepředvídatelně, ani nesmí poškozovat ani příliš omezovat systém, na kterém běží (musí stále zůstat dostatek volných systémových prostředků a musí se počítat s vnějšími vlivy, například řešit možnost příjmu příchozího volání nebo příjmu *sms* během hraní).

4. Herní engine

Hned na počátku je vhodné práci rozdělit na samotnou hru a veškeré složité operace, jejichž řešení bude hra vyžadovat. Engine je právě ta vrstva, poskytující tato řešení. Přesněji herní engine poskytuje řešení problémů, týkajících se programů z oblasti počítačových her. Měl by být navrhnut tak, aby se nad ním, maximálně jen s malými úpravami, dala postavit i jiná hra, s podobnými požadavky čí s podobnou specifikací.

Zde se engine ještě dále dělí na framework a vyšší funkce. Je to z důvodu odlišení části komunikující se systémem a části zajišťující potřeby hry. Díky tomuto separování funkcí je pak snazší řešit tzv. *portaci*, čili upravení programu, aby byl spustitelný na dalších systémech se zajištěním stejné funkcionality.

Tento klasický model si lze představit jako tři vrstvy ležící na sobě, kde každá parazituje pouze na té pod sebou. Vespod nad operačním systémem leží framework, na něm engine a na vrchu samotná hra.

4.1 Framework

Důležité je předem určit, jaké funkce bude poskytovat framework, určit tedy jeho rozhraní. Implementace pak závisí na dostupných API funkcích konkrétního operačního systému [5][7], na kterém má aplikace běžet.

Často si hra vystačí s inicializací po spuštění, prací se soubory, zajištěním herní smyčky (neustále se opakující volání jedné metody, tzv. *heartbeat* nebo *volání snímku, 11.4*), upozorněním na přepnutí na jinou aplikaci a zpět (aby mohl program reagovat pozastavením hry), uživatelský vstup, primitivní grafický a zvukový výstup, zprostředkování komunikace po síti a funkce na ukončení celého programu.

To, jak jsou vnitřně tyto potřeby zajištěny, se může lišit podle cílového systému. Přepsáním frameworku, aby zároveň stále poskytoval stejné rozhraní, lze výsledný program přenést na jiné zařízení, aniž by již bylo nutností zasahovat do vyšších vrstev. Právě to je jeho účel.

4.2 Engine

Zde až jsou definovány jednotné datové typy, rozhraní, struktury a metody zajišťující předpokládané potřeby hry, z větší části rozebírané ve zbytku textu.

Nejobsáhlejší poskytovanou částí je schopnost vykreslování 3D grafiky (8.3) a definice objektů pro práci s grafikou (8, 9), dále pak je obsažena správa paměti, načítání multimediálních dat, jednotné uživatelské prostředí (11.3), základní správa herní scény (7), matematické struktury řešící detekce kolize (10) a operace s vektory a maticemi (8.3.1).

5. Ukládání dat

Data uložená spolu s aplikací říkají jak se chovat, co zobrazovat a s čím pracovat. Jsou rozdělena do jednotlivých souborů určitých typů a každý takovýto soubor nese jisté informace. Jde o různé konfigurační soubory, obrazové soubory, zvukové soubory, 3D modely objektů a mapy herních scén.

5.1 Rastrové obrazy a zvuky

V případě bitmapových obrázků a jednoduchých zvuků je optimální využít již existujících formátů, například JPEG (*Joint Pictures Expert Group*), BMP (*Bitmap*) a WAV (*Waveform Audio*). Odpadá pak potřeba konverze do nějakého vlastního formátu a o načítání se skrze framework postará systém sám. Navíc tyto formáty plně postačují potřebám her.

Některé hry si definují vlastní formát pro zvuky a obrázky, aby tak mohli lépe data spravovat. Někdy je do z důvodů šetření místa, kdy jsou menší soubory umístěny do jednoho velkého. (Soubory menší než velikost sektoru na disku zabírají celý sektor, umístěním tisíců malých souborů do jednoho velkého se dá velmi ušetřit.) Jindy je potřeba k datům doplnit další informace (popis materiálu, fyzické rozměry, …) a je vhodné je umístit do společného souboru, popřípadě ještě s použitím jednoduché rychlé komprese (např. LZW).

5.2 Konfigurační soubory

Různé parametry programu se ukládají do konfiguračních souborů. Ty nesou pouze hodnoty předem daných atributů a ze strany programu nedochází k jejich modifikaci. Pro uložení je nejvhodnější obyčejný textový soubor. Jednoduše se čte, je rozšiřitelný a uživatel ho může modifikovat.

5.3 Modely a scény

Otevřených vektorových formátů je velké množství, většinou je možné jich využívat bez větších problémů. Přeci jen ale bývají pro prostorové modely a definice herní scény navrhovány vlastní formáty. Výhodou jsou data přesně ušitá potřebám aplikace, nezabírající žádné zbytečné místo navíc, nevýhodou pak nutnost sestrojit externí aplikaci pro tvorbu těchto dat (14.1, 14.2.4, 14.2.5).

5.4 Datové soubory

Setkáme se s dvěma kategoriemi souborů a podle toho k nim je přistupováno.

Jednak máme soubory nesoucí statické informace, nebudou modifikovány a obsahují grafická data, definice scén, zvuky, jazykové překlady apod. V každé instanci programu a na každém zařízení budou reprezentovat stejné informace.

Další jsou menší soubory, do kterých si aplikace ukládá stav a nastavení, aby uživatel nepřišel o nic, co si v programu navolí či vytvoří ani po opětovném spuštění. Tyto soubory mohou mít konstantní velikost, pouze se mění obsah. Práce s nimi je často realizována pouhým uložením/načtením 1:1 předem dané binární struktury.

5.5 Sekvenční ukládání po blocích

Zvláště pro modely a scény je vhodné umožnit budoucí rozšiřitelnost. Bylo by i pěkné ustanovit nějaké jednotné rozhraní pro práci s vlastním formátem těchto binárních souborů. Pokud chceme načíst nějaký objekt, bude uložen v samostatném souboru, jehož název známe.

Data v těchto souborech jsou rozdělena na fragmenty (tzv. *chunks*). Fragmenty jsou uloženy sekvenčně po sobě a jsou určeny hlavičkou. Ta obecně definuje *typ* pomocí 32bitového identifikátoru a *velikost fragmentu* v bajtech. Aplikace může skákat po jednotlivých hlavičkách a postupně podle typu a obsahu fragmentu sestavit daný objekt.

Fragmenty umožňují seskupovat data jednoho objektu, každý fragment může mít jinou velikost, aplikace může číst soubor po částech, není nutná velká vyrovnávací paměť a je zaručena rozšiřitelnost pouhým přidáním svého nového typu fragmentu.

Jednotlivé typy a formáty fragmentů jsou rozebírány v samostatných kapitolách dále.

6. Návrh scény

Prvotním problémem je rozhodnout, co bude hra nabízet, jak bude vypadat, co bude účelem. A to vše v závislosti na předpokládaném zařízení, na kterém má být spouštěna. Právě to ovlivní výsledný návrh nejvíce, neboť zde se setkáme s netypickými počítači, jež disponují pomalým procesorem, malou operační pamětí, ale zároveň barevným displejem s větším rozlišením. Dále tvarem, klávesnicí ani výdrží baterie to nejsou zařízení designována pro hry. Proto nelze zkopírovat návrh od úspěšných již ověřených produktů z herních konzolí nebo stolních počítačů (pouze těch opravdu starých).

Díky malému počtu kláves dojde ke snížení pohyblivosti hráče, málo paměti a nižší výkon zabraňuje přílišné složitosti. I tak lze dosáhnout příjemného kompromisu.

Ze zadání vezměme prvky, které musí být obsaženy. Paintball je venkovní hra o 4 a více hráčích rozdělených v týmech, rychlou chůzí se pohybují prostorem (*scénou*), který je osazen různými překážkami. Za pomoci jedné zbraně střílející barevné kuličky se hráči snaží bolestivě označit protihráče, čímž ho vyřadí ze hry. Další varianty hry přidávají každému týmu jedno své stanoviště s vlajkou, přičemž cílem je donést vlajku nepřítele na své stanoviště, aniž bychom byli vyřazeni. Tím tedy dostaneme scénu, tvořenou postavami se zbraněmi rozdělených do týmů, vlajkami, nějakou zemí (podlahou), po které se dá utíkat, možná nějaké stropy, obloha, překážkami (zdi, stěny) a spoustou barevných cákanců od rozprsknutých kuliček. Budeme vyžadovat umělou inteligenci jako spoluhráče pro uživatele, ale také možnost hrát jednu hru ve více živých lidech na více zařízeních za pomoci síťové komunikace mezi zařízeními. Umělá inteligence (*AI*) také vyžaduje jistou přípravu, jisté informace. Není možné nechat ji, aby se ve scéně orientovala stejným způsobem, jako by to dělal člověk, tedy zrakem. Necháme ji pohybovat se po předem daných cestách, nejspíše reprezentovaných neorientovaným grafem.

Spočítáním si nároků na plně obecnou scénu tvořenou těmito prvky zjistíme, že není možné v reálném čase vykreslovat scénu z pohledu hráče, ani ji spravovat a aplikovat obecně fyzikální zákony. Můžeme buď hledat jiné možnosti a trochu upravit zadání nebo scénu postupně omezovat.

Jednou z alternativních možností, používanou dříve, je zvolit pevnou kameru (*objekt, ze kterého je scéna sledována, 8.3.3*), všechny nepohyblivé prvky si předkreslit a dále

v reálném čase pouze přikreslovat pohybující se objekty. Takto by bylo dosaženo rychlosti a plynulosti zobrazení, ovšem samotná hra se velmi znepřehlední. Scéna musí být totiž rozsáhlá a zároveň hráč nesmí vidět své protivníky. Záměrem také bylo umístit kameru přímo na místo pohledu hráče.

Obvykle se tedy volí druhá cesta, zjednodušení a upřesnění vlastností některých prvků scény. Tím, že předem o některých objektech budeme znát jisté informace, velmi zrychlíme a zjednodušíme práci s nimi. S ohledem na rozměrově malé výstupní zařízení a omezené ovládání, můžeme udělat jisté ústupky, které budou velmi nápomocné, nebudou scénu příliš znatelně ochuzovat a naopak zpřehlední orientaci uživatele v ní. Pozorovací směr kamery bude vodorovný a nebude se otáčet kolem své osy. Překážky neboli stěny budeme předpokládat vždy svislé, sahající až k zemi. Rozdělíme je na stěny plné výšky, kde nikdy nebude vidět scéna za nimi, a stěny částečné, kterých bude jen málo. Stěny plné výšky nelze přelézt a zcela zakrývají scénu za sebou, což umožní další optimalizace, zároveň přidává omezení na výšku celé hrací plochy. Podlaha bude pouze vodorovná, tvořena trojúhelníky, ve výšce maximálně poloviny plné stěny. Tím je zaručeno, že hráč se nedostane do takové výšky, aby mu bylo umožněno za plné stěny vidět. Trojúhelníky umožní vytvářet všemožné tvary podlahy a nebude problém počítat kolize s nimi. Co se týče nějakého grafického znázornění pozůstatků po střelách na stěnách a podlaze, které bývají vytvářeny za běhu hry, je třeba omezit jejich maximální počet. Nikdy nesmí být uživateli dovoleno vytvářet libovolný počet čehokoliv, neboť pokud bude moci, udělá to. Při vzniku jednoho nového fleku od střely je tedy vhodné nechat zaniknout jeden starý, pokud je jejich předem daný maximální počet překročen. Všechny tyto prvky, ať jsou to různé stěny nebo fleky, budou popsány polohou a tvarem. Tvar není obecně definovaný, je specifický pro různé typy objektů, nabízí tak různé možnosti parametrizace. Ta umožňuje formovat jeden typ objektu do mnoha dalších podob, je totiž záměrem vyvarovat se stereotypu a okoukanosti, se zachováním jednoduchosti návrhu. Nakonec si ještě necháme prostor pro přidání dalších efektů, které ale nebudou mít za následek znatelnou ztrátu výkonu.

7. Reprezentace scény

Všechny objekty, se kterými se během hry má pracovat, musí být k dispozici a musí s nimi být možno provádět požadované operace rychle a jednoduše. Operacemi jsou hlavně vytvoření, pohyb, vykreslení, odstranění a informování o stavu. O hromadné řešení těchto úloh se stará *herní scéna*.

Tento objekt zaobaluje a spravuje fyzickou reprezentaci, živé objekty, ale i další informace, jako textura oblohy (*textura 8.2.4*) nebo předdefinované cesty (). Je postaráno o načítání, uložení v paměti, vykreslování, pohyb a vytváření a zánik všech objektů ve scéně. Jde tedy o takový kontejner dalších objektů, herní scéna řeší přístup k nim a jednotné volání jejich metod.

Další důležitou součástí je optimalizace přístupu v případě přístupu k objektům a optimalizace řešení viditelnosti během vykreslování. To si vyžaduje odlišení statických dat a dynamických objektů. S každou touto skupinou je jednodušší a výhodnější zacházet jinak.

7.1 Statická data

Ve scéně je velké množství nepohyblivých objektů, mezi které patří hlavně stěny a podlaha. Ty budou tvořit větší část herní scény a budeme vyžadovat jejich rychlé nalezení podle polohy. Pro zrychlení tohoto vyhledání je můžeme jednou na začátku zatřídit do složitější vyhledávací struktury a následně již jen provádět dotazy, kterým může být rychle vyhověno. Třídí se obvykle podle polohy a dotazy se provádějí na skupiny objektů podle dané lokality.

Dalším objektem, který není během hry modifikován, je *cesta*. Na tomto objektu jsou prováděny jiné operace, hlavně nalezení cesty z jednoho místa do druhého. Hledání uzlu na cestě se provádí jen velmi zřídka a nehledá se podle přesně dané lokality, spíše se hledá nejbližší uzel k danému bodu v prostoru. Roztřídit tak tento objekt do stejné vyhledávací struktury jako ostatní objekty není vhodné. Objekt *cesta* řeší optimalizace vyhledávání sám jiným způsobem.

- **Optimalizace přístupu**

K uložení statických objektů je vhodné použít binární vyhledávací strom. Technika se v grafice označuje jako *BSP-tree* (*Binary Space Partitioning tree*). Jednotlivé uzly jsou určeny lokalitou, do které zasahují, reprezentovanou kvádrem (*obálkou*) v prostoru, dceřiné uzly pak dělí kvádr svého rodiče na polovinu v místě nejdelší hrany. Kořen stromu zasahuje přes celou scénu, jejíž rozměry jsou předem známy. Rekurze dělení probíhá, dokud není dosažena daná minimální velikost kvádru. Listy stromu pak obsahují seznam objektů, které svým středem do obálky daného listu zapadají.

Musí se také počítat s tím, že objekty zatříděné do stromu mají nezanedbatelný rozměr. Proto po sestavení této struktury musí být rozšířeny hraniční oblasti listů a následně rekurzivně jejich rodičů tak, aby žádné vnitřní objekty nepřesahovaly hranice uzlů, které leží na cestě od kořene k listu s daným objektem. To proto, že zatřídění se provádí podle souřadnic středu objektu. Kdybychom brali v úvahu během zatřiďování celou oblast objektu, ve výsledku by jedna jeho instance zapadla do více listů, což je nežádoucí.

Nyní lze jednoduše vyhledat celé skupiny objektů v dané oblasti s logaritmickou složitostí, průchodem stromu od kořene k listům skrze uzly pronikající do žádané oblasti.

Statickými objekty zatříděnými do tohoto binárního stromu ve scéně jsou konkrétně objekty podlahy, stropu, zdí a cákanců od střel.

- **Podlaha, strop**

Podlaha a strop jsou jedny ze stavebních kamenů scény. Musí mít definovanou polohu a vzhled. Podle polohy jsou roztříděny do vyhledávací struktury. Operace na nich prováděné budou vykreslení a v případě podlahy ještě hledání průsečíku s přímkou. Strop je pro hráče nedosažitelný a pro střely také, počítání kolize s ním by tedy bylo zbytečné.

Tvar definován vodorovným trojúhelníkem v prostoru a vzhled určen texturou pokrývající objekt umožňují vytvářet dostatečné množství variací.

Důležitým předpokladem je právě fakt, že tyto objekty jsou ve scéně umístěné vodorovně. To umožní značné optimalizace v oblasti vykreslování (*8.3*, 8.3.5), hledání

kolize a dodržování fyzikálních zákonů. Také tvorba scény se tím zjednoduší. Toto upřesnění umístění má však za následek jisté ochuzení hráče o některé možnosti.

Podlaha je důležitým prvkem, který definuje výšku země. Na hráče ve scéně má působit gravitace, a proto je třeba mít definovanou výšku země v bodě (*10*).

Strop má pouze dekorační úlohu, pokud ale počítáme s implementací vykreslování podlahy, nebude problém vykreslovat i strop.

- **Stěny**

Pokud máme prostor omezený na výšku a pohybujeme se tak pouze v jedné vrstvě, nabízí se možnost velmi snadné implementace optimalizace ořezáváním scény. Zavedeme objekt stěny, který je svislý a sahá vždy od podlahy až do dané maximální výšky a neprotíná se s jinými objekty. Dále má danou orientaci a v úvahu se bere pouze stěna, jež je z pohledu kamery orientována zleva doprava. Vše směrem od kamery za viditelnou stěnou nemusí být vůbec vykresleno, tyto části scény jsou tedy stěnou oříznuty.

Tato technika je podobná metodě *portálů* [[1].6.3], kde je svět rozdělen na uzavřené místnosti s obdélníkovými okny. Pokud je vidět některé z oken, vykresluje se i vnitřek portálu v dané oblasti viditelné skrze okno. Metoda portálů ušetří 70-80% zpracovaných polygonů, metodou ořezávání stěnami je dosaženo podobných výsledků, za naší podmínky omezeného prostoru. Spolu s dalšími technikami ořezávání (8.3.6) tak bývá dosaženo požadované rychlosti zpracování a vykreslení scény.

Se stěnami bude dále třeba hledat kolize (10.1), aby se zabránilo hráčům procházet jimi a aby bylo možno určovat pozice, kde se mají vytvořit fleky od střel (10).

Pokud budeme umět vykreslit a hledat kolize se stěnami, můžeme scénu obohatit o další podobné objekty odvozené od základní stěny. Každý s odlišnými vlastnostmi. Pro potřeby této hry budeme mít tedy:

- Původní plné stěny zajišťující optimalizace vykreslování ořezáváním.
- Částečné zdi, které nebudou sahat do maximální výšky a nebude tak možno jimi ořezávat. Bude třeba je pouze vykreslovat a hledat kolize se střelami.
- Keře, odvozené od částečných zdí, navíc s děravými texturami (8.2.4.3). Musí být tedy vykresleny nakonec, nebude s nimi počítána žádná kolize, pouze umělá inteligence by skrze ně neměla vidět. Hráč se za nimi tedy bude moci skrýt.

- Vyvýšené zdi, nad plnými stěnami, sloužící pouze jako dekorace. Tyto stěny mají za úkol obohatit prostor nad výškově omezenou scénou a je třeba je pouze vykreslovat. Není třeba hledat kolize s nimi a není možno je ořezávat plnými stěnami, neboť je plné stěny nemohou zcela zakrýt.

Cesty

Ve virtuálním prostoru musí být aplikace schopna najít cestu z jednoho místa do jiného a to hlavně pro potřeby umělé inteligence(11.2.1). Tento klasický problém může být řešen několika způsoby, v závislosti na daném prostoru. Cestu bychom hledali jinak na šachovnicové hrací ploše se čtvercovými překážkami a jinak při hledání cesty pro nezanedbatelně velký objekt v obecném 3D prostoru s obecnými 3D překážkami. V prvním případě lze velmi rychle najít nejkratší cestu bez jakékoliv přípravy předem. V druhém je buď potřeba si připravit trajektorie, po kterých se můžeme bezpečně pohybovat (*cesty*) nebo použít složitější algoritmy, zahrnující detekce kolize s nemalou časovou složitostí.

V našem případě jde o zjednodušený 3D prostor s předem zadanými trajektoriemi, hledání cesty bude rychlé a cesty realistické (neboť je „nakliká“ sám návrhář scény). Na cestu můžeme pohlížet jako na neorientovaný graf, jednotlivé vrcholy leží v 3D prostoru a víme, že po hranách mezi uzly lze bezpečně projít, aniž by došlo ke srážce s nějakým statickým objektem (stěnou).

Na tomto grafu není požadováno hledat posloupnost vrcholů na cestě mezi dvěma vrcholy, ale užitečnější je nalezení sousedního vrcholu ke zdrojovému, abychom takto postupně došli k cílovému. Nechceme tedy celou cestu najednou, ale postupně. Je to operace velmi často využívaná, hodilo by se tedy si cesty předem nalézt a uložit do paměti. Chceme nalézt nejkratší cesty ze všech zdrojů do všech vrcholů, váhy hran se rovnají jejich délkám v prostoru a jsou tedy kladné a nenulové. Graf tvoří jednu komponentu. S malou úpravou je tak vhodné využít *Dijkstrův* algoritmus. Pro každý vrchol coby zdroj vygenerujeme *dijkstrovým* algoritmem seznam předchůdců na cestě do ostatních. Z každého vrcholu se tedy zpětně dovedeme dostat do zdroje. Celý výsledek pro všechny zdroje uložíme do matice n×n, kde n je počet vrcholů, řádky jsou jednotlivé seznamy, a číslo řádku udává číslo zdrojového vrcholu. Tato matice nám pak přesně řeší daný problém, neboť pokud chceme číslo vrcholu *V*, který následuje na nejkratší cestě po

zdrojovém vrcholu *A* do cílového vrcholu *B*, stačí sáhnout do řádku *B* sloupce *A*, kde leží předchůdce vrcholu *A* na cestě z B do A – čili námi hledaný vrchol *V*.

Dalším problémem je napojení se na cestu, pokud na ní právě nestojíme. Jsme-li totiž mimo, musíme nejprve dojít na některý z vrcholů, abychom bezpečně mohli použít cesty ve scéně. Toto se nestává příliš často, pokud už umělá inteligence stojí na grafu cesty, nemá možnost jak ji opustit. Problém se týká spíše živého hráče. V pravidlech hry totiž stojí, že po zásahu musí být hráč vrácen na svou startovní pozici, ten v tu chvíli ale může stát téměř kdekoliv. Proto je třeba ho nejprve dostat na nejbližší vrchol grafu a pak teprve použít cesty stejným způsobem jako to dělá umělá inteligence k návratu na startovní pozici.

Nalézt nejbližší, z nějaké výchozí pozice, viditelný vrchol grafu cesty opět není tak jednoduché, neboť na dráze k němu mohou stát překážky (*stěny*). Stačí ale setřídit vrcholy od nejbližšího vzestupně. Pak jen vybírat vrcholy od nejbližšího a postupně testovat kolize úsečky mezi výchozí pozicí a pozicí testovaného vrcholu s okolní scénou, za použití rutin *detekce kolize se scénou* (10.1), čímž nalezneme nejbližší vrchol v přímé viditelnosti. Jelikož ze setříděného seznamu vybíráme postupně vrcholy od nejbližšího, k uspořádání vrcholů se hodí použití haldy, ze které postupně vybíráme minimum, dokud nenalezneme hledaný vrchol.

7.2 Dynamická data

Dynamických objektů se předpokládá jen malé množství a bude téměř vždy přistupováno ke všem v každém snímku. Je jednodušší uchovat je v lineárním spojovém seznamu. Při častém pohybu, zrodu a zániku dynamických objektů, odpadá režie neustálého spravování a vyvažování vyhledávací struktury, která navíc není ani potřeba.

Mezi tyto objekty patří konkrétně samotní hráči, střely a vlajky. V každém snímku bude ke všem přistupováno, musí se zajistit jejich pohyb a vykreslení. Před vykreslováním je samozřejmě snaha vyřadit objekty, které ve výsledku nebudou vidět. O to se stará hned několik optimalizačních metod (7.1.3, 8.3.3, 8.3.6).

7.3 Formát uložení statické scény

Celá struktura scény je uložena v jednom souboru, rozděleném na části identifikované hlavičkou (5.5). Obrázky a modely objektů jsou v dalších souborech, na které scéna

odkazuje. O uložení se stará externí aplikace MapEditor(14.1.2). Hra musí data pouze načítat a uspořádávat do svých datových struktur. Kromě výše zmíněných statických dat (7.1) musí být v souboru uloženy informace o startovních pozicích hráčů a to včetně názvu souboru s modelem hráče, dále pozice vlajek, názvu souboru s obrázkem oblohy a unikátní číselný identifikátor scény, pro synchronizaci během inicializace hraní po síti.

Tato data pak je možné použít podle uvážení třeba v závislosti na herním režimu. Díky rozšiřitelnosti použitého formátu ukládání je možné přidávat v budoucnu další informace. Jednotlivé segmenty v souboru je vhodné mít v tom pořadí, v jakém je pro hru optimální je ukládat do svých datových struktur. Nejprve tedy obecné informace, jako jsou rozměry celé scény, pro inicializaci BSP stromu. Bez této informace by nemohla být statická data zatříděna do této datové struktury a musela by být uchovávána v dočasné paměti, dokud nebude BSP strom vytvořen.

Nyní jsou stanoveny předpoklady pro fungování celé aplikace dle požadavků. Máme data a informace potřebné pro fungování hry, zbývá tedy data použít, zobrazit a rozhýbat je.

8. Grafika

Ve hrách obecně, na rozdíl od užitečného softwaru, se přikládá grafice vysoká důležitost, vždy je cílem aby uživatel hry koukal na pěkné obrázky. Konkrétně zde jde o podstatnou část, na které celá hra stojí. V této kapitole je popsáno, jakými metodami již vytvořené prostředí zobrazovat uživateli na displej [1], [2], [3], [4].

8.1 Render Target

Cílem všeho vykreslování je *render target*, který zaobaluje fyzický display a objekty *backbuffer (8.1.1)* a *z-buffer (8.1.2)*. Je potřeba spravovat tyto objekty, vytvořit je, přistupovat k jejich funkcím a měnit jejich rozměry podle změn parametrů cílového displeje.

Zařízení, na které probíhá vykreslování, je rastrové. Druhou možností je vektorový výstup, se kterým se ale v oblasti dnešních her již nesetkáme.

Backbuffer

Backbuffer je rastrová (dvourozměrná) mapa (*mřížka*), často rozměry shodná s displejem, respektive s oblastí určenou pro vykreslování hrou (čili *oknem*).

Jednotlivé elementy mřížky, dané svou pozicí [*x*, *y*]*, se* nazývají *pixely* (*picture element*) a nabývají různých hodnot v závislosti na zvoleném formátu pixelu. Nejčastěji je pixel ve formátu RGB, čili složen ze tří složek zastupujících intenzitu tří základních barev červená-zelená-modrá. Formát zároveň určuje, kolik bitů je pro jednotlivé barevné složky k dispozici. Dalšími formáty pixelů jsou YUV (televizní vysílání), CMY a CMYK (tiskárny), HSV (pro uživatele) a další [[2].1.1].

Jednotlivé pixely bývají uloženy v jednorozměrném poli o délce $šířka * výška$. Ty se pak adresují podle své souřadnice $[x, y]$ jako $p = x + šířka * y$.

Úkolem backbufferu je eliminování problikávání, způsobeného okamžitým zobrazováním vykreslovaných objektů na displej při absenci backbufferu. Uživatel díky backbufferu tedy uvidí až výsledný obraz vytvořený v rámci jednoho *snímku* (*11.4*), složený vykreslením elementárních objektů, a ne postupně se tvořící mezivýsledky. Takto jsou mezivýsledky skládány v backbufferu a na konci snímku je systémová rutina rychle

překlopí na displej. To, jak je překlopení implementováno, je lepší ponechat na systému, ty v dnešní době mívají pro tyto účely speciální funkce, zpřístupněné *frameworkem* (4.1).

Neméně podstatnou úlohou je přemostit rozdíly mezi formátem pixelů cílového zařízení a formátem pixelů, se kterými je herní engine schopný pracovat a který používá backbuffer. Jde o konverze formátů během překreslení celého backbufferu uživateli na displej.

Zajímavou možností využití backbufferu je vytvoření tzv. efektu televizní obrazovky nebo zrcadla. Backbuffer je stejně jako textura(8.2.4) rastrová mapa, lze tedy do nějakého dočasného *backbufferu* vykreslit dynamickou scénu (třeba z pohledu odraženého pohledu od zrcadla), následně použít výsledný obraz jako texturu pro vykreslení jiné scény.

Řešení viditelnosti

Spolu s backbufferem lze ukládat i informace o vzdálenosti jednotlivých pixelů od pozorovatele (*kamery 8.3.3*). Tím dostáváme paměť hloubky (z-buffer), která se implementuje jako samostatná dvourozměrná mřížka, se shodnými rozměry jako má backbuffer, kde jednotlivé elementy udávají vzdálenost odpovídajícího pixelu v backbufferu. Mluvíme o rastrovém algoritmu viditelnosti [[2].11.3.1, [1].4.2]. Také je nutno uvést formát elementů této mřížky. Lze použít obyčejné reálné číslo s plovoucí desetinnou čárkou, to se ale z důvodů rychlosti výpočtů nepoužívá. Nejčastější jsou to 16ti nebo 32ti bitová celá čísla (někdy bez znaménka), se kterými procesory umějí pracovat rychleji (hlavně ty v mobilních telefonech (ARMI)).

Rastrová paměť hloubky není jediné možné řešení viditelnosti, ale je nejjednodušší, nejspolehlivější a je jediné možné, co lze implementovat na této nejnižší úrovni. Nevýhodou je pouze paměťová náročnost potřebná k uchování celého bufferu.

Podstatou je zobrazení tak, aby se nevykreslovaly pixely skryté v prostoru za jinými již vykreslenými. Při pokusu zapsat pixel do backbufferu se jednoduše zkontroluje jeho vzdálenost s hodnotou již zapsanou na daném místě v z-bufferu a podle toho k vykreslení dojde a přepíše se údaj o vzdálenosti novým, nebo k vykreslení nedojde. Na začátku vykreslování snímku je samozřejmě nutné inicializovat všechny elementy v z-bufferu nějakou maximální hodnotou.

Možnou modifikací paměti hloubky se uvádí, kromě jiných, *hierarchická paměť hloubky* (tuto metodu lze nalézt v obvodech grafických karet ATI). Je to rozšíření o stromovou strukturu (nejčastěji quad-*tree*) postavenou nad výše zmíněnou dvourozměrnou mřížkou, kde každý uzel stromu obsahuje údaj o maximální hodnotě svých potomků, přičemž každý list leží na určité části z-bufferu a obsahuje informaci o maximální hodnotě elementů v této části (tedy v každém uzlu lze říci, jaký nejvzdálenější pixel má smysl ještě do oblasti pod uzlem vykreslovat). Toto přináší velké zrychlení, neboť lze již na úrovni objektů říci, zda má smysl se o vykreslení vůbec pokoušet. Určitá režie ale spadá na udržování této struktury, v závislosti na hloubce stromu nebo pořadí vykreslování (nejlepší je, jak z algoritmu vyplývá, vykreslovat vše odpředu dozadu).

Další řešení viditelnosti a současné optimalizace jsou tedy implementovány na vyšších úrovních, kde se pracuje s celými objekty (7.1.3, 8.3.6) a ne už s jednotlivými připravenými pixely.

- **Render State**

Následující elementy budou při vykreslování řídit svůj výsledný obraz podle určitých parametrů. Některé tyto parametry jsou společné pro více elementů, jiné před vykreslením musí projít nezanedbatelně pomalým výpočtem, a jelikož většinou dochází k mnohonásobnému volání vykreslovacích rutin, znamenalo by to zbytečné zpomalení. Tyto parametry bývá vhodné nastavit společně předem, upravit je pro potřeby programu, výsledky uchovat a až ty použít při vykreslování.

Typickým příkladem bývá nastavení barvy, kterou budou vykresleny jednotlivé elementy. Programátor nebo uživael zadává barvu v určitém společném formátu, který rozsahem pokryje ostatní formáty (ARGB s 32bitovými reálnými elementy, kde A je zkratka pro míru průhlednosti a RGB pro klasické červená-zelená-modrá nebo na mobilních zařízeních plně postačuje ARGB s 8bitovými celočíselnými elementy). Backbuffer, do kterého vykreslení probíhá, bývá v jiném formátu, často shodném s formátem pixelů aktuálního grafického režimu, aby následné překlopení bylo co nejrychlejší. Zadanou barvu je tak nejlepší převést na cílový formát, aby při vykreslování do backbufferu nebylo zapotřebí provádět opakované konverze.

Takových parametrů bývá mnoho. Dalším používaným je možnost vypnutí zapisování hodnot do z-bufferu(8.1.2) při vykreslování nebo vypnutí kontroly hloubky ze z-bufferu.

Tím se dá docílit zrychlení, pokud toto testování viditelnosti není zapotřebí nebo je nežádoucí. Opět je lepší nastavit tyto vlastnosti spolu s dalšími jednotně předem, neboť se program na následné vykreslování může lépe připravit (8.2.1).

Další výhodou předem nastavených parametrů je možnost měnit celý mezivýsledný obraz. Ten je složen z mnoha vykreslovacích elementů, kterým je takto všem najednou jednoduše možné změnit chování, bez nutnosti změny vstupních dat. Celému složenému objektu lze tak změnit barvu, texturu (8.2.4), míru a metodu průhlednosti nebo velikost.

8.2 Dvourozměrné objekty

Když je do čeho kreslit, zbývá vyřešit, jak kreslit. Objekty, které chceme zobrazit na výstup, musí být zpracovány, části skryté za okraji backbufferu oříznuty, viditelná část obrazu převedena do dvourozměrné mřížky (*rasterizace*) a ve formě pixelů spolu s informací o hloubce zapsány na *Render target* (8.1).

Jednotný výstup

Samotné vykreslení (nastavení pixelů) na *render target* (8.1) se umísťuje do společné metody (*Scan line processing unit*), která se chová podle předem nastavených parametrů (8.1.3). Těchto metod je více, každá přesně ušitá na určitou konfiguraci možných hodnot parametrů. Při změně konfigurace program rozhodne, kterou tuto metodu použít a odkaz na ní si zapamatuje jako metodu, která se bude používat pro vykreslování až do další změny konfigurace. Odpadá tak neustálé testování a zpracovávání některých parametrů při každém vykreslování.

Vstup těchto metod musí být dostatečně obecný, aby vystačil potřebám různých vykreslovacích rutin. Při vykreslování se používá rozklad na řádky, kvůli rychlejší adresaci do backbufferu při zapisování pixelů, metoda tedy bývá podle toho uzpůsobena. Údaj o poloze řádku, hloubce na koncích a mapování textury (8.2.4) se předá aktuální vykreslovací metodě, která již správně zapíše pixely do backbufferu a informace o hloubkách do z-bufferu. Tyto metody jsou malé, musí být rychlé, ale je jich větší množství. Pokud máme například 5 společných parametrů určující stav zapnutí a vypnutí, je potřeba 2^5 různých společných metod, mezi kterými se při nastavení parametrů rozhoduje.

Metoda tedy vezme předanou řádku a podle nastavených parametrů správně zapíše pixely do backbufferu a z-bufferu.

- **Body**

Nejjednodušším vykreslovaným elementem je bod. Každý vykreslovací *engine* by měl být schopný vykreslovat samostatné body, i když se tato funkce prakticky nevyužívá. Vykreslení jednoho bodu je zanedbatelně časově náročné, obnáší pouhé nalezení souřadnice a zapsání jednoho pixelu do backbufferu, případně i údaj o hloubce do z-bufferu.

- **Úsečky**

Metod pro vykreslování úseček je více [[2].3.1]. Zde je potřeba rozložit úsečku na jednotlivé řádky, které jsou dále vykresleny aktuální vykreslovací metodou.

Rozklad probíhá jednoduše. Dva koncové vrcholy úsečky je nejprve třeba seřadit podle y-souřadnice. Lineární interpolací postupně seshora dolů stačí spočítat x-souřadnici každého řádku, druhá x-souřadnice řádku je shodná se souřadnicí následujícího řádku.

8.2.3.1 Lineární interpolace

Interpolace se používá ke zjištění hodnoty v místě, kde ji neznáme, na základě hodnot z okolí. V tomto případě máme dva body v rovině a hledáme souřadnice bodů na úsečce mezi nimi. Potřebujeme veličinu, přes kterou chceme interpolovat a druhou veličinu, jejíž hodnotu hledáme. Lineární (jednorozměrná) interpolace pracuje pouze se dvěma hodnotami z okolí. Interpolováním přes souřadnici *y* získáme všechny souřadnice *x* na přímce, použitím rovnice x=x1+y*(x2-x1)/(y2-y1).

Lineární interpolace se vždy stává předmětem optimalizace, neboť v základním tvaru obnáší provádění dělení v každém kroku, což je operace pomalejší než sčítání i násobení. To se dá jednoduše odstranit předpočítáním x-rozdílu (x2-x1)/(y2-y1). Toto číslo udává posun x-hodnoty, ke kterému v každém kroku dojde a jelikož se údaj zakládá na konstantních veličinách, je možné si tuto hodnotu spočítat předem. Pak tedy stačí pro každou další řádku přičíst tento rozdíl.

Dalším problémem je snaha pracovat rovnou s celočíselnými typy. Spočítaný rozdíl musí být přesnější, než umožňují celá čísla, ale neustálé konverze z reálných čísel na celočíselné, ve kterých je třeba předávat informace vykreslovací metodě, jsou pomalé. Řešením může být použití čísel s pevnou desetinnou čárkou. Jde vlastně o celá čísla, jejichž posledních ‚*S*' bitů je stanoveno pro desetinnou část. Celou část lze získat jediným

bitovým posunem o ‚*S*' bitů doprava. I to lze ale odstranit. Použitím 32b čísla se znaménkem a vymezením poloviny na desetinnou část dostaneme číslo s pevnou desetinnou čárkou a dostatečnou přesností. Abychom získali prvních 16bitů (čili celou část) bez bitového posunu, stačí přetypovat ukazatel na toto číslo na ukazatel na celé 16b číslo se znaménkem (u procesorů používajících BigEndian ne, ty ale nebereme v úvahu). V jazyce *C++* si lze práci trochu zpříjemnit použitím struktury *union*, kde stačí položit 32bitové celé číslo se znaménkem přes dvojici 16bitových celých čísel, první se znaménkem, druhé bez znaménka. Pro sečtení stačí sečíst 32bitová čísla. Násobit lze buď pouze celými částmi, nebo obě 32bitová čísla předem posunout o 8 bitů doprava a násobit je, jinak by došlo k přetečení a ztrátě informace. Právě násobení dvou čísel s pevnou desetinnou čárkou se ale během interpolace nepoužívá. Převod z reálného čísla se provede pouhým vynásobením číslem 2^{16} a přetypováním na číslo celé. Pro získání celé části stačí sáhnout na první 16bitové číslo v *union* struktuře. Touto technikou dochází k velkému zrychlení, neboť není třeba pracovat s reálnými čísly, která jsou na ARMI procesorech (cílové mobilní telefony) znatelně pomalejší. Technika je používána u všech lineárně interpolačních metod v textu, kde postačuje takto omezená přesnost a interpolace po krocích přičítáním rozdílu (aditivní).

- **Textury**

Textura je rastrový obrázek, určený k polepení jiných objektů, od obdelníků po obecné polygony v prostoru, a následném vykreslení. Do textury lze i zapisovat, stejně jako do backbufferu. Data textury by herní engine měl být schopen přímo načíst z obrazového souboru (BMP, JPEG). Operační systémy jsou často sami schopny obrázky v různých grafických formátech rychle načítat. Nechává se proto tato funkcionalita na vrstvě *frameworku*, který načítání ponechá na systému a vrátí již načtené a převedené pole pixelů.

Stejně jako backbuffer, musí mít i elementy mřížky textury (*pixely*) nějaký daný formát. Jelikož nejčastější operací je zapisování pixelů z textury do backbufferu, měl by být formát textury shodný s formátem backbufferu, aby odpadla potřeba konverze. Pokud bychom chtěli obraz z textury vždy před vykreslením nějak předzpracovávat, například aplikovat osvětlovací model, stínovat nebo provádět vyhlazení, formát bychom zvolili jiný, nějaký, se kterým se lépe manipuluje, a konverzím bychom se nevyhnuli.

8.2.4.1 Implementace textury

Nejčastější operací je tedy zapisování pixelů textury do backbufferu, musíme být proto schopní rychle vyhledávat hodnoty pixelů v textuře podle souřadnice $[x,y]$. Rastrový obraz je tvořen obyčejným dvourozměrným polem, respektive jednorozměrným o velikosti $šířka * výška$. S obecnými rozměry textury by k nalezení adresy pixelu bylo zapotřebí násobení a sčítání, jelikož adresu počítáme $p = x + šířka * y$. Násobení *šířkou* lze ale nahradit rychlejší operací – bitovým posunem – za předpokladu, že rozměry textury jsou mocninou čísla 2. Údaj o šířce a výšce tak nahradíme údajem, o jakou mocninu čísla 2 jde. Adresu tak získáme vzorečkem $p = (y << bitmul) + x$. Při provádění několika desítek tisíc těchto operací za snímek se zrychlení oproti původnímu vzorečku výrazně projevuje (přibližně o 20%).

Podstatné je i zabránění hledání hodnot pixelů na souřadnicích zasahujících za hranice textury, tedy pro hodnoty menší než 0 a vetší než *šířka*, aby program nehledal adresu mimo data textury. Textury se často vykreslují opakovaně vedle sebe tak, že se mapují (8.2.4.2) souřadnice daleko od sebe, přesahující samotné rozměry textury. Je tedy třeba souřadnice před hledáním pixelu upravit. Testovat pokaždé hodnotu není nutné, jelikož rozměry jsou mocninou čísla 2, lze si pro každou texturu připravit bitovou masku, která pokrývá číslo udávající rozměr (pro šířku 2^N bude bitová maska pokrývat zprava *N* bitů). Nové hledání adresy podle souřadnice bude tedy ve tvaru $p = (x\&bitmask)|((y\&bitmask) << bitmul)$, kde ‚+' mohlo být ještě nahrazeno binárním ‚*nebo*'.

8.2.4.2 Mapování textury

Metoda, kterou se určuje, jak se textura nalepí na jiné objekty, se nazývá *mapování* [[2].13.1]. Těchto metod je více, nejpodstatnější je mapování zadáním souřadnic textury $[u,v]$ v různých vztyčných bodech na povrchu objektu a následným bilineárním interpolováním mezi těmito souřadnicemi (neboli interpolací mezi dvěma již lineárně interpolovanými hodnotami), podle relativní polohy zkoumaného bodu na povrchu. Pro vykreslování *potexturovaného* obdelníku stačí definovat souřadnice textury v rozích

obdelníka a v každém vykreslovaném bodě si vypočítat souřadnice textury bilineární interpolací mezi souřadnicemi v rozích.

8.2.4.3 Děravé a poloprůhledné textury

Při vykreslování jsou občas potřeba tzv. děravé textury nebo textury v různých místech s různou mírou průhlednosti (*alpha, 8.2.7.3*). Je proto každý pixel obohacen o hodnotu udávající míru průhlednosti. Jelikož je formát pixelů textury shodný s formátem backbufferu, je potřeba údaje o průhlednosti uložit stranou. Vznikne tak pole rozměrově shodné s rozměry textury, jehož elementy udávají průhlednost pixelů textury (*alpha layer*). To má i výhodu v tom, že toto pole program vůbec nemusí alokovat, pokud průhlednost dané textury nevyžaduje a tím se ušetří paměť, kterou mobilní telefony tolik nedisponují. Při vykreslování může aktuální vykreslovací metoda (8.2.1) tyto informace použít podle potřeby. Tedy metoda, která je použita po nastavení *render states* (8.1.3) týkajících se průhlednosti.

- **Text**

Vykreslování znaků na backbuffer se dá realizovat několika způsoby. Nejjednodušší je ponechat práci na cílovém operačním systému, což je ale dost omezené řešení, které nezaručuje, že bude výsledek na každém zařízení stejný a zabraňuje v použití všemožných transformací a efektů. Proto se volí jiná cesta.

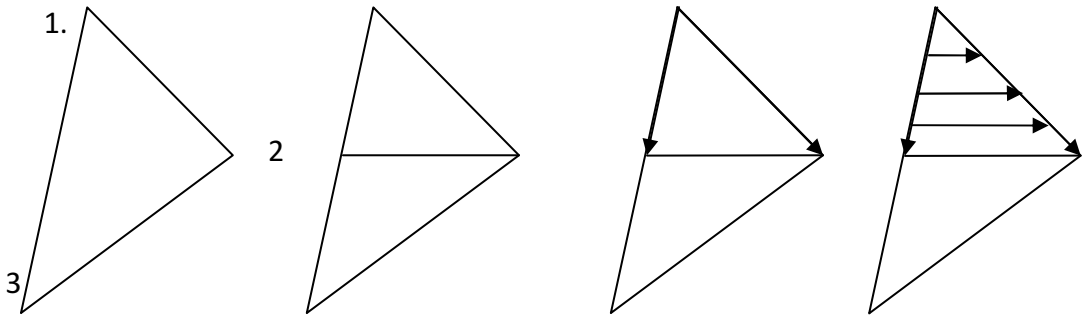
Používá se textura (8.2.4), do které jsou předem znaky vykreslené, respektive se načte takováto textura z předem připraveného souboru. Tím se získá tabulka s rastrovým fontem. Požadovaných znaků je 256, optimální je použít texturu o rozměrech 256×256, znaky umístěné po sobě v řádcích, kde každý bude zabírat prostor o rozměrech 16×16. Souřadnice levého horního rohu znaku s ordinální hodnotou ‚*C*' bude [(C&0x0f)<<4,(C&0xf0)]. (Verze pro UNICODE by se do žádné rozumné textury nevešla.)

Pokud máme texturu s fontem, lze jednoduše použít obdelník nebo jiný vykreslovací element a texturu na jeho povrch podle výše zmíněných souřadnic namapovat (8.2.4.2). Tímto způsobem se vykreslí každý znak, včetně všeho, co umožňují stávající vykreslovací rutiny. Tato metoda je dokonce rychlejší než poskytuje operační systém, a několikanásobně rychlejší pokud je namísto vlastního vykreslování používána hardwarová akcelerace.

Trojúhelníky

Často používané je vykreslování trojúhelníků. Lze z nich sestavit libovolný polygon a spolu s texturami (8.2.4) a parametrizací vykreslování (8.1.3) je možné tvořit téměř jakýkoliv obraz. Nejvíce využití se najde v oblasti 3D grafiky (8.3, 9.1, 9.2).

Problémem je tedy rychle trojúhelník rasterizovat, rozložit na řádky a ty nechat vykreslit aktuální vykreslovací metodou (Obrázek 8-1) [[1].3.1-3.4].



Obrázek 8-1: Postup při vyplňování trojúhelníku

Postupuje se podobně jako v případě úsečky (8.2.3). Vrcholy trojúhelníku, které definují souřadnici [*x*, *y*, *z*] a mapovací souřadnici textury [*u*,*v*], jsou seřazeny podle y-souřadnice. Prostřední vrchol rozdělí element na dva trojúhelníky s rovnoběžnou základnou. Pak se pro každý zvlášť postupuje odshora, po řádcích se podle y-souřadnice protilehlých stěn interpolují *x* a *z*-souřadnice a také mapovací souřadnice textury pomocí upravené lineární interpolace (8.2.3.1). Každý takto získaný řádek se předá vykreslovací metodě, která řádku dále interpoluje podle x-souřadnice, čímž získá výsledné mapování textury pro každý pixel na řádce.

Jediné, co by metodu zpomalovalo, je lineární interpolace a konečné zpracovaní vykreslovací metodou. Oboje je dostatečně optimalizované, vykreslovací metoda je dále závislá na nastavených parametrech.

Ve vrcholech bývá obvykle definována i barva a také normálový vektor kvůli osvětlovacímu modelu, který udává kolmici na povrch v bodě. Tyto údaje, stejně jako mapování textury, se po složkách bilineárně interpolují mezi vrcholy. Ve hře nejsou ale potřeba a interpolace navíc by zbytečně zpomalovaly, plně postačuje mapování textury.

Obrázek 8-2: Rozdíl v mapování textury bez použití korekce perspektivy (vlevo) a s použitím korekce perspektivy (vpravo)

Dalším problémem vyskytujícím se až v 3D grafice (8.3), tedy pokud využíváme i údaj o hloubce (z-souřadnice) v každém vrcholu, je korekce perspektivy. Pokud mapovací souřadnice textury interpolujeme lineárně, na větších plochách a s většími rozdíly mezi údaji o hloubce se objeví chyby, v podobě nerealistického zobrazení textury na povrchu (Obrázek 8-2).

Chyba se dá odstranit (*perspective correction*) interpolováním mapovacích souřadnic vynásobených z-souřadnicí v daném vrcholu ($[u_1,v_1]=[u_0*z_0,v_0*z_0]$, čili vrátit zpět aplikování perspektivy v provedené projekci 0). Správná hodnota se pak před vykreslením

každého pixelu získá zpětným vydělením upravené interpolované mapovací souřadnice interpolovanou z-souřadnicí ([u,v]=[u_1/z, v_1/z]). Údaj o hloubce (z-souřadnice) se také musí opravit. Nedělí se sama sebou, ale místo toho se interpoluje její převrácená hodnota ($z_1=1/z_0$), která se v každém pixelu musí zpátky převrátit, abychom získali správnou hodnotu ($z=1/z_1$). Jak je vidět, tento postup obnáší mnoho nežádoucích operací navíc, zase na druhou stranu zaručuje realistický výsledek. Jsou dvě možnosti, jak výpočet urychlit. U menších polygonů korekci zcela vynechat, chyba by nebyla příliš vidět, jen při pohybu se objeví malé odchylky od toho, co by lidské oko očekávalo. U větších polygonů, kde by chyba byla téměř neúnosná, se postupuje následovně. Korekce se provede jen každý například 16tý řádek a 16tý pixel na řádce. Mezi těmito správně (a pomalu) interpolovanými vztyčnými body se dále provádí obyčejná rychlá lineární interpolace. Chyba se sice projevuje, ale pouze po bližším prozkoumání vykreslené scény. Tato metoda se používá v softwarovém vykreslovači (*rendereru*) hry Quake II [[1].3.3.1.1)]. V této práci je použita pouze při vykreslování stěn, podlahy a stropu (8.3.5), které jsou velké. U ostatních polygonů, které jsou malé, chyba nepůsobí rušivě.

- **Úpravy obrazu**

Existuje mnoho algoritmů na různé úpravy rastrového obrazu [[2].4]. Zde bych uvedl některé, které lze snadno implementovat pomocí stávajících funkcí. Tedy jak jednoduše a rychle provádět úpravy na texturách a backbufferu.

8.2.7.1 Ztmavení

Pokud máme jednotlivé pixely rastrového obrazu ve formátu RGB, kde každá složka určuje míru zastoupení jedné ze základních barev, lze jednoduše celý obraz ztmavit. Efekt se často používá pro zvýraznění části displeje tím, že se zbytek nechá takto utlumit, nebo jako podklad pod text, aby byl lépe čitelný. Obecně tento problém řeší *alpha míchání* (8.2.7.3), překreslením plochy poloprůhledně černou barvou.

Ke ztmavení pixelu dojde, pokud stejnou mírou snížíme hodnotu každé jeho složky. Nabízí se snižovat vždy na polovinu nebo na čtvrtinu, jelikož pak postačuje provést bitový posun doprava na každou barevnou složku. Abychom se vyhnuli zbytečnému vyseparovávání barevných složek a pak skládání RGB hodnoty zpět, je lepší provést jeden bitový posun na celý pixel najednou a poté nebo předtím přes bitovou masku oříznout

nízké bity každé složky, které se po posunu dostaly do pozice nejvyšších bitů sousední složky RGB hodnoty. Celý efekt se dá provést jednoduchou operací (C>>1)&bitmask. Bitová maska je závislá na použitém RGB formátu, přesněji na velikosti každé složky. Pro velikosti R8-G8-B8 (tedy s každou složkou o velikosti 8 bitů) by ‚*bitmask*' měla hodnotu 7F7F7F.

8.2.7.2 Změna rozměrů, otočení

Obraz z textury je nyní také velmi jednoduché vykreslit v libovolných rozměrech, polohách či natočení. Rohy textury stačí namapovat (8.2.4) na vrcholy dvou trojúhelníků, uspořádaných do tvaru kvádru. Ten pak za pomocí stávajících vykreslovacích funkcí (8.2.6) nechat vykreslit na libovolném místě, v libovolném tvaru, libovolné hloubce, přes a skrz jiné objekty, včetně všeho co nabízí parametrizace (8.1.3) a vykreslovací metody (8.2.1).

8.2.7.3 Poloprůhlednost

Pro zobrazení pixelu průhledně (respektive poloprůhledně) je zapotřebí dodatečná informace o míře průhlednosti. Spolu s ní, lze v této míře po složkách smíchat vykreslovaný pixel s pixelem na pozadí. Tato míra, označovaná ‚A' (*jako Alpha, od toho alpha míchání*) v intervalu [0,1] říká, jak moc má být nový pixel vidět a zároveň jak málo má zůstat z původního pozadí. Výsledná hodnota každé barevné složky C se tedy spočítá $\mathbf{C = (1 - A) * C^1 + A * C^2}$, kde C^1 je barva pozadí a C^2 nová barva s mírou průhlednosti A.

Tento výpočet je oproti původnímu $C_RC_GC_B=C_RC_GC_B{}^2$, který pouze nový pixel kopíroval na jiné místo, velkým zpomalením. Obecně používání poloprůhlednosti velmi zpomaluje. Navíc, máme-li více poloprůhledných ploch, které chceme zobrazovat přes sebe, musíme je vykreslovat postupně od té nejvzdálenější, takže se před vykreslením nevyhneme třídění podle hloubky.

Alespoň první část je možné urychlit. Separovat od sebe barevné složky, aplikovat na ně vzoreček zahrnující násobení a pak zase složky skládat dohromady je zbytečné. Urychlení spočívá v reprezentaci čísla A celočíselně v intervalu [0;255] a následnou kombinací několika bitových posunů, bitových masek a násobení celočíselnou průhledností, v závislosti na konkrétním RGB formátu.

8.3 3D

Nyní je na řadě zobrazit prostorové objekty na dvourozměrném zařízení, konkrétně backbufferu. Převodu trojrozměrných objektů do dvojrozměrné podoby se říká promítaní (*projekce*, 0). Před touto projekcí jsou polohy objektů v prostoru často ještě upraveny (*transformace*, 8.3.1).

Transformace

Jedněmi z nejčastěji používaných operací v počítačové grafice jsou geometrické transformace [[1].2.2-2.3, [2].19, [4].7.4]. Těch je více, zde si vystačíme hlavně s lineárními, mezi které se řadí posunutí, otáčení, změna měřítka, zkosení a transformace z nich složené. Další transformací je projekce (8.3.2) určená k převodu trojrozměrného obrazu na dvojrozměrný. Celý objekt pak transformujeme aplikací transformace na body, ze kterých je objekt složený.

Při reprezentaci bodu kartézskými souřadnicemi $[\mathbf{X},\mathbf{Y},\mathbf{Z}]$, stačí k vyjádření transformace jediná matice, protože jde o transformace lineární a souřadnice jsou *homogenní*. *Transformační matice* pak udává koeficienty lineárních rovnic, definujících transformaci.

Uspořádanou čtveřici čísel $[x,y,z,w]$ nazýváme pravoúhlé homogenní souřadnice bodu P s kartézskými souřadnicemi $[\mathbf{X},\mathbf{Y},\mathbf{Z}]$ ve třech rozměrech, platí-li: $X=\frac{x}{w}, Y=\frac{y}{w}, Z=\frac{z}{w}, w\neq \mathbf{0}$

Bod P je svými homogenními souřadnicemi určen jednoznačně. Souřadnici *w* nazýváme váhou bodu P. Homogenní souřadnice transformovaného bodu P‘ s kartézskými souřadnicemi $[X',Y',Z']$, se označují $[\mathbf{x'},\mathbf{y'},\mathbf{z'},\mathbf{w'}]$. Často se volí $w=\mathbf{1}$, potom jsou homogenní souřadnice bodu $[\mathbf{X},\mathbf{Y},\mathbf{Z},\mathbf{1}]$. [[2].19.1]

Matice *A* reprezentující lineární transformaci bodu P je regulární matice 4×4 ve tvaru:

$$A=\begin{bmatrix} a_{11} & a_{12} & a_{13} & \mathbf{0} \\ a_{21} & a_{22} & a_{23} & \mathbf{0} \\ a_{31} & a_{32} & a_{33} & \mathbf{0} \\ a_{41} & a_{42} & a_{43} & \mathbf{1} \end{bmatrix}$$

Aplikace transformace na bod (*vektor*) a aplikace skládání transformací se řeší klasickými funkcemi pro násobení matic. V tomto případě transformaci bodu P provedeme jako součin $P' = PA$.

Sestrojení opačné transformace je pouze inverze matice.

Transformaci složenou získáme postupným vynásobením dílčích transformačních matic. Samozřejmě je třeba dát pozor na pořadí, ve kterém se násobení provádí [[4].7.5].

8.3.1.1 Posunutí

Transformace posunutí je dána vektorem posunutí $[x, y, z]$. Matice *T* reprezentující posunutí (*translaci*) se nazývá *translační matice* a je definována takto:

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ x & y & z & 1 \end{bmatrix}$$

8.3.1.2 Otáčení

Otáčení (*rotace*) kolem dané souřadnicové osy je dáno úhlem α . Libovolné otočení pak získáme jako složení dílčích rotačních transformací. Jejich matice R_X,X_Y,R_Z pro rotaci kolem os X, Y, Z jsou v tomto tvaru:

$$\square_X = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha & 0 \\ 0 & -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, R_Y = \begin{bmatrix} \cos\alpha & 0 & \sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, R_Z = \begin{bmatrix} \cos\alpha & \sin\alpha & 0 & 0 \\ -\sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

8.3.1.3 Změna velikosti

Vektorem $[s_X, s_Y, s_Z]$ (scale) je dána transformace měnící velikost ve třech souřadnicových osách. Scale transformace je ve tvaru:

$$S = \begin{bmatrix} s_X & 0 & 0 & 0 \\ 0 & s_Y & 0 & 0 \\ 0 & 0 & s_Z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- **Projekce**

Funkce, která bod v trojrozměrném prostoru převede na dvojrozměrný, se nazývá projekce (*projection, promítání*). *Průmětna* je rovina, na kterou je scéna promítnuta. Vzhledem ke zjednodušení výpočtu projekce, umísťuje se průmětna kolmo na osu ‚*z*' (čili

na osu udávající vzdálenost) a do počátku souřadnicového systému. To se provede tak, že se transformuje celá scéna tak, aby se průmětna dostala do této polohy (*Kamera* 8.3.3).

Těchto funkcí existuje mnoho a libovolné si lze nadefinovat. Ve virtuální realitě, kde se klade důraz na podobnost s reálným světem, se používá *středové promítání*, kde se všechny paprsky kolmé na průmětnu sbíhají v nekonečnu. Mluvíme tedy o perspektivě. Pokud si umístíme střed průmětny do počátku souřadnicového systému, výpočet obnáší pouze vydělení souřadnic $[x, y]$ hloubkou z . [[1].2.1, [2].9.2]

Kamera

Před projekcí je třeba celou vykreslovanou prostorovou scénu umístit tak, aby průmětna byla umístěna do počátku souřadnicového systému a natočit tak, aby průmětna byla rovnoběžná s osami XY (tedy kolmá na osu Z) a viditelná polorovina sahala do daného směru, například do kladné poloosy Z. Až pak je možné jednoduše provést projekci.

Tuto transformaci definuje poloha kamery (*pozorovatele*) [[1].2.4.1]. Ta je určena pozicí v prostoru, ze kterého je scéna pozorována, směrem pozorování a rotací kolem směru (osy) pozorování. Rotace kolem osy pozorování je dána vektorem, udávajícím směr ‚*nahoru*', kolmým na směr pozorování. S těmito třemi vektory lze sestavit transformační matici. Rotace je určena směrem pozorování a směrem ‚*nahoru*'. Posunutí pak podle souřadnice kamery.

Inverzí této transformace (tedy inverzí této transformační matice) získáme transformační matici, která pozorovanou scénu transformuje do požadované polohy připravené k projekci.

Dodatečnou informací vyplývající z konkrétní polohy kamery je definice pohledového jehlanu (*pohled*, *view frustum*) [[1].6.2 (view cone), [2].9.3 (pohledový objem), [6] (viewing frustum)]. Komolý jehlan přesně obaluje část scény, která je viditelná s použitím dané kamery a projekční funkce. ‚*View frustum*' se pak používá k optimalizacím při vykreslování (8.3.6), kde jsou celé skupiny objektů vyloučeny z vykreslování ještě dříve, než dojde k samotným transformacím. Stačí totiž zkontrolovat, zda objekt nezasahuje do jehlanu, pak je jistota, že by po vykreslení nebyl vidět. Místo testování, zda do jehlanu zasahuje objekt, používá se tzv. obálka objektu (*bound box*) [[2].14.3.1]. Ta je definována kvádrem, těsně obklopujícím objekt. S kvádrem je totiž rychlejší testovat existenci průniku

s jehlanem, jako logický součin příslušnosti vrcholů kvádru k jednotlivým polorovinám určeným stěnami komolého jehlanu. Pokud jsou všechny vrcholy v jedné stejné polorovině, kvádr je pak jistě mimo viditelnou část scény.

- **Rendering**

Za použití výše popsaných metod je snadné vykreslovat dvojrozměrné objekty zadané v trojrozměrném souřadnicovém systému. Stačí nadefinovat požadovanou transformaci, kameru a projekci. Souřadnice objektů transformovat a rasterizovat. Před tím a během těchto operací jsou ještě prováděny optimalizace vyřazováním zakrytých a jinak neviditelných částí scény (8.1.2, 8.3.6).

- **Stěny, podlaha**

Tělesa stěny a podlahy je možné vykreslovat standardní cestou, definovat je jako trojúhelníky a nechat zbytek práce na již hotových vykreslovacích rutinách. Jelikož jsou ale velmi často vykreslovány a jejich parametry se liší od ostatních ve hře použitých, je vhodné jejich vykreslování vypracovat zvlášť.

Stěny a podlahy tvoří rozsáhlé plochy a je proto potřeba aplikovat korekci perspektivy (8.2.6). Také jsou vykreslovány vždy stejným způsobem, další parametrizace (8.1.3) tedy není potřeba. Nakonec, jelikož většina výsledného obrazu bude vlastně tvořena stěnami a podlahou, je důležité jejich vykreslování co možná nejvíce urychlit.

Co se týče korekce perspektivy, lze ji v tomto případě zjednodušit. Není třeba ji provádět každý 16tý řádek a 16tý sloupec. Stěny jsou vždy svislé a kamera se neotáčí. Postačuje tedy opravovat interpolaci v případě stěn jen každý 16tý sloupec a v případě stropů a podlahy jen každý 16tý řádek.

Také případné stínování lze urychlit. Obecný vzoreček pro míchání dvou barev v poměru daném alpha úrovní (8.2.7.3) je zbytečný. K plynulému ztmavení jednotlivých pixelů je možné druhou polovinu vzorečku vynechat, neboť by počítala a přičítala černou barvu, která má v RGB formátu hodnotu nula.

Největší optimalizace spočívá ve využití faktu, že plné stěny (7.1.3) zakrývají scénu za sebou (tedy i další stěny). Před vykreslováním je tedy nejdříve vytvořen seznam stěn přivrácených a viditelných danou kamerou. A dále jsou v tomto seznamu ořízнuty nebo odstraněny stěny zakryté jinými. Stěny v seznamu, respektive jejich zbytky, stačí nakonec

vykreslit bez dalších testů viditelnosti. Tento seznam oříznutých stěn se pak uchová a použije se k testování viditelnosti dalších objektů, které se vykreslí pouze, pokud nejsou nějakou stěnou v seznamu zakryty.

- **Optimalizace**

Zrychlení na vyšší úrovni vykreslování spočívá v eliminaci těles z vykreslovacího procesu o kterých dovedeme rozhodnout, že nebudou ve výsledku vidět. Ideální by bylo, kdyby se na tato tělesa vůbec nepřistupovalo. Jelikož máme celou scénu připravenou v hierarchické struktuře, přesněji ve stromu s uzly udávajícími rozsah dceřiných uzlů a listů pod sebou (7.1.1), a známe pohledový kužel kamery, docílíme částečně této optimalizace eliminace objektů. Od kořene jsou procházeny pouze uzly, zasahující do pohledového kuželu. Tímto postupem se dostaneme pouze do listů zasahujících do pohledu. Další zrychlení na této úrovni vykreslování řeší tělesa stěny (8.3.5), které vyřadí z procesu vykreslování objekty zakryté nějakou plnou stěnou.

Vykreslování těles složených z trojúhelníků se, kromě jiného, často zrychluje úplným vynecháním trojúhelníků odvrácených od kamery [[1].6.1]. To jsou ty, které mají po transformaci pořadí vrcholů deklarováno proti směru hodinových ručiček (*CCW, counter-clockwise*) (nebo opačně (*CW*, *clockwise*), podle vstupních dat) respektive ty s normálovým vektorem (kolmice na povrch) odvráceným od kamery. Tyto definice jsou identické, každá vyplývá z té druhé, jelikož kolmici na povrch trojúhelníku získáme jako součin vektorů hran a orientace této kolmice je dána právě pořadím vrcholů, ze kterých se hrany počítají. Pro připomenutí, tato kolmice se počítá $\vec{N} = \vec{E_1} \times \vec{E_2}$, kde $\vec{E_1}$ a $\vec{E_2}$ jsou hrany trojúhelníka. Kolmice se často uvádí v normalizovaném tvaru, tedy upravená tak, aby délka $|\vec{N}| = 1$. Trojúhelníky odvrácené od kamery je možné vynechat, neboť v prostoru jistě existují ještě takové, které odvrácenou stěnu před kamerou zakrývají (jednoduše, nikdy neuvidíme odvrácené stěny zavřené krabice, pokud nejsme v ní).

- **Techniky vykreslování**

Výsledný obraz je nyní závislý na způsobu použití implementovaných funkcí. Jejich různými kombinacemi lze docílit různých výsledků. Zde bych uvedl alespoň jednu zajímavou a ve hře použitou techniku.

Klasickým přímým zobrazením prostorových těles za použití výše popsaných funkcí je sice rychle získán obraz, ale cílem je také, aby byl výsledek něčím zajímavý. Jednou možností, a ve hře použitou, je vykreslování s efektem kresleného filmu (*cartoon rendering*). To se za použití softwarového vykreslování řeší poměrně jednoduše.

Prvním krokem je použití nižšího počtu barev, preferovány jsou pastelové. Textury je třeba upravit, snížit počet barev a obrázky připravené do hry mít jako by namalované. Samotná úprava vstupních dat ale nestačí. Důležité je nějakým způsobem obtáhnout hrany těles černou čarou. Hrana zobrazovaného tělesa, která má být obtažena, je každá hrana vykresleného dílčího polygonu, jejíž sousední polygon je odvrácen od kamery. Při načítání takového objektu si tedy stačí u každého polygonu zapamatovat sousedy k jeho hranám a poté během vykreslování zaznamenávat, které polygony jsou odvrácené. Následně jsou obtaženy hrany, jejichž jeden přilehlý polygon je odvrácený a druhý ne, tak vznikne jednoduchý efekt kresleného filmu.

9. Prostorový model

Celý prostorový model není složen jen ze samotného tělesa. Model popisuje i mapování textur a odkaz na samotné textury, případně obecně materiály, ze kterých je model složen. Také dodatečné informace jako hmotnost objektu může být součástí popisu. Případně zde nalezneme i nějakou vnitřní strukturu kostí (9.2.1) a možných pohybů na ně aplikovaných včetně ozvučení (*animace,* 9.2.2). Dostaneme tak samostatnou strukturu, která obecně popisuje objekt, včetně jeho fyzických parametrů, a nejen vzhled.

9.1 Hraniční reprezentace tělesa

Nejčastěji používanou reprezentací prostorového tělesa v počítačové grafice je hraniční (plošková), kde je povrch tělesa tvořen polygony [[2].6.1]. Místo obecných polygonů se spíše používají trojúhelníky, protože u nich se rychleji řeší rasterizace, zjišťuje orientace, konvexita a hledají průsečíky s přímkou. Také hardwarové akcelerátory i softwarové vybavení počítače s nimi často umí lépe pracovat. Je to tedy reprezentace vhodná, pokud je vyžadována vysoká rychlost.

Tvar

Povrch určují trojúhelníky (*plošky*,*face*) umístěné v prostoru, které jsou vykreslovány výše uvedenými metodami. Nejsou ale uloženy zvlášť, jelikož několik sousedních trojúhelníků vždy sdílí společný vrchol, docházelo by k redundanci dat. Těleso je tedy tvořeno polem s unikátními vrcholy a mapou po trojicích indexů do tohoto pole. Každá trojice indexů definuje jeden trojúhelník. Vrcholy (*vertexy*) nesou informace o poloze vrcholu v prostoru $[x, y, z]$, mapovacích souřadnicích textury $[u, v]$ v tomto místě povrchu, dále také normálovém vektoru (kolmici k povrchu) $[x_n, y_n, z_n]$, pokud je potřeba například kvůli osvětlovacímu modelu, a případně další informace, jako jsou barvy $[d_A, d_R, d_G, d_B]$.

Dodatečnou informací může být i identifikátor transformační matice (8.3.1), která se má pro který vrchol před vykreslením použít. Různé části modelu pak mohou být jinak transformovány. To se využívá při animaci (9.2).

Často využívaná je mapa sousedních trojúhelníků (*adjacency buffer*), kde je pro každý trojúhelník k dispozici trojice indexů, odkazující na jiné trojúhelníky, které mají spolu s prvním společné dva vrcholy a jsou tedy těsně vedle (jsou vedlejší, sousední). Kromě různých optimalizačních metod se využívá *adjacency buffer* při technice kresleného filmu (8.3.7).

- **Textura**

Každý trojúhelník nebo skupiny trojúhelníků mívají danou texturu. Ta se pak použije při vykreslování trojúhelníků (8.2.6) s danými mapovacími souřadnicemi ve vrcholech.

Občas se definuje více vrstev s různými texturami. Tyto vrstvy je možno při vykreslování míchat dohromady nebo použít vždy jen jednu a vykreslovat tak model podle toho v různých převlecích. Tvar tělesa tak zůstane stejný a jednoduše lze měnit vzhled. To se využívá například při převlékání počítačových postav.

9.2 Animace

S použitím transformačních matic lze prostorový model rozhýbat. Lze jím posouvat, otáčet a provádět další lineární transformace (8.3.1). To ale není vše, díky vrstvám textur je dále možné animovat i obrázek na povrchu objektu pouhou cyklickou změnou použité vrstvy v každém snímku.

Zajímavější, a v této bakalářské práci také implementované, je použít různé transformační matice na různé části tělesa, tedy na různé vrcholy [[6](SkinnnedMash), [2].17]. Transformovat se dají nejen souřadnice vrcholu, ale i barvy, normálové vektory a mapovací souřadnice textur. Tím lze docílit opravdové volnosti při animování modelů. Tímto způsobem jsou pak vytvářeny například animace počítačových modelů zastupujících živé herce ve filmech, včetně animací složité mimiky obličeje. Otázkou zůstává, jak tyto transformace definovat a jak určit vztahy mezi nimi. Následující řešení je primárně určeno pro animace souřadnic vrcholů modelů postav (resp. všeho se strukturou ‚*kostí*', což mohou být zvířata ale i auta). Podle potřeby, pohyb jiných elementů (barva, normálový vektor, …) se provádí podobně, většinou i jednodušeji, ale využívá se zřídka.

- **Soustava kostí**

Nejprve je potřeba definovat vztyčné body, jejichž pohyb ovlivní polohu okolních vektorů. Každý tento bod pak určuje transformační matici, která se použije na okolní body. Občas se podle potřeby na vrcholy mezi dvěma vztyčnými body použije transformace smíchaná ze dvou okolních.

Další možností je, jako vztyčné body definovat tzv. *kosti*. Ty jsou určeny polohou, směrem, rotací kolem osy, délkou a rodičovskou kostí, což ve výsledku tvoří hierarchickou acyklickou strukturu kostí (*kostru*). Každá kost pak má určenu množinu vrcholů, jejichž polohu transformační matice této kosti ovlivňuje. Výhoda spočívá například v tom, že změna polohy jedné kosti ovlivní rekurzivně i kosti dceřiné. Tato struktura vyhovuje většině modelů.

- **Definice animace**

Animace modelu se nyní převádí na animaci jeho kostí. Jelikož ty pak ovlivňují polohu vrcholů.

Animaci lze definovat libovolně, například výčtem poloh kostí (*poloha*, *směr*, *rotace kolem osy*) v každém snímku. Zajímavější a praktičtější je ale tyto polohy zaznamenávat jen pokud dojde k výrazné změně, v tzv. *klíčových snímcích* (určených časovou značkou). Poloha kosti v libovolném čase se zjistí interpolací složek souřadnice kosti ze dvou sousedních klíčových snímcích. Samozřejmě záleží na použité interpolační technice. Většinou stačí lineárně interpolovat jednotlivé složky souřadnice. Výsledek ale vypadá lépe, pokud se místo interpolace po přímce použije například bèzierova křivka.

- **Transformace modelu**

Když je v každém snímku získána relativní poloha každé kosti ke svému rodiči, zbývá pro každou kost zjistit transformační matici a aplikovat ji na vrcholy modelu.

Zde se využije skládání transformací (8.3.1). Absolutní transformace určená každou kostí je složena ze své relativní transformace a absolutní transformace svého rodiče, což se provede vynásobením těchto transformačních matic. Transformační matici M_{B_i} každé kosti B_i s relativní transformací R_{B_i} a rodičem B_j s transformační maticí M_{B_j} získáme

složením $M_{B_i} = R_{B_i} M_{B_j}$. Vyjímku tvoří páteřní kost (základní, kořenová), která nemá rodiče a její relativní poloha je totožná s absolutní ($M_{B_0} = R_{B_0}$).

Relativní transformace R_{B_i} každé kosti B_i je určena polohou, směrem a rotací kolem osy, z těchto údajů není problém spočítat transformační matici (8.3.1).

Tyto získané transformace každé kosti není ještě možné aplikovat na model, neboť on sám ve své původní podobě je vlastně od začátku již transformován základní polohou kostí. Proto se při inicializaci modelu musí nejprve vrcholy transformovat opačnou transformací určenou polohou kostí v základním tvaru. Pro kost B_i, její konkrétní absolutní transformační matici M_{B_i} a iniciální transformační matici této kosti $M^{0}_{B_i}$, je souřadnice každého vrcholu P náležícího kosti B_i vyjádřena vztahem $P' = \left(PM^{0^{-1}}_{B_i}\right)M_{B_i}$, kde právě ($P$ [illegible]) se spočítá pouze jednou při inicializaci.

9.3 Další informace popisující model

Model jako takový není určen jen k vykreslování. Často popisuje i fyzické parametry. Proto bývá jeho součástí i jméno, hmotnost, definice směru předku nebo souřadnice těžiště. Díky tomu lze věrněji řešit například fyzikální problémy, jelikož jsou parametry na míru ušité každému modelu. Také změnou použitého modelu se změní i fyzikální parametry. Proto je vhodně tyto informace udržovat pohromadě.

9.4 Formát uložení modelu

Každý model bývá uložen v samostatném souboru [14.2.4]. Jako první se ukládá hlavička s dodatečnými informacemi a počtem vrcholů, trojúhelníků a kostí. Aplikace se tak mohou na následující data připravit. Následují pole s unikátními vrcholy a pole trojúhelníků tvořené indexy (9.1.1). Dále se ukládají tzv. materiály, kde se pro dané množiny trojúhelníků (často dané intervalem) určí soubor s texturou a další parametry společné pro tuto množinu. Také se ukládá pole indexů mapující každý vrchol na jeho příslušnou kost (čili transformační matici) a nakonec iniciální polohy kostí a animace složené z klíčových snímků.

9.5 Souhrn

Prostorové modely jsou podstatnou částí počítačových 3D her. Často představují samostatné žijící objekty, se kterými se uživatel může i ztotožňovat. Mají svůj tvar, své vlastní specifické pohyby, fyzikální vlastnosti i jméno. Je proto i dobré je implementovat samostatně, se vším co je může od sebe odlišovat.

10.Detekce kolize

Jakmile některý objekt ve scéně změní polohu, musí být zkontrolováno, zda nedošlo ke kolizi a následně případnou kolizi vyřešit.

10.1 Hledání kolize

Kolize se hledají klasickými goniometrickými funkcemi, v závislosti na tvaru dvou testovaných objektů. Jelikož ty ale mají v počítačové grafice často obecný tvar složený z polygonů, používají se místo toho elementární tělesa; koule, válec, kvádr (*obálka objektu*), někdy postačuje i bezrozměrný bod. Pokud záleží na přesnosti, je hledání rozděleno do dvou fází, kdy se kolize nejprve najdou nahrubo, s použitím těchto jednoduchých těles, a následně až se provede detekce přesněji.

Problémem je, že se nehledá pouze průnik dvou objektů, ale vzdálenost průniku trajektorií tvořených pohybem dvou objektů během posledního *snímku* (*11.4*). Naštěstí mají tyto trajektorie často tvar přímky. Také proto je nejjednodušší hledat kolize trajektorie bodu s jinými objekty. Většinou také postačuje zanedbat pohyb druhého objektu a přistupovat k němu jako k nepohyblivému a jako k elementárnímu tělesu, jako je rovina, kvádr nebo válec.

Ve hrách se setkáme s několika různými přístupy ke hledání kolize. Jedním je nalezení kolize samotného hráče se stěnami a podlahou. Je tak možno kolize vyřešit a zabránit průchodu stěnami a propadem podlahou. Hráč bývá zjednodušen na válec, kouli či bod. Nalezení kolize se stěnou (rovinou, 7.1.3) a podlahou (také rovinou, 7.1.2) je proto rychlé. Dále se hledají existence kolize střel se stěnami, podlahou a jinými hráči. Střela je jednoznačně bod a její testovaná trajektorie v posledním snímku je úsečka. Se stěnou a podlahou je nalezení kolize rychlé. Pro nalezení kolize s jiným hráčem se použije obálka hráče.

Při hledání kolize se musíme vyhnout testování všech objektů se všemi. Proto se nejprve každému objektu počítají obálky samotných trajektorií, jako roztažení obálky objektu před pohybem do směru pohybu. Testovány jsou pak pouze dvojice objektů, jejichž obálky trajektorií se protínají. Co se týče statických nepohyblivých částí scény, ta bývá rozdělena nějakou hierarchickou strukturou (7.1.1), jsou proto velmi rychle nalezeny ty části, do kterých zkoumaná trajektorová obálka zasahuje.

Cílem této fáze je určit, zda ke kolizi mezi konkrétními dvěma objekty došlo, a pokud ano, jak daleko od původní polohy se tak stalo. Tyto informace jsou předány do druhé fáze, kdy je případná kolize vyřešena.

10.2 Řešení kolize

Ke kolizi je přistupováno podle typu kolidovaných objektů. Některé kolize lze zcela zanedbat, jiné zapříčiní odsunutí objektů do nekolizní polohy, tedy po trajektorii zpět do vzdálenosti, kde ke kolizi došlo. Právě druhý případ bývá složitý, může totiž dojít k řetězové srážce objektů a odsunutím jednoho zpět do správné polohy dojde k dalším srážkám. Někdy je tato chyba ignorována a řešena až v dalším snímku, kdy je kolize opět detekována, jindy je detekce prováděna rekurzivně, dokud není řetězová kolize vyřešena. Tyto problémy obecně řeší fyzikální enginy, které nalezneme ve složitějších hrách.

Nakonec bývá ještě spolu se všemi dostupnými informacemi předáno řízení samotným objektům, aby mohli na kolizi zareagovat ještě jinak, než odsunutím. Zde dochází k přehrání zvuku, aktualizaci herních statistik nebo eliminování objektu.

11.Hra

Nad herním enginem se staví samotná logika hry. Ta využívá stávající funkce, které jsou ve většině her používány velmi podobně.

11.1 Pravidla hry

Součástí herní logiky bývá souhrn pravidel, jejich kontrola a rozhraní nabízející informace o pravidlech a správu statistik. Také je zde stanoven konec hry. Modifikací tohoto by mělo jít měnit chování hry.

Herní režimy

Pro potřeby odlišení chování hry k uživateli, jsou zavedeny herní režimy. Je tak možno to samé hrát jiným způsobem, s jinými možnostmi. Většinou se nabízí možnost hraní ve více hráčích, různé turnaje na postup nebo volná hra.

Herní módy

Pravidla ovlivňují až herní módy, kde je stanoveno, co se smí, co se ve hře objeví a co je samotným cílem.

Pro představu, v této práci tvoří herní módy různé konfigurace zapnutí a vypnutí několika možností. Existence vlajek, možnost po zásahu hrát znovu, hraní v týmech či samostatně a volbou, kdy hra končí.

11.2 Umělá inteligence

Pro možnost hraní v jednom člověku musí být přítomna nějaká umělá inteligence. Ostatní hráči ve scéně jsou tedy kontrolováni hrou (*umělou inteligencí*) a měli by být schopni oponovat živému hráči. Tak jako uživatel, má umělá inteligence (*AI, artificial intelligence*) v této hře stejné možnosti, musí se tedy rozhodovat kam jít a kam střílet.

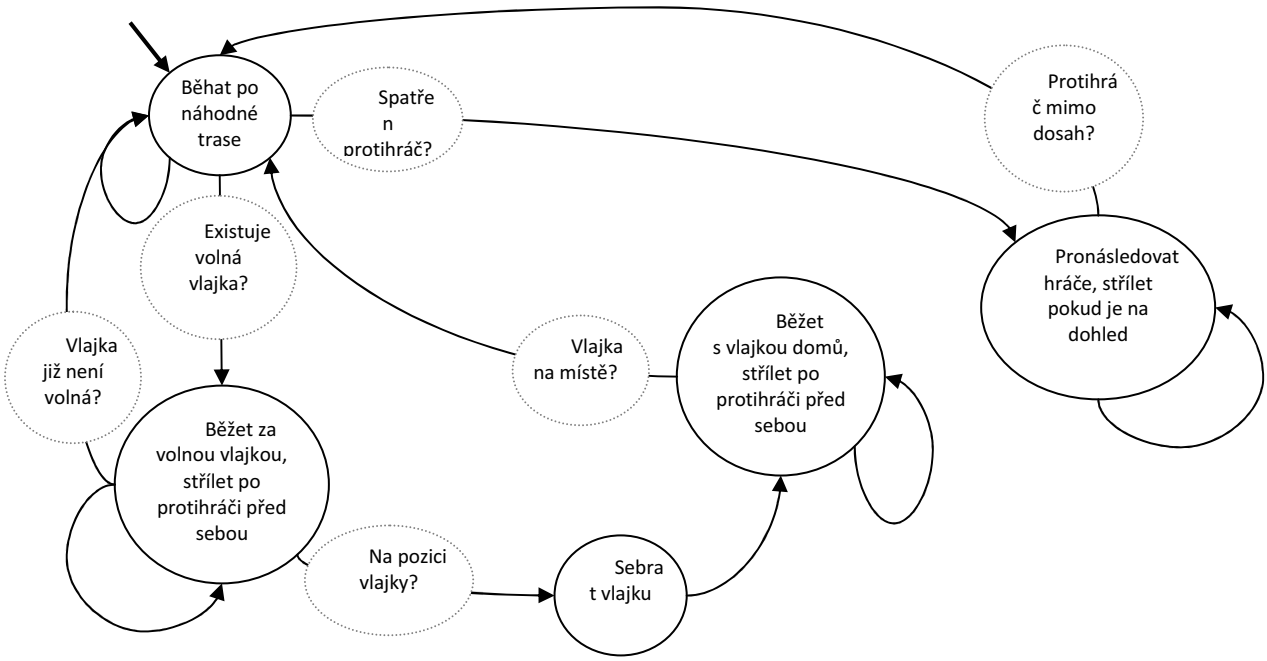
Pohyb ve scéně

Hráč kontrolovaný umělou inteligencí se musí umět pohybovat po scéně. Na místo implementace počítačového vidění se často používá předdefinované cesty (). Je tak možné se jednoduše pohybovat z jednoho místa do jiného. Dokonce není potřeba detekce kolize se stěnami, neboť se předpokládá, že cesty jsou definovány správně.

Pokud je umožněn pohyb scénou po nejkratší cestě, zbývá vyřešit, kam se vlastně vydat, kde se zastavit a kdy střílet.

Rozhodování a plánování

Umělá inteligence ve hře přechází mezi jednotlivými stavy, popisujícími situace, ve kterých se nachází. Pro jednoduché situace, jaké se vyskytují zde, vystačuje neučící se stavová umělá inteligence. Celý problém se tak převede na stavový diagram, kde jednoduché podmínky rozhodují o přechodech mezi stavy (Obrázek 11-1).



Obrázek 11-1: Stavový diagram umělé inteligence

Jednotlivé přechody (kroky) se řeší po jednom v každém snímku (*11.4*). Zbývá určit rozhodovací funkce.

- ‚Protihráč' je každý hráč, v jiném týmu.
- ‚Spatřen' je podmínka, která může být interpretována jako viditelnost v zorném poli. To se řeší neexistencí kolize scény a polopřímky od daného hráče k protihráči, kde polopřímka musí se směrem pohledu hráče svírat úhel nejvýše 45° a protihráč musí být nejvýše v nějaké dané maximální vzdálenosti. Testují se ty části scény, které umělá inteligence nemá prokouknout, tedy stěny a například i křoví.

- ‚Mimo dosah' je další podmínka, pro potřeby hry ji stačí implementovat jako porovnání vzdálenosti protihráče s nějakou danou maximální vzdáleností.
- ‚Volná vlajka' může být vlajka, kterou nedrží žádný hráč.

S tímto jednoduchým popisem stavů, rozhodovacími funkcemi a metodami umožňujícími procházení a hledání cesty ve scéně si inteligence hry vystačí.

11.3 Uživatelské rozhraní

Pozornost je třeba věnovat i uživatelskému rozhraní. Není neobvyklé, že se do počítačových her vyvíjí vlastní systém rozšiřitelného jednotného uživatelského rozhraní. Nejen, aby byl vzhled stejný na každém zařízení a na každém operačním systému, ale také aby se přístup ke hře zjednodušil a uživatel se neztrácel a neměl problém s porozuměním. Jednotnému rozhraní se volí i jednotné ovládání.

11.4 Herní smyčka

Základem her zobrazujících scénu v reálném čase (*real-time*) je zajištění herní smyčky (*snímek*, *heartbeat*) [[6] (main loop)]. Její volání musí být implementováno funkcemi hostujícího operačního systému. Bude ji tedy v tomto případě zajišťovat *framework* (4.1).

Jde o cyklické opakování většinou tří základních rutin.

Nejprve je proveden další krok případné animace uživatelského rozhraní, pak přečtení vstupu od uživatele a možného odeslání stavu scény na jiná zařízení, kvůli podpoře hraní ve více hráčích na více zařízeních (11.5). Nejpodstatnější je pohyb objektů ve scéně, včetně aplikace fyzikálních zákonů a detekcí kolize (10). Během toho bývají kontrolována pravidla a rozhodnuto o případném vítězi. Běh těchto funkcí je závislý na rychlosti herní smyčky. Stačí údaj o délce snímku neboli času (reálné číslo ve vteřinách), který uplynul od začátku předchozího volání herní smyčky. Podle tohoto údaje je stanoveno, jak moc s objekty pohnout a o kolik mají postoupit animace. Délka snímku neboli rychlost smyčky se jinak udává jako *FPS* (*frames per second*, *snímků za vteřinu*), kde $FPS = \frac{1}{\text{délka snímku}}$. Samozřejmě je snahou dosáhnout co nejvyššího FPS, přičemž pro lidské oko optimální se udává *FPS* vyšší než 30, při nižších hodnotách je postřehnutelné trhání.

Běh často nejvíce zpomaluje samotné vykreslení scény. To se provádí v herní smyčce ihned po pohybu objektů, aby byla změněná scéna co nejdříve zobrazena uživateli a ten nebyl znevýhodněn prodlevou.

Nakonec je zvykem ponechat určitý prostor hostujícímu operačnímu systému. Ten zde může reagovat na vnější události, včetně samotného zpracování vstupu uživatele. Umožní se tak okamžitě reagovat na události jako příjem telefonního hovoru a zabrání se možnému hromadění událostí ve frontě s událostmi k zpracování.

11.5 Sdílení scény na více zařízeních

Nemůže chybět možnost hraní s více ‚živými' spoluhráči. Obnáší to sdílet scénu a dění na ní na více zařízeních propojených nějakou sítí.

Propojení a samotnou komunikaci musí implementovat framework (4.1) neboť na každém systému bude komunikace zprostředkovávána jinými API funkcemi a přes jinou síť. I použitý protokol závisí na konkrétní implementaci frameworku, přesto pro své vlastnosti, hlavně pro schopnost přijímat pakety v tom pořadí, v jakém jsou odeslány, je nejpraktičtější protokol TCP/IP. Rozhraní frameworku se tedy rozšíří o navázání spojení (buď jako klient nebo server), ukončení spojení, poslání pole bajtů a příjmu pole bajtů. Zde se vlastně trochu obrátí zažité role klienta a serveru. Klient je brán jako ten co čeká na spojení (naslouchá a čeká na spojení, ale pouze na jedno; přičemž naslouchat bude více klientů) a server se jednorázově pokusí o připojení k naslouchajícím klientům. Rozdíl těchto dvou rolí je také v tom, že klient odesílá data pouze směrem na server, ale server odesílá najednou data všem připojeným klientům, navíc také přeposílá některá data přijatá od klienta ostatním klientům. Aplikace si pak deklaruje své binární struktury s jednotnou hlavičkou, které přetypované na pole bajtů posílá funkcemi implementovanými konkrétním frameworkem. Přijatá data jsou aplikací přetypována zpět na binární strukturu určenou jednotnou hlavičkou. Struktury obsahují binární data, která popisují přenášenou informaci. Například identifikátor hráče, jeho nová souřadnice a jeho aktuální použitá animace reprezentuje informaci o pohybu hráče.

Různé instance programu na různých zařízeních si musí vyměňovat informace, díky kterým budou jejich kopie scény stejné. Používá se architektura klient/server, kde server obstarává propojení s klienty a rozesílání změn ve scéně. Klienti pak jen posílají pohyb

svého jednoho hráče, případně objekty, které jejich hráč vytvořil (například střely). Je tak zajištěno, že pohyb hráčů si každý bude zajišťovat sám, bude se tedy u nich projevovat okamžitě, a zároveň budou všichni vědět o všech. Konkrétně úloha klientů spočívá pouze v odeslání pohybu svého jednoho hráče a bezmyšlenkovité úpravy stavů a polohy ostatních objektů podle získaných informací od serveru. Server má během hry za úkol kontrolovat případného výherce, určovat kdo je zasažen, kterou vlajku kdo sebral a upustil, získávat informace o pohybu hráčů klientů a klientům vše rozesílat. [[8], [5] (client/server, bluetooth, networking)]

Jako první se musí provést inicializace. Server tedy připojí klienty, odešle jim informace o aktuální scéně (postačuje název souboru scény, její jedinečný identifikátor a použitý herní mód), pak si klienti vytvoří svou kopii scény a odešlou nazpět potvrzovací zprávu. Pak server určí jednoznačnou identifikaci hráčů, a který klient bude ovládat kterého hráče. Nakonec opět čeká na potvrzení od klientů, aby mohla být hra spuštěna všude současně.

Nežádoucí je také zpoždění, způsobené přenosem informací na server, zpracováním a následným rozesláním klientům. To ale bývá velmi malé, pokud je komunikace zprostředkována po lokální síti (s nízkým zpožděním, LAN nebo PAN). Na konci snímku klienta jsou odeslány informace na server (poloha hráče, nové vystřelené střely), na začátku snímku serveru jsou přečteny informace od klientů a podle nich upravena lokální kopie scény. Na konci snímku serveru jsou rozeslány nové zpracované informace zpět klientům (polohy všech hráčů, nové vystřelené střely všech hráčů), zpoždění je tedy rovno

délce dvou snímků, což bývá kolem sekundy, plus zpoždění sítě.

Samozřejmě každá instance programu má v různé chvíle různé délky snímků. Ve scéně je vždy alespoň jeden hráč ovládaný někým jiným (*vzdálený hráč, remote player*) a případné nepravidelnosti v obdržování jeho nové polohy způsobí trhání jeho pohybu. Proto bývá součástí každé obdržené polohy vzdáleného hráče i aktuální směr a rychlost. Pří vynechání komunikace je tak v následujícím snímku dopočítána předpokládaná nová poloha a trhání se tak eliminuje. Tento pohyb lze jednoduše popsat konkrétní animací (9.2), která se na model hráče aplikuje. Dalšími objekty, jejichž poloha se automaticky dopočítává bez potřeby obdržovat pravidelně informace jsou střely. Ty stačí jednou

inicializovat směrem a rychlostí a následný ničím neovlivňovaný pohyb je jednoduché všude v každém snímku dopočítat stejně, až dokud nedorazí informace o zničení střely, kterou rozešle server po detekci kolize střely se scénou či hráčem nebo dokud střele nevyprší *life-time* (maximální délka existence, pro případ že se střela s ničím nesrazí nebo server nečekaně ukončí připojení).

Tento klasický obecně používaný systém komunikace také zajišťuje potřebnou synchronizaci mezi programy na různých zařízeních. Také tím, že o všem rozhoduje jeden server, se vyhneme konfliktům. Pokud by dva klienti najednou vyžadovali přístup ke stejnému objektu, například chtějí sebrat vlajku, učiní tak zasláním žádosti, server vyhoví jednomu žadateli a při následujícím odesílaní informací klientům je připojena i událost o sebrání vlajky daným hráčem. Klienti na událost reagují tak, že své kopii daného hráče vlajku vnutí. O všech problémech tedy rozhoduje server a klienti pouze zasílají požadavky a reagují na získané události od něj úpravou své kopie scény.

12.Závěr

V bakalářské práci jsem popsal a implementoval techniky používané při tvorbě počítačových her zobrazujících prostorovou scénu v reálném čase. Použité techniky jsou vhodné a uzpůsobené pro využití na mobilních zařízeních, která se oproti stolním počítačům vyznačují hlavně znatelně pomalejšími procesory, řádově menší operační pamětí, menším displejem a omezenými možnostmi ovládání.

Je vidět, že v některých oblastech se takový program musí velmi lišit od alternativ na stolní počítače, hlavně z hlediska návrhu ovládání a návrhu scény, potřeby zajištění přenositelnosti a nutnosti vytvoření vlastních vykreslovacích rutin, zatímco podoba herní smyčky, hraní ve více hráčích či ukládání dat se realizuje stejně.

Některé zde implementované součásti bývají stejně řešené i ve složitějších systémech. Například definice a zpracovávání prostorových modelů včetně vnitřní struktury kostí a popisu animací se ve stejné podobě používá i v moderních počítačových hrách. Také všechny zde zmíněné optimalizace řešení viditelnosti jsou základem nejen pro hry, ale také pro další software zobrazující prostorovou scénu. Mezi ty patří různé modelářské programy, nástroje pro architekty a programy zobrazující virtuální realitu.

Za povšimnutí stojí, že podstatná část programu je dělána velmi obecně a s předpokladem dalšího využití pro nové programy. Není tedy problém celý herní engine oddělit a použít jako základ pro jiné aplikace. Nemusí být použito vše, animované prostorové objekty jsou dobrým příkladem, čím je možno obohatit budoucí programy o zajímavé možnosti. Logika hry (11) je od enginu (4, 7, 8, 9, 10) odlišena na úrovni zdrojového kódu (a souborového systému). Postačí oddělit příslušné zdrojové soubory definující požadované objekty a objekty jimi používané. Častějším řešením používaným v komerčních systémech je jednou oddělit tyto objekty, zkompilovat je do samostatné dynamicky nebo staticky linkované knihovny, kde se veřejné třídy a metody exportují.

13.Reference

[1] Web 3DICA (http://tfpsly.free.fr/Docs/3dIca/3dica.htm)

[2] Moderní počítačová grafika (Jiří Žára, Bedřich Beneš, Petr Felkel)

[3] Grafika I, II (http://cgg.ms.mff.cuni.cz/~pepca/lectures/, Josef Pelikán)

[4] Matematické principy grafických systémů (Dalibor Martišek)

[5] Symbian SDK (http://developer.symbian.com/main/tools/sdks/)

[6] DirectX 7,8,9 SDK (http://msdn.microsoft.com/directx/sdk/)

[7] Programování ve Win32 API (Ch.Petzold)

[8] Síťové programování pod Windows a programování Internetu (Josef Pirkl)

14.Dodatky

14.1 Aplikace pro tvorbu dat

- **EditorModel**

Prostorové modely jsou vytvořeny, přesněji importovány a pak upraveny, v externím programu EditorModel. Zde je možné použít (importovat) existující model ve formátu X (Direct 3D X file), obohatit ho o dodatečné informace používané ve hře a uložit ve formátu, který používá herní engine. Dodatečnými informacemi jsou hlavně struktura kostí, animace, jméno a hmotnost.

- **MapEditor**

Pro tvorbu binárních souborů popisujících herní scénu se používá aplikace MapEditor. Zde lze vytvářet, upravovat a ukládat herní scénu s pomocí nástrojů stěna, podlaha, strop, cesta, pozice vlajky a pozice hráčů. Nastavují se i parametry název scény, soubor s texturou oblohy a soubor s texturou náhledu scény.

14.2 Formát souborů

- **Jazykový soubor**

Nezbytné je zajistit vícejazyčné prostředí programu. Proto program interně všechny texty identifikuje pomocí klíče, který se přeloží na řetězec napsaný v konkrétním textovém jazykovém souboru. V tom je na každém řádku dvojice klíč a řetězec, oddělené bílými znaky, kde klíč je řetězec bez bílých znaků a řetězec je text v uvozovkách. Podřetězce \“, \\, \0, \t, \n jsou reprezentovány jako uvozovky, zpětné lomítko, znak s ordinální hodnotou 0, tabulátor a odřádkování. Změnou jazykového souboru se změní i všechny zobrazené texty v programu.

14.2.1.1 Použití v programu

Program udržuje použité dvojice [klíč,text] ve vyhledávací struktuře. Pokud požadovaný klíč ve struktuře ještě není, pokusí se jej načíst z jazykového souboru. Pokud není ani v souboru, je použit jako text přímo identifikátor klíče. Při změně jazykového souboru je všem klíčům ve vyhledávací struktuře aktualizován text.

Vyhledávací struktura nejprve rozřadí hodnotu podle hash hodnoty klíče, dále hodnoty třídí v binárním vyhledávacím stromě. Pro zjednodušení jsou rozlišována malá a velká písmena.

- **Konfigurační soubory**

Konfigurační soubory (5.2) jsou textové soubory určující hodnoty různých atributů. Každý atribut začíná symbolem ‚#' na nové řádce, za ním identifikátor, oddělovač (mezery nebo tabulátory) a následuje hodnota v uvozovkách. V případě, že by hodnota měla obsahovat samotné uvozovky, musí být před ně předsazen znak ‚\'. Podřetězec ‚\0' je čten jako znak s ordinální hodnotou 0. Zdvojení znaku ‚\' je čteno jako jedno ‚\'.

14.2.2.1 Nastavení po spuštění

Různé parametry hry, které jsou použity po spuštění, jsou ukládány do konfiguračního souboru „startup.txt". Definuje atribut *maps*, kde jsou vypsány dostupné soubory se scénami oddělené znakem s ordinální hodnotou 0. Dále atributy *aim*, *aim_target*, *splash*, *ball*, *background*, *gun*, *flag*, *sndFire*, *sndHit*, *sndSplash* určují názvy souborů s texturou pro zaměřovač, zvýrazněný zaměřovač, skvrnu, kuličku, pozadí menu, 3D model popisující zbraň, model vlajky a zvukové soubory pro výstřel, zásah hráče a zásah stěny.

14.2.2.2 Popis hráče

Konfigurační soubory spjaté s konkrétní pozicí v konkrétní scéně popisují hráče, který se na tuto pozici může umístit. Obsahuje pouze jediný atribut *MODEL_FILE* určující 3D model, který se pro hráče použije.

14.2.2.3 Herní módy

Obecně popsané dostupné herní módy jsou uložené v konfiguračním souboru „gamemodes.txt". Obsahuje atribut *modes_count*, udávající počet dále popsaných herních módů. Následují atributy name_modeX, respawning_modeX, useflags_modeX, countfrags_modeX, thumbnail_modeX, teamplay_modeX, kde X je číslo módu od 1 do *modes_count*. Určují název módu X jako klíč do jazykového souboru, *true*|*false* pro povolení|zakázání opětovného hraní po zásahu, používání vlajek a počítání zásahů do skóre. Atribut *thumbnail_modeX* říká jaký použít soubor s obrázkem pro náhled módu před spuštěním hry a teamplay_modeX obsahuje jednu z hodnot [true|false|alone] pro

specifikaci zda se bude hrát v týmu, zda bude každý hrát za sebe nebo zda budou všichni hrát proti jednomu hráči.

- **Animované textury**

Jednoduché rozhýbání textur lze přidat do hry použitím souboru popisujícím animaci textury. Jde o textový soubor, kde je na prvním řádku rychlost animace v počtu snímků za vteřinu a na následujících řádcích jsou jednotlivé názvy obrázkových souborů coby jednotlivých snímků.

- **Prostorový model**

Prostorové modely jsou uloženy v binárním souboru, obsahujícím fragmenty identifikované hlavičkou s velikostí fragmentu (16b číslo bez znaménka) a identifikátorem fragmentu (16b číslo bez znaménka) (5.5). Každý fragment je určen binární strukturou. Jednotlivé fragmenty jsou deklarovány v hlavičkovém souboru *„/GameEngine/DataTypes.h“*.

- **Scéna**

Popis scény je uložen stejným způsobem jako soubor prostorového modelu (14.2.4). Deklaraci je možno naleznout v souboru *„/DataTypes2.h*“.