VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

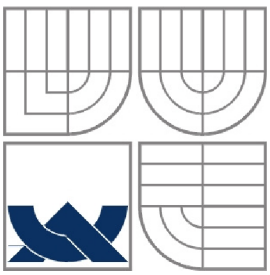
VÝVOJ INDIE GAME

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

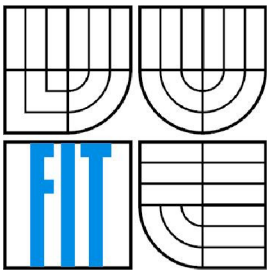
AUTOR PRÁCE
AUTHOR

BC. MICHAL ZACHARIÁŠ

BRNO 2011



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

VÝVOJ INDIE GAME

INDIE GAME DEVELOPMENT

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

BC. MICHAL ZACHARIÁŠ

VEDOUCÍ PRÁCE
SUPERVISOR

ING. RUDOLF KAJAN

BRNO 2011

ZADÁNÍ DIPLOMOVÉ PRÁCE

Abstrakt

Tato diplomová práce se zabývá vývojem *indie game* - tedy nezávislé vyvinuté hry. Popisuje důležité momenty v historii počítačových her. Objasňuje pojmy jako zlatý věk videoher a krach herního průmyslu roku 1983. Dále vysvětluje historii a vznik fenoménu *indie game*. Stručně popisuje některé rozdíly při vývoji nezávislé a komerční hry. V další části uvádí hlavní rysy některých z herních enginů a nástrojů, které lze použít pro tvorbu *indie games*. Nakonec popisuje návrh a implementaci jednoduchého herního enginu a hry na něm postavené.

Abstract

This master's thesis deals with development of *indie game* - independently-developed game. It describes important moments in computer games history. It clarifies terms like golden age of video arcade games and video game crash of 1983. Further it explains history and origin of *indie game* phenomenon. It describes some of the differences between independent and commercial game development. In next chapter it presents some game engines which are suitable for independent game development. And in the last chapter it describes the design and implementation of game engine and game running on it.

Klíčová slova

historie videoher, indie game, vývoj počítačových her, zlatá věk videoher, krach herního průmyslu v roce 1983, unreal engine, torque engine, xna, real-time strategy

Keywords

video games history, indie game, computer games development, golden age of video arcade games, video game crash of 1983, unreal engine, torque engine, xna, real-time strategy

Citace

Zachariáš Michal: Vývoj indie game, diplomová práce, Brno, FIT VUT v Brně, 2011

Vývoj indie game

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Rudolfa Kajana. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Michal Zachariáš
23. května 2011

Poděkování

Děkuji svému vedoucímu Ing. Rudolfu Kajanovi za poskytnutý čas a odbornou pomoc při práci na mém projektu. Také děkuji mým týmovým kolegům Davidu Jozefovovi a Martinu Wilczákovi za výbornou spolupráci na hře *FireFighters: Whatever it takes*.

Obsah

1 Úvod

Tématem této diplomové práce je vývoj *indie game* - tedy vývoj nezávislé hry. Zmíněná nezávislost se především vztahuje na finanční podporu některé velké herní společnosti nebo distributora. Hry jsou většinou financovány z úspor samotných vývojářů, a tudíž nemají velký rozpočet. Fenomén nezávislé hry se objevil poměrně nedávno a právě teď zažívá rapidní rozvoj.

Následující kapitola *Historie videoher* obsahuje stručný přehled historie počítačových her jako celku a zahrnuje období od 50. let 20. století až po současnost. Čtenář se dozví o prvních počítačových hrách, které byly více laboratorními pokusy než snahou o vytvoření plnohodnotné hry, avšak inspirovaly nemalé množství dalších vývojářů. Dále ho seznámí s pojmy zlatý věk videoher a krach herního průmyslu v roce 1983. Přiblíží rozmach herních konzolí a osobních počítačů v moderním věku a nakonec se zmíní o prvních 3D grafických kartách, násilí ve hrách a vzniku společností jako ESRB, které se specializují na hodnocení vhodnosti her.

Ve třetí kapitole *Indie game* si řekneme, co tento pojem vůbec znamená, jak vznikl a jaká je jeho historie. Zmíníme první nezávislé herní projekty, které však ještě nenesly označení *indie game*, osvětlíme, co je *demoware* a *shareware*, období izolovaných herních enginů a pokusíme se určit, kdy vznikl samotný pojem *indie game*. Také se dozvíme, jaký je nejčastější způsob distribuce těchto her.

Čtvrtá kapitola *Vývoj nezávislé a komerční hry* stručně popisuje rozdíly mezi vývojem nezávislého a komerčního herního projektu a čeho by se měl vývojář *indie game* vyvarovat.

Pátá kapitola *Nástroje pro tvorbu indie game* obsahuje přehled nejrozšířenějších grafických aplikačních rozhraní - DirectX a OpenGL. Také obsahuje popis dvou herních enginů, se kterými je možné vytvářet *indie games* - Torque Game Engine Advanced a Unreal Engine. Nakonec ukáže, jak pracuje XNA Game Studio společnosti Microsoft.

Šestá kapitola *Tvorba enginu* podrobně popisuje návrh a implementaci enginu postaveného na frameworku XNA Game Studio 4.0. Detailně rozebere jednotlivé komponenty, které lze využit především pro tvorbu her v žánru *Real-time strategy*.

Poslední kapitola *Tvorba hry* vysvětluje návrh a implementaci částí důležitých pro vývoj hry *FireFighters: Whatever it takes*, která byla přihlášena do soutěže *Imagine Cup*, pořádanou společností Microsoft a to konkrétně do kategorie game design.

2 Historie videoher

V této kapitole se nachází stručný přehled historie videoher. Popisuje pouze hry, vývojáře, konzole a události, které byly důležité pro vývoj herního průmyslu do stavu, jaký známe teď. Především se zabývá **průkopnickými léty** herního průmyslu, kdy byly hry spíše jen pokusy, které měly otestovat výkon sálových počítačů až na hranice možností. Dále pak tzv. **zlatým věkem videoher**, kdy herní automaty dosáhly vrcholu a staly se velmi populární mezi americkou veřejností. **Moderním věkem videoher**, ve kterém se herní konzole a osobní počítače dostávají do domácností a nakonec popisuje trendy, které se objevují v **současných videohrách**.

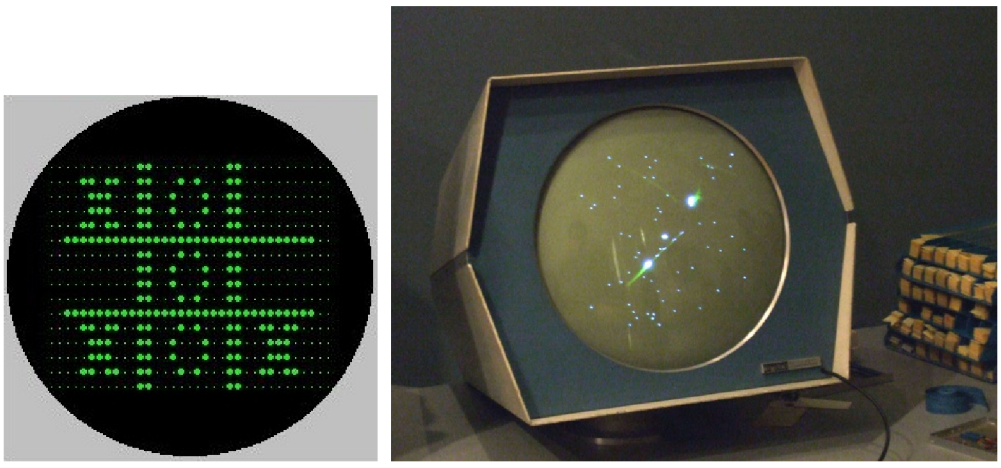
2.1 Průkopnická léta

Historie počítačových her a videoher zahrnuje již více než padesát let. Do povědomí širšího okruhu lidí se však dostaly až v polovině sedmdesátých let a v tehdejším Československu se počítačové hry objevily až po revoluci, tedy po roce 1989.

V úplných počátcích běžela většina počítačových her na sálových počítačích amerických univerzit a byla vyvíjena jednotlivci jen jako koníček. Přístup k těmto sálovým počítačům byl velmi omezen, a tudíž takovýchto her nevzniklo mnoho a byly povětšinou zapomenuty [1].

V roce 1947 byla již televize v Americe poměrně rozšířená a tak vznikl nápad umožnit na ní lidem hrát hry. Thomas Goldsmith a Estle Mann, zaměstnanci televizní společnosti Dumont, vymysleli zařízení, které nazvali **Cathode-Ray Tube Amusement Device** a které mělo lidem umožnit virtuálně odpalovat rakety na cíl a řídit jejich trajektorii. V témže roce tento nápad také patentovali, avšak Dumont jej nikdy nezačal vyvíjet, natož prodávat [2].

Roku 1952 byla dokončena první grafická počítačová hra **OXO** (také známá jako **Noughts and Crosses**), což byla v podstatě známá hra *Tic-Tac-Toe*. Byla ovládána telefonním rotačním číselníkem a výstup hry se zobrazoval na CRT s rozlišením 35×16 pixelů (viz obrázek 2.1).



Obrázek 2.1: Vlevo grafický výstup hry OXO, vpravo vektorový zobrazovací systém s hrou Spacewar!

Hra **Spacewar!** je považována za první „střílečku“ a také jako první využívá vektorový zobrazovací systém (viz obrázek 2.1). Byla vytvořena studenty univerzity MIT Martinem Greatzem, Stevem Russellem a Waynem Wiitenem, kteří ji vytvořili na mini-počítači DEC PDP-1. Tato hra inspirovala mnoho dalších vývojářů, jako byli Bill Pitts a Hugh Tuck.

Ti roku 1971 na strandfordské univerzitě sestrojili první automatovou hru na mince – **Galaxy Game**. K masové produkci však nikdy nedošlo, neboť hardware stál přibližně 20 tisíc dolarů. V strandfordském kampusu však byla velmi oblíbená a zůstala zde až do roku 1979, kdy byla odstraněna, kvůli poškozeným displejům.

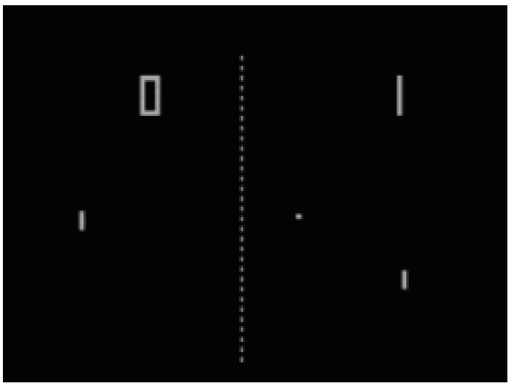
Inspirováni Galaxy Game naprogramovali Nolan Bushnell a Ted Dabney automatovou hru **Computer Space** (viz obrázek 2.2), která je považována za první komerčně prodávanou hru vůbec. Zajímavostí je, že nepoužívá žádný mikroprocesor, ROM ani RAM. Celý systém je založen na principu konečného automatu a je sestaven ze 74 TTL čipů.

Obrázek 2.2: Automat hry Computer Space

2.2 Zlatý věk videoher

Anglicky označovaný jako *Golden age of video arcade games*. Různé zdroje se liší v názoru, kdy přesně zlatý věk započal. Webový portál *The History of Computing Project* jeho počátek datuje od roku 1971, kdy byl vytvořen první herní automat [4]. Avšak Steven L. Kent, ve své knize *The Ultimate History of Video Games*, umístil toto období do let 1979 až 1983 [3].

Rok po vydání Computer Space, tedy v roce 1972, založili Bushnell a Dabney firmu **Atari**, jejíž první hrou byl **Pong** (viz obrázek 2.3). Díky této hře se za méně než šest měsíců stala z neznámé firmičky vedoucí firma herního průmyslu. Pong se brzy stal velmi populární a jen v Americe se prodalo přes 500 tisíc kusů. Pong také inspiroval řadu tvůrců a to nejen v oblasti počítačových her. Inspirací byl prý i pro režiséra slavného sci-fi filmu **Tron** Stevena Lisbergera. Počítačové triky vznikaly na počítačích s 2 megabajty paměti a postprodukce zaměstnala přes 500 animátorů a grafiků – přesto snímek nedostal nominaci na Oscara za speciální efekty, protože akademie považovala použití počítačů za podvádění [13].



Obrázek 2.3: Obrázek ze hry Pong

V tomto období dochází především k rapidnímu vývoji hardwaru specializovaného na herní automaty, snížení výrobních nákladů a tedy i k masovému rozšíření herních automatů mimo obvyklé herny a to do nákupních středisek, restaurací, čerpacích stanic atd. Také vznikly první cenově dostupné herní konzole, které mohli lidé připojit ke svým televizním stanicím a zahrát si v pohodlí svého domova. Poprvé tak lidé mohli ovlivnit to, co se na televizní obrazovce děje a ne jen sedět a přepínat kanály. První takovou hrou byl **mini-Pong** (viz obrázek 2.4), což byla domácí verze výše zmiňované hry Pong. Na Vánoce roku 1976 se těchto konzolí prodalo v Americe 13 miliónů kusů.

Obrázek 2.4: Konzole společnosti Atari mini-Pong

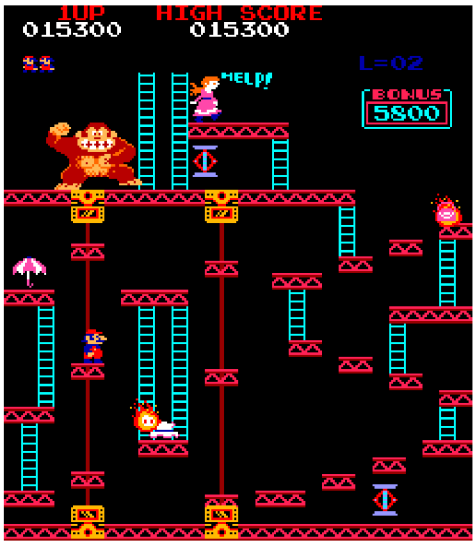
Na konci 70tých let přicházejí mikroprocesory, které zjednodušily a zpřístupnily vývoj herních konzolí širšímu okruhu vývojářů. Vzniklo velké množství různých herních konzolí napodobujících mini-Pong, ale také řada originálních.

Jelikož byl hardware té doby stále velmi omezený, museli se vývojáři zaměřovat především na výbornou hratelnost a znovuhratelnost. Díky tomuto faktu vznikaly hry, jejichž odkaz doteď nacházíme v mnoha moderních hrách. Existuje nespočet remaků herních klasik jako **Pong**, **Space Invaders**, **SpaceWars!**, **Asteroids** nebo **Lunar Lander**.

2.3 Moderní věk

V roce 1983, kvůli nestálému hernímu trhu a klesajícímu zájmu o počítačové hry vůbec, nastává tzv. krach herního průmyslu, anglicky *Video Game Crash*. Částečně byl také způsoben sérií špatných her a tedy snížením zisků distributorů, kteří postupně o hry ztráceli zájem. Dalším důvodem byl nárůst popularity osobních počítačů (PC), protože mnoho rodičů věřilo, že lepší investicí je koupě zařízení, jež má širší využití, než jen hraní her.

Nicméně tyto skutečnosti a obavy nezasáhly Japonsko a herní průmysl zde dále vzkvétal. To je také důvod, proč je japonská společnost Nintendo označována za otce moderního věku videoher. Prvním obrovským úspěchem společnosti bylo vydání jejich 8bitové herní konzole **Famicom** (později známý jako **Nintendo Entertainment System** nebo zkráceně **NES**). Velký podíl na úspěchu měla hra, která se s touto konzolí prodávala - **Donkey Kong** (viz obrázek 2.5). [4]



Obrázek 2.5: První hra herní konzole Famicom - Donkey Kong

Za zmínku stojí hra **I, Robot** (viz obrázek 2.6) vyvinutá v roce 1983 společnosti Atari, konkrétně Davidem Theurerem, který se podílel i na dřívějších herních projektech, jako **Tempest** a **Missile Command**. Je to vůbec první 3D polygonová hra, která absolutně předběhla svou dobu, neboť další 3D polygonové hry začaly vznikat až okolo roku 1991.

Obrázek 2.6: Revoluční hra I, Robot

V moderním věku videoher také vznikají první 16, 32 a dokonce 64bitové platformy a konzole. Jednou z nich je například **Super Famicom** (v Americe a Evropě známý jako **Super Nintendo Entertainment System** nebo také zkráceně **SNES**), další je dodnes známý **PlayStation** od společnosti Sony. To umožnilo vytvářet složitější a výpočetně náročnější hry, nemluvě o přechodu z 8bitové barevné hloubky na vyšší.

Toto období trvá přibližně do roku 1995 a vzniklo v něm velké množství herních legend, mezi které patří například **Dungeon Master**, **The Sims**, **SimCity**, **Myst**, **Donkey Kong Country** a mnoho dalších.

2.4 Současné videohry

Období současných videoher se datuje od roku 1995 a vznikají v něm stále nové 32, 64 a 128bitové systémy. Osobní počítače se masivně rozšiřují z kanceláří a firem i do domácností. Dále narůstá popularita online hraní, online sázení (např. kasína, sportovní událost atd.) a sociálních sítí. Rapidně se zkracují období mezi objevováním nových technologií a s tím souvisejícím vydáváním nových her, které tyto technologie využívaly.

Obrázek 2.7: První grafická karta s 3D akcelerací Voodoo

Objevují se první grafické karty s 3D akcelerací, jejichž asi nejznámější zástupce je **Voodoo** od společnosti **3DFX** (viz obrázek 2.7), které však musely být doplněny o VGA kartu, neboť nepodporovaly 2D výstup. Později na trh přichází kanadská **ATI** a kalifornská **Nvidia**, která koupila všechen majetek firmy 3DFX a získala tak množství výborných zaměstnanců a jejich *know-how*.

Tento pokrok umožnil tvůrcům her navrhovat hry s daleko reálnějším prostředím, modely a efekty. Poprvé v historii bylo možné víceméně reálně zobrazit násilí, hororové scény a sexuální obsah (viz obrázek 2.7), což v důsledku způsobilo nárůst počtu her, jejichž cílové obecenstvo byli dospělí lidé – např. **Resident Evil**, **Tomb Raider**, **Final Fantasy VII**, **Metal Gear Solid**, **Soldier of Fortune** a **Silent Hill**.

Proti těmto hrám se samozřejmě zvedla vlna odporu a nevole ze strany rodičů, puritánů a pacifistů. V USA se dokonce uvažovalo o zavedení zákona, který by úplně zakazoval násilí v počítačových hrách. V této době však vzniká první společnost hodnotící hry **ESRB** (**Entertainment Software Rating Board**) a zákon tedy nebylo nutné dále prosazovat. Evropa opožděně (až v roce 2002) následovala příkladu a vzniká společnost **PEGI** (**Pan-European Game Information**), jejíž

hodnotící systém je dnes v Evropě nejrozšířenější. Tato hodnocení jsou výhodná jak pro rodiče, kteří se mohou na jejich základě rozhodnout, zda hru svému dítěti koupí, tak pro vývojáře a vydavatele, kteří tak zajisté ušetří spoustu peněz za případné soudní spory s rozzlobenými rodiči [6].

Rozvoj internetu, sociálních sítí, mobilních telefonů, online hraní, narůstající softwarová i hardwarová podpora pro herní vývojáře a především nechuť některých vývojářů pracovat na hře jen kvůli zisku dala vzniknout zcela novému odvětví herního průmyslu – *indie game*.

3 Indie game

Tato kapitola popisuje fenomén nezávislých her – *indie game*. Podrobněji vysvětluje **samotný pojem**, dále pak **vznik a historii** a nakonec **současné trendy** tohoto fenoménu.

3.1 Samotný pojem

Independently-developed game (zkráceně **Indie game**) je taková hra, která vzniká bez finanční podpory některé z velkých herních společností nebo distributorů. Hra většinou nemá velký rozpočet, neboť vývojáři často čerpají peníze z vlastních úspor. Tým se zpravidla skládá jen z malého počtu lidí, někdy na hře pracuje dokonce jen jednotlivec. Vývoj od základu do konce může trvat několik let, ale není výjimkou, že je celá hra vytvořena v řádu dní nebo dokonce hodin – záleží na složitosti projektu, zkušenostech a dovednostech vývojového týmu a mnoha dalších faktorech. Tyto hry si nezakládají na *high-end* grafickém zpracování a efektech, ale především na originálnosti a hratelnosti [8].

3.2 Vznik a historie

3.2.1 První nezávislé herní projekty

Za první nezávislé herní projekty lze označit hry, které vznikaly již koncem 70. let, kdy žádný ucelený herní průmysl neexistoval, a všechny hry byly vytvářený malým týmem nadšenců, kteří nehleděli na zisk, ale chtěli dále rozvíjet herní průmysl a tvořit hry podle svých představ. Mezi takové hry lze zařadit například **SpaceWars!** (viz kapitola 2.2). Komerční společnosti, které později začaly vznikat, tyto programátory vyhledávaly a zaměstnávaly, neboť měli s tvorbou her nemalé zkušenosti [8][9].

3.2.2 Shareware a demoware

Na konci 80. a začátku 90. let je prodejní model většiny indie her založen na *shareware* nebo *demoware*. To znamená, že část hry je zadarmo, ale za zbytek musí hráč zaplatit. Hry, které tento model používaly, byl slavný **Doom** a **Commander Keen** (viz obrázek 3.1). Tento systém hráče motivoval ke koupi i zbytku hry, protože velmi často byla zdarma jen první epizoda a další již byly placené.

Obrázek 3.1: Screenshoty z her Commander Keen a Doom

3.2.3 Vlna izolovaných herních enginů

V období od roku 1985 do roku 2004 vznikalo velké množství nezávislých herních enginů, kdy většina z nich byla vytvořena pro velmi populární 8bitový domácí počítač Commodore 64 (zkráceně C64). První hry byly většinou ztraceny, protože neexistovala žádná možnost komunikace mezi jednotlivými vývojáři a žádný způsob jejich distribuce. Takové hry si mohli zahrát jen samotní vývojáři a lidé okolo nich. Nicméně i dnes lze na Internetu najít sbírky těchto her a je možné si je zahrát na některém z emulátorů.

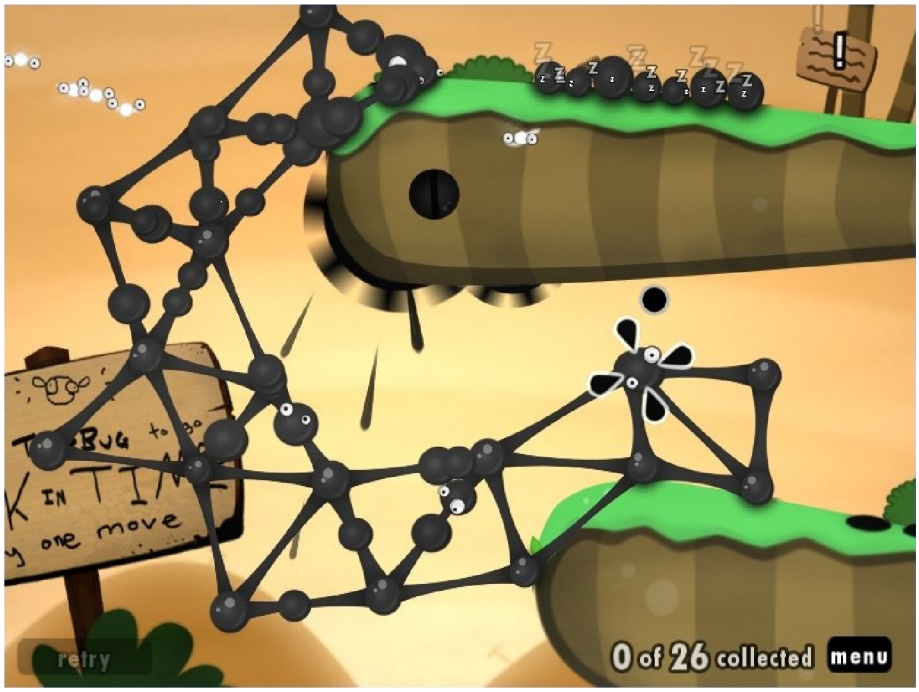
Jeden z prvních populárních nástrojů, které umožňovaly vytvářet počítačové hry bez rozsáhlých znalostí programování, byl **ZZT** vytvořený Timem Sweenym (zakladatel společnosti Epic Megagames). Pomocí tohoto nástroje byly vytvořeny tisíce her a s příchodem Internetu bylo snadnější je mezi hráče distribuovat. Dalšími známými i méně známými herními enginy byly například **RPGMaker**, **Inform**, **ClickTeam**, **MegaZeux** nebo **Game Maker**. Poslední zmíněný je dodnes jeden z nejoblíbenějších nástrojů pro tvorbu her a používán množstvím indie vývojářů. Je to především pro jeho jednoduchý *drag&drop* systém a jednoduchý proprietární programovací jazyk – *Game Maker Language* (GML) [10].

3.2.4 Vznik pojmu indie

Komunita okolo indie her byla příliš roztříštěná. Skládala se z velkého množství subkomunit, které se vždy utvářely okolo určitého enginu (např. subkomunita okolo Game Makeru, subkomunita okolo ClickTeamu atd.). Nedocházelo k žádné výměně nápadů, postřehů a zkušeností. Hry se nedostávaly mimo svou subkomunitu a byly hrány jen jejími příznivci. To se změnilo v roce 2006 sérií událostí. Některá nezávislá herní studia se začala označovat jako *indie developers* – jedno z nejznámějších je studio **Introversion**, mezi jejichž nejznámější počiny patří **Uplink** (viz obrázek 3.2), ve které se hráč stal hackerem a plnil různé úkoly pro mezinárodní organizace. Tyto úkoly zahrnovaly hackování počítačových systémů konkurenčních společností, krádeže údajů z výzkumu, sabotáže ostatních společností, praní špinavých peněz, mazání důkazů nebo podstrčení falešných důkazů [13]. Mezi další velmi úspěšné hry společnosti patří **Multiwinia**, **DEFCON** a **Darwinia** (viz obrázek 3.2) [12].

Obrázek 3.2: Screenshoty z her Uplink a Darwinia

V roce 2006 se od velké komerční společnosti **Electronic Arts** (zkáceně EA) odtrhli dva vývojáři, **Ron Carmel** a **Kyle Gabler**, kterým se nelíbil tlak společnosti, jaký na své zaměstnance vyvíjela, a její hon za ziskem na úkor zábavnosti a hratelnosti her. Tito dva muži založili nezávislé dvoučlenné herní studio, které nazvali **2D Boy**. Aby minimalizovali náklady na vývoj, používali jako kanceláře San Franciské kavárny s bezplatným připojením k Internetu. Kdyby se někdo o něco takového pokusil o pět let dříve, tak by zajisté neuspěl, neboť jediná možnost, jak tehdy dostat hru k zákazníkovi bylo skrze komerční distribuční síť. Avšak v roce 2006 bylo již pokrytí širokopásmovým Internetem dostatečně velké, aby bylo možné hry distribuovat i tímto způsobem. Mezi jejich hry patří například **World of Goo** (viz obrázek 3.3) [2].



Obrázek 3.3: Screenshot ze hry World of Goo

3.3 Současné trendy

3.3.1 Typy indie game

Současná nezávislá herní scéna se dělí na dvě základní větve - **casual** a **hard-core** [20]. Dva protikladné žánry se zaměřením na jiné cílové publikum.

Casual

Jedná se o velmi nenáročné hry s jednoduchým ovládáním, jasným cílem a nesložitým příběhem (pokud nějaký vůbec má). Hráč nemusí studovat složité manuály, aby si takovouto hru zahrál. Složitost úrovní se zvyšuje jen velmi pozvolna a dává hráči dost času na pochopení principu hry. Z těchto důvodu jsou většina *casual* hráčů ženy. Marketing je u těchto her minimální, většinou je stačí jen prodat distributorovi nebo je umístit na některý z *games-on-demand* portálů (viz kapitola 3.3.2).

Hard-core

Hard-core hry jsou pravým opakem *casual*. Často nemají triviální ovládání a jsou určeny pro hráče s určitou úrovní herních zkušeností. Někdy jsou dokonce tak komplikované, že je potřeba do hry implementovat tutoriál nebo alespoň nápovědu. Obtížnost u těchto her neroste pozvolna, ale již první úrovně bývají dosti obtížné. Mohou také obsahovat prvky násilí, krev a sexuální obsah, což může způsobit odmítnutí některých portálů a distributorů.

3.3.2 Způsob distribuce

V nedávné době se objevil nový způsob distribuce počítačových her mezi zákazníky, který jim umožňuje streamovat nebo stáhnout hru přímo do jejich počítače. Tento systém je nazýván *games-on-demand* (zkráceně GoD). Díky němu si může uživatel koupit a obdržet hru z pohodlí svého domova bez nutnosti navštívit kamenný obchod nebo si hru nechat posílat poštou. Tento způsob má však i své nevýhody (např. absence krabicové verze) a je na hráči, zda převáží nad výhodami [14]. **Naprostá většina indie her je distribuována právě pomocí systému games-on-demand** .

Existují čtyři hlavní **kategorie systému games-on-demand**, lišící se podle typu her, které distribuují, a způsobem, jak je distribuují:

- **Streamování her přes webové prohlížeče** – většinou se jedná o malé hry napsané na platformách jako Java, Shockwave nebo Flash. Díky malé velikosti her (většinou okolo 1 megabajtu) mohou hráči začít hrát v podstatě okamžitě. Dnes již hry v této kategorii zasahují do všech herních žánrů, ale zpočátku byly takto distribuovány především různé *puzzle* hry.

- **Hry, které jsou distribuovány jen přes Internet** – tyto hry nejsou streamovány přes webové prohlížeče, ale celý jejich obsah je stažen na uživatelův počítač. Většinou tyto hry nelze koupit v kamenném obchodě. **Právě takovýmto způsobem je distribuována drtivá většina indie her.**

- **Hry, které se prodávají v kamenných obchodech a jako doplňkový prodej i přes Internet** – pro tyto hry existují dvě hlavní kategorie:

    - **Obsah hry je stažen kompletně** – hráč nemusí mít při hraní přístup na Internet.

    - **Hybridní** – ze hry je stažena jen část a zbytek je stahován podle potřeby až při hraní. Hráč musí mít po celou dobu hraní přístup na Internet.

Vybral jsem pět poskytovatelů služby *games-on-demand*, které jsou v dnešní době nejvíce vývojáři a hráči využívány a stručně jsem shrnul jejich klady a zápory:

App Store

Na tomto portálu lze nalézt jen aplikace na zařízení společnosti Apple, tedy iPhone, iPod Touch nebo iPad. Než začne vývojář publikovat své aplikace, musí zaplatit poměrně vysoký vstupní poplatek 100 dolarů. Výrobky společnosti Apple jsou velmi rozšířené, a proto je zde šance, že se aplikace dostane k velkému množství lidí. Toto může být zároveň výhoda i nevýhoda, neboť se také může stát, že se vaše aplikace mezi kvanty ostatních jednoduše ztratí.

Android Market

Tento portál se specializuje na prodej aplikací pro mobilní telefony s operačním systémem Android společnosti Google. Jako u App Store je i zde vstupní poplatek, ale značně menší - jen 25 dolarů. Okolo Androidu existuje velká komunita vývojářů i hráčů a je tedy snadné nalézt pomoc při řešení problému. Tomu také dopomáhá velmi slušná dokumentace celého operačního systému. Android podporuje 2D i 3D grafiku, akcelerometry, dotykové displeje a další moderní technologie, a proto je vhodný i pro tvorbu her.

Wiiware

Jak již napovídá název, je tento portál zaměřen na hry, které běží na herní konzoli Nintendo Wii. I když jejich manažeři prohlašují, že Wii podporuje *indie game* vývojáře [18], je získání Wii Development Kitu téměř nemožné. První barierou je jeho cena, která se může vyšplhat až na 10 tisíc dolarů. Další podmínkou je, aby měli vývojáři předchozí zkušenost s vývojem kvalitních her. A poslední absurdní požadavek je, že kanceláře společnosti nesmí být v domě, ve kterém některý z vývojářů bydlí [17]. Nicméně Wii je velmi rozšířen a pokud již člověk kit získá, tak na tom určitě neprodělá.

Stream

Stream se zaměřuje na platformu PC a nedávno svou působnost rozšířil i na Mac. Ke stahování her je zapotřebí mít nainstalovanou aplikaci Stream Store a založený uživatelský účet u společnosti Valve. Stream vývojářům nabízí řadu API, které mohou implementovat do své hry a tím umožnit hráčům lepší komunikaci, získávání achivementů, zobrazovat herní statistiky aj.

4 Vývoj nezávislé a komerční hry

Vývoj *indie game* se od vývoje komerčních her značně liší. První z rozdílu je ve velikosti týmu. Na komerčních AAA hrách (velmi kvalitní hra s vysokým rozpočtem [22]) může pracovat až několik stovek vývojářů. Naopak u *indie game* je pravidlem, že by se na projektu nemělo podílet více než 5 lidí.

Dalším důležitým prvkem při vývoji *indie game* je totiž efektivita a především rychlost, které samozřejmě s narůstajícím počtem dalších lidí klesá. V takto malém týmu se nemá cenu zdržovat vybíráním vhodného vývojového modelu a podobných věcí. Pro velmi malé týmy (1-2 lidé) není dokonce ani potřeba sepisovat rozsáhlý *Game Design Document* (zkráceně GDD), ale stačí jen pár poznámek a komunikovat osobně [21].

Je potřeba vytýčit si milníky a striktně je dodržovat. Tyto milníky musí být splnitelné, proto není rozumné vytvářet například 3000 typů nepřátel, když máme k dispozici jen jednoho grafika, a to jen na poloviční úvazek. Bohatě nám budou stačit 3 typy [20].

	Nezávislý vývojový tým	**Komerční vývojový tým**
Počet pracovníků	1-5	Desítky až stovky
Rychlost a efektivita	Hraje velkou roli, zisk ze hry není tak velký, aby si tým mohl dovolit strávit vývojem velké množství času.	Pochopitelně i zde jsou velmi důležité, ale tým si může dovolit jít s určitými problémy více do hloubky a jejich řešením strávit více času.
Model vývoje	Pro tak malý tým je nepotřebný a jeho vypracování by zbytečně zabralo příliš mnoho času.	Naopak pro tak velký tým je velmi důležitý a je vhodné do něj investovat peníze a čas.
Game Design Document	U velmi malého týmu (1-2 lidé) není potřeba. Avšak pokud se jedná o složitější projekt, či větší tým, je dobré jej sepsat.	Velmi důležitý. Tvořen simultánně více lidmi (*game designer*, *level designer* a další). Mohou obsahovat až stovky stran.

Tabulka 1: Srovnání nezávislého a komerčního vývoje

5 Nástroje pro tvorbu indie game

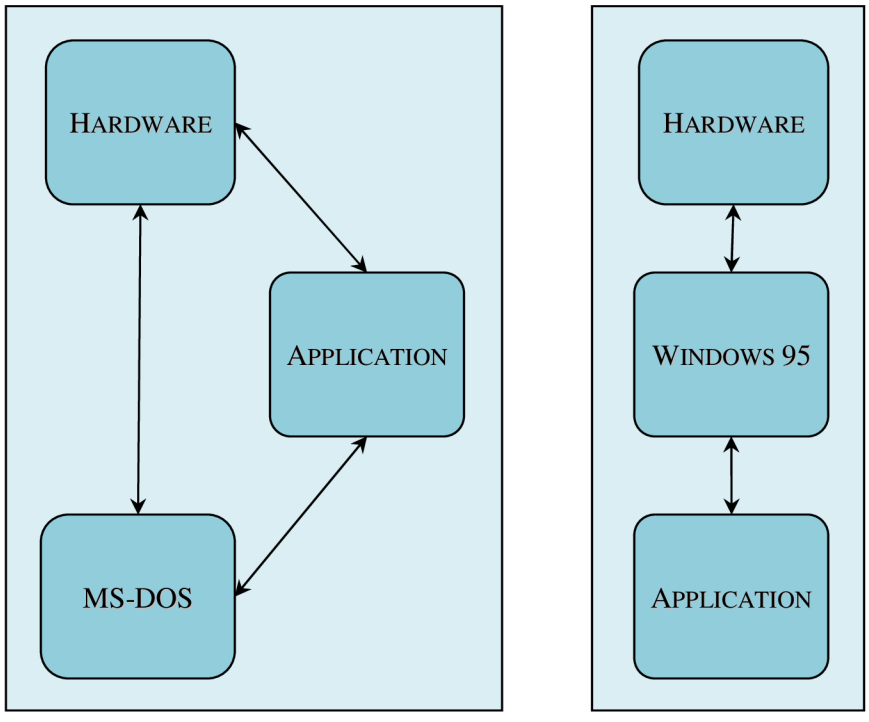
Tato kapitola obsahuje přehled grafických aplikačních rozhraní (anglicky Application Programming Interface a zkráceně API), herních enginů a nástrojů, které může vývojář použit při tvorbě nezávislé hry.

5.1 Aplikační rozhraní

Aby bylo možné využít potenciálu dnešního grafického hardwaru, musíme mít možnost s tímto hardwarem pohodlně pracovat. Samozřejmě by nebylo příliš pohodlné pracovat přímo s instrukcemi grafických karet, a proto existují API vyšší úrovně, která nad těmito instrukcemi vytvářejí jistou úroveň abstrakce. Toto dnes umožňují především dvě nejrozšířenější aplikační rozhraní - **DirectX** a **OpenGL**. V této kapitole si stručně popíšeme jejich historii a vlastnosti.

5.1.1 DirectX

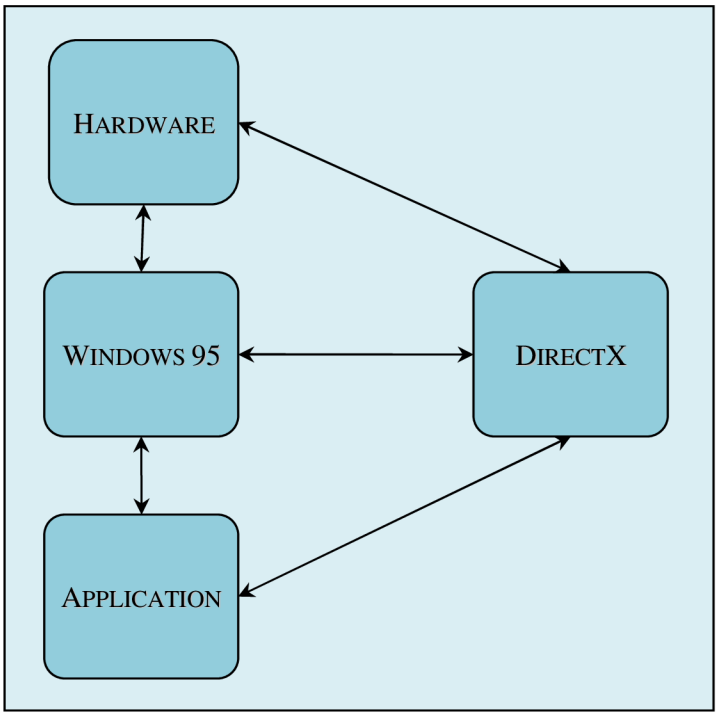
DirectX je vlastně souhrn aplikačních rozhraní, jejichž primárním účelem je práce s multimédii. Je zaměřen především na programování her a práci s videem. Celý DirectX je uzavřený standard a je vyvíjen a spravován pouze jedinou firmou - Microsoft.



Obrázek 5.1: Srovnání komunikace hardwaru s operačním systémem. Vlevo MS-DOS, vpravo Windows 95.

V roce 1994 byl vydán nový operační systém Microsoft Windows 95. Na rozdíl od předchozího operačního systému společnosti Microsoft, MS-DOS, neumožňovaly Windows 95 přímý přístup k periferiím počítače, jako je myš, klávesnice, zvuková zařízení nebo grafické karty (viz obrázek 5.1). Většina herních vývojářů proto dále vyvíjela hry pro MS-DOS a na Windows 95 se

dívala skepticky. Microsoft musel přijít se způsobem, jak jim tento přístup umožnit, avšak bez narušení chráněného paměťového modelu nového operačního systému. Řešení byl DirectX, jehož první verze (tehdy ještě pod označením Windows Games SDK) byla vydána v září roku 1995 (viz obrázek 5.2). Microsoft poté zahájil intenzivní vývoj DirectX, protože v něm viděl možnost, jak ke svému operačnímu systému přitáhnout nejen velké množství herních vývojářů a studií, ale posléze i samotných hráčů. Přibližně každý rok Microsoft vydává novou vylepšenou verzi. Aktuální verze je již DirectX 11.



Obrázek 5.2: Řešení problému s přístupem do chráněné paměti pomocí DirectX

DirectX je uzavřený standard, a tudíž nepodporuje rozšiřování své funkčnosti pomocí extenzí, což byla zpočátku nevýhoda oproti OpenGL, neboť podpora nových funkcí grafických karet je přidána vždy až s novou verzí. Dnes již Microsoft úzce spolupracuje s výrobci grafických karet a jiných periferních zařízení a nové verze jsou vydávány současně, nebo dokonce dříve, než hardware, který tyto funkce implementuje [24][25].

Na rozdíl od aplikačního rozhraní OpenGL je DirectX kompletně objektově orientovaný a je zaměřen pouze na operační systém Microsoft Windows. Existují však snahy o jeho implementaci i na jiných systémech (např. Wine pro Linuxové systémy).

Jak již bylo zmíněno výše, je DirectX souhrn aplikačních rozhraní. Jednotlivá aplikační rozhraní se označují jako komponenty. Skladba komponent se v jednotlivých verzích často liší. Většinou mezi verzemi dojde ke spojení nebo vypuštění některých komponent. Následující tabulky ukazuje a stručně popisuje komponenty verzí 9.0, 10.0 a 11.0 [26].

DirectX 9.0	
Komponenta	**Popis**
DirectDraw	Vykreslování 2D grafiky, zastaralé, jen kvůli zpětné kompatibilitě.
Direct3D	Vykreslování 3D grafiky. Podstatná část celého DirectX.
DirectPlay	Obstarává síťovou komunikaci.
DirectInput	Zpracování uživatelského vstupu (myš, klávesnice, gamepad...).
DirectX Media	Přehrávání multimediálního obsahu, streamování, akcelerace videa.
DirectMusic	Nahrávání a přehrávání hudby (mp3, ogg...).
DirectSound	Nahrávání a přehrávání zvuků (wav...).
DirectSound3D	Přehrávání prostorového zvuku.
DirectX Media Objects	Obsahuje kodeky, audio a video efekty.
DirectSetup	Pomocná komponenta pro instalaci součástí DirectX, detekci verze...

Tabulka 2: Komponenty DirectX 9.0

DirectX 10.0 a 11.0	
Komponenta	**Popis**
DirectGraphics	Tato komponenta slučuje komponenty, které mají na starost grafiku. Jsou to DirectDraw, Direct2D, Direct3D, DXGI a DirectWrite.
DirectCompute	Komponenta pro obecné výpočty na GPU.
DirectPlay	Zaměněný za Games for Windows neboli Live.
XInput	Náhrada za DirectInput. Podporuje i *next-get* ovladače.
Ostatní komponenty zůstaly stejné.	

Tabulka 3: Komponenty DirectX 10.0 a 11.0

5.1.2 OpenGL

OpenGL (Open Graphics Library) bylo poprvé veřejnosti představeno v roce 1992 společností Silicon Graphics (zkráceně SGI) a byla původně vyvinuta pro operační systém Irix. Vlastnictví sice náleželo společnosti SGI, ale o vývoj se starala OpenGL Architecture Review Board (zkráceně ARB), která se skládala z předních hardwarových a softwarových firem. Ty se pravidelně scházely, aby probraly změny ve specifikaci OpenGL a schválily směr jeho dalšího vývoje [27]. V roce 2006 převedla firma SGI kontrolu nad vývojem na konsorcium Khronos Group, do které však patří skoro všechny společnosti původní ARB.

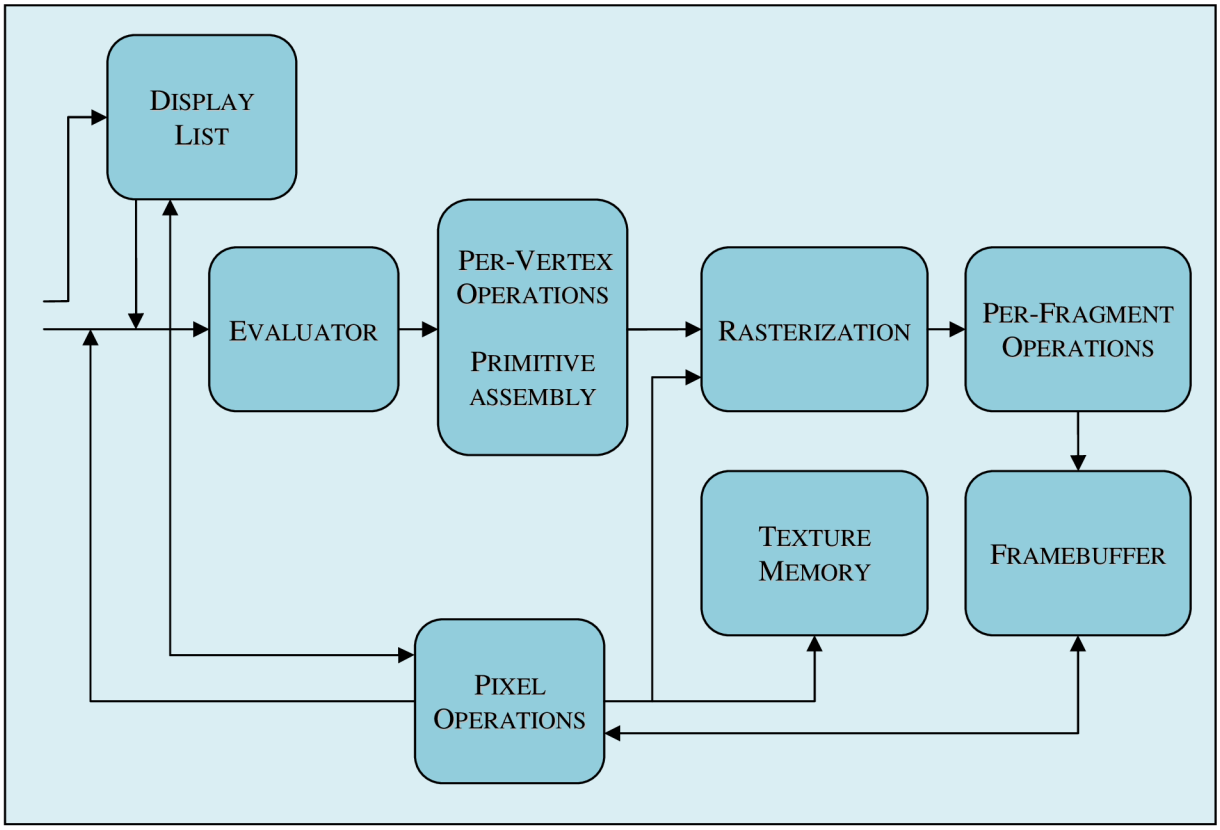
Cílem tohoto grafického aplikačního rozhraní byla hned od počátku nezávislost na platformě, hardwaru a programovacím jazyce [23]:

- **nezávislost na platformě** - Jak již bylo zmíněno, OpenGL byl původně vytvořen pro operační systém Irix, ale díky jeho multiplatformní architektuře je dnes dostupný na většině operačních systémů (Widnows, Unix, Linux, Irix, Sun...)

- **nezávislost na hardwaru** - OpenGL dokáže na rozdíl od DirectX veškeré funkce provádět i softwarově, bez přítomnosti grafického hardwaru, který by tyto funkce vykonal. V důsledku to znamená, že není potřeba ověřovat, zda dostupný grafický hardware tuto funkci zvládne, či nikoli.

- **nezávislost na programovacím jazyce** - Aby bylo dosaženo nezávislosti na programovacím jazyce, je na rozdíl od DirectX OpenGL procedurální (protože ne všechny jazyky podporují objektově orientovaný model). Dnes je OpenGL použitelný téměř v každém jazyce - C, C++, Fortran, ObjectPascal, Java, Ada, assemblery apod.

Tyto vlastnosti jsou hlavním důvodem, proč se OpenGL využívá v oblastech jako je Computer-aided design (CAD), architektura nebo filmový průmysl.



Obrázek 5.3: Vykreslovací řetězec OpenGL

OpenGL obsahuje přes 250 funkcí a jejich přetížení pro různé datové typy. Většina z těchto funkcí je určena buď k odeslání primitiva (bodu, úsečky, polygonu případně pixmapy nebo bitmapy) do vykreslovacího řetězce (viz obrázek 5.3) nebo ke změně jeho stavu. Tento stav platí pro všechna primitiva poslaná do řetězce, dokud není opět změněn. OpenGL je tady rozsáhlý a komplikovaný stavový stroj [27].

5.2 High-level nástroje pro tvorbu her

Aplikační rozhraní popsaná výše umožňují práci přímo s grafickým hardwarem na poměrně nízké úrovni abstrakce. Programátor sice nepoužívá přímo instrukční sadu konkrétního grafického hardwaru, ale je mu velmi blízko a může takto řídit i ty nejnižší vlastnosti a funkce. Tato nízká úroveň abstrakce však v určitých oblastech vývoje není výhodná nebo je dokonce nežádoucí. Jedna z těchto oblastí je právě vývoj her.

Jelikož je tvorba her velmi specializovaná oblast vývoje, je nasnadě, že existuje velké množství specializovaných nástrojů usnadňujících vývojářům jejich práci - jsou označovány jako **herní enginy** (anglicky **game engines**). Často poskytují běžné stavební bloky, ze kterých se skládá většina her - komponenty starající se o uživatelské rozhraní, fyzikální chování objektů, částicové systémy, hierarchické uspořádání objektů ve scéně (graf scény), zvuk, uživatelské vstupy, síťovou komunikaci (pro multiplayerové hry), umělou inteligenci a mnohé další. Obsahují také pomocné struktury pro práci s grafikou, jako jsou 2D a 3D vektory, quaterniony, matice, 2D a 3D textury, obalová tělesa (bounding volumes) a tak dále. Herním vývojářům tak usnadňují jejich práci, neboť nejsou nuceni vytvářet tyto základní bloky a struktury pro každou hru znovu.

5.2.1 Torque Game Engine Advanced

Torque Game Engine (zkáceně TGE) je dnes již zaniklý 2D a 3D herní engine, který byl původně vyvinut společností Dynamix v roce 2001 pro hru Tribe 2. Velká část vývojářů z týmu podílejícím se na tvorbě této hry odešla z Dynamix a založila společnost GarageGames. TGE si vzali s sebou, přepracovali jej, aby využívala nejnovější technologie a nakonec jej přejmenovali na Torque Game Engine Advanced (zkráceně TGEA). GarageGames byla později odkoupena společností InstantAction.

Velmi rychle se stal oblíbených nástrojem pro vývoj nezávislých her, neboť cena za licenci byla 75 dolarů na programátora, pokud společnost nevydělávala přes 250 tisíc dolarů ročně. Pokud ano, tak byla cena 2250 dolarů na programátora, což byla (v porovnání například s Unreal Enginem 2, jehož licence stojí 350 tisíc dolarů) nadále velmi nízká částka.

Dne 11.11.2010 však InstantAction oznámilo, že zastavuje všechny své aktivity a tím pádem i vývoj TGEA. V současné době stále hledá pro TGEA kupce, který by v jeho vývoji pokračoval [30].

Hlavní rysy

Torque Game Engine Advanced se dělí na několik variant, které se liší v platformě, na kterou jsou navrženy a zda jsou určeny pro 2D nebo 3D hru. Následuje jejich seznam:

- **Torque 3D a 2D** - Tyto varianty jsou určeny pro vývoj her na PC a Mac.
- **iTorque 2D** - Jak již první písmeno názvu napovídá, je tato varianta určena pro vývoj na platformách iPhone, iPad a iTouch. Zatím existuje jen 2D verze a jelikož je další vývoj enginu pozastaven, není ani jisté, zda bude někdy 3D verze existovat.
- **Torque X** - Tato varianta umožňuje vývoj na PC a Xbox 360. Na rozdíl od předchozích je napsána v jazyce C# a je postavena na Microsoft XNA Game Studio.

Torque je velmi rozsáhlý a propracovaný herní engine, proto zde zmíníme jen ty nejdůležitější hlavní rysy, které z něj dělají tak žádaný engine:

- Vykreslování 2D a 3D grafiky pomocí nejmodernějších metod.
- Skriptování, síť, zvuk.
- Vestavěné editory: terén, řeky a cesty, částicové systémy, materiály, úrovně.
- Pokročilé metody osvětlení: *Per-pixel* osvětlení, *normal & parallax occlusion mapping*, *volumetric fog* a *volumetric light*.
- Integrace fyzikálního enginu společnosti NVidia - PhysX.
- Velmi nízká cena za licenci.
- Pokrytí skoro všech platforem, na kterých lze vyvíjet a hrát hry.
- Obsáhlá a velmi kvalitní dokumentace.

Screenshoty

Ukázky z některých her, které byly vytvořeny pomocí herního enginu Torque Game Engine Advanced.

Obrázek 5.4: Screenshoty ze hry Dreamlords od společnosti Lockpick Entertainment

Obrázek 5.5: Screenshoty ze hry Buccaneer: The Pursuit of Infamy společnosti Stickman Studios

5.2.2 Unreal Engine

Unreal Engine je herní engine vyvíjený společností Epic Games. Jeho první verze byla použita v roce 1998 v FPS (*First Person Shooter*) hře Unreal. Stala se také základem pro další hry, jako jsou Unreal Tournament, Deus Ex, Turok a další. I když byl tento engine původně vytvořen pro FPS hry, byl úspěšně použit i v jiných žánrech, například v MMORPG (Massive(ly)-Multiplayer Online Role-Playing Game) hře Vanguard: Saga of Heroes.

Jelikož je jeho jádro napsáno v C++ a podporuje zobrazování pomocí DirectX i OpenGL, je velmi snadno přenositelný na různé platformy. Jeho nejnovější verze (Unreal Engine 3) je schopna fungovat na platformách Microsoft Windows, Linus, iOS, Mac OS, dále pak i na řadě konzolí - Dreamcast, Xbox, Xbox 360, Playstation 2 a Playstation 3.

Obsahuje řadu pomocných nástrojů, které vývoj hry velmi usnadňují. Skoro veškeré programování probíhá v proprietárním skriptovacím jazyce UnrealScript a není tak potřeba významně zasahovat do jádra enginu. Pro vytváření jednotlivých úrovní se používá editační nástroj nazvaný UnrealEd, který se velmi snadno ovládá a designer okamžitě vidí, jak bude úroveň ve hře vypadat.

Unreal Engine již existuje ve své třetí verzi a jsou na něm postaveny hry jako je Bioshock, Bioshock 2, Gears of War (všechny tři jeho díly), XIII, Unreal Tournament 3 a další. Je zajímavé vidět porovnání vykreslovacích schopností této verze s předchozími (viz obrázek 5.6).

Obrázek 5.6: Porovnání vykreslovacích schopností Unreal Enginu 1, 2 a 3 na modelu herní postavy Malcom

V zimě roku 2009 uvolnilo Epic Games verzi Unreal Enginu, která je pro nekomerční užití kompletně zdarma - nazvalo ji Unreal Development Kit (UDK). V případě komerční hry musí vývojáři zaplatit pouhých 99 dolarů a poté 25% ze zisku, pokud přesáhne 5000 dolarů. Epic Games se tak snaží rozšířit svůj engine i mezi *indie game* vývojáře, neboť tato nízká částka (vzhledem k tomu, jak je Unreal Engine propracovaný) je pro ně dosažitelná.

Epic Games oznámilo, že další verzi Unreal Enginu neplánují dříve, než se na trhu objeví nová generace herních konzolí. Jakmile se tak stane, bude Unreal Engine 4 vytvořen exkluzivně jen pro tyto konzole a s implementací na jiné platformy se nepočítá [33].

Hlavní rysy

- Velmi robustní herní engine, který obsahuje vše, co může vývojář potřebovat.
- Pro nekomerční užití od listopadu roku 2009 zdarma.
- Proprietární skriptovací jazyk UnrealScript - podstatná část vývoje probíhá právě v něm, bez nutnosti zasahovat do jádra enginu.
- Editor úrovní - UnrealEd - se velmi snadno používá. Designer okamžitě vidí, jak bude úroveň vypadat ve hře.
- Velké množství zabudovaných nástrojů - UnrealCascade (částicové systémy, exploze, ohně…), UnrealMatinee (in-game videa), UI Editor (uživatelské rozhraní), Sound Cue Editor (zvuk a hudba) a další.

Screenshoty

Ukázky z her, které byly na Unreal Enginu postaveny. Konkrétně se jedná o hry Unreal Tournament ve svých verzích 2004 a 3.

Obrázek 5.7: Screenshoty ze hry Unreal Tournament 2004

Obrázek 5.8: Screenshot ze hry Unreal Tournament 3

5.2.3 Microsoft XNA Game Studio

XNA Game Studio je sada knihoven a nástrojů, zaměřená na usnadnění tvorby počítačových her. Je to uzavřený produkt a o jeho vývoj se stará pouze společnost Microsoft, což zajišťuje výbornou podporu a dokumentaci. XNA je jednotná platforma pro vývoj na PC, Xboxu a dokonce i na Windows Phone 7. Jedná se v podstatě o zapouzdření funkcí rozhraní DirectX 9.0 s vysokou úrovní abstrakce. Avšak pokud se k tomu programátor rozhodne, má stále možnost přímo pracovat s nejelementárnějšími funkcemi grafického hardwaru. Celé XNA je vystavěno na .NET frameworku 2.0 a tak lze k vývoji her teoreticky použít kterýkoli z jazyků kompatibilních s .NET (například Visual Basic, Visual J# nebo Visual C++), avšak jen Visual C# je oficiálně podporován a navíc je jako jediný integrován do Visual Studia tak, aby při vývoji XNA hry programátorovi napomáhal při psaní zdrojového kódu.

Základem každého projektu je instance třídy `Game`, která obsahuje virtuální metody, jež XNA volá automaticky v následujícím pořadí. Do těchto metod vývojář implementuje většinu zdrojového kódu [35].

- `Initialize` - Volá se jako první a slouží k inicializaci všech negrafických zdrojů (např. otevření souborů, inicializace pomocných struktur atd.). Je zavolána jen jednou po spuštění hry.
- `LoadContent` - Tato metoda je zavolána přímo z metody `Initialize` a slouží k načtení veškerého grafického obsahu hry (např. modelů, textur atd.). Je zavolána jednou při spuštění hry a pak vždy, když je potřeba znovu načíst grafické zdroje, například při restartu grafického zařízení.
- `Update` - Aktualizuje herní mechanismy (např. fyziku, pohyb hráče, stav jeho zdraví atd.)
- `Draw` - V této metodě probíhá vykreslování scény.
- `UnloadContent` - Metoda je zavolána při ukončení hry a má za úkol uvolnit všechny načtené grafické zdroje.

I když XNA tvorbu her velmi usnadňuje, nelze jej označit za plnohodnotný herní engine. Neobsahuje v podstatě žádné obecné herní komponenty, jako jsou například graf scény nebo podpora uživatelského rozhraní. Je však možné vytvářet herní komponenty a služby, které lze poté bez velkého úsilí zakomponovat do jakéhokoliv XNA projektu.

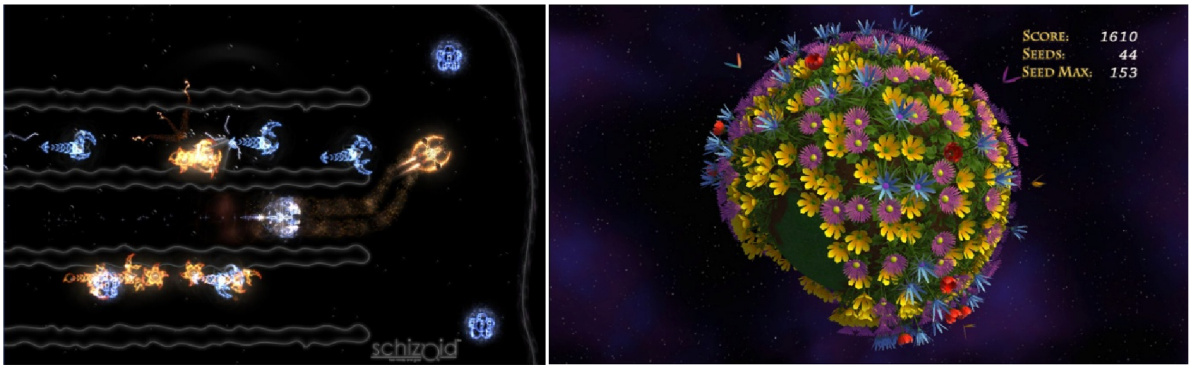
K vytvoření komponenty stačí naimplementovat svou třídu, která dědí z XNA třídy `GameComponent` nebo `DrawableGameComponent` (pokud má komponenta něco přímo vykreslovat do scény - například komponenta zobrazující aktuální počet snímků za sekundu). Takto vytvořená třída má pak stejné metody jako třída `Game` a již ji stačí jen zaregistrovat, aby byly tyto metody automaticky volány.

Jakákoli třída může implementovat rozhraní, které poté může nabízet jako službu. Tuto službu musí, podobně jako u komponent, zaregistrovat u hlavní třídy `Game`. Poté může kterákoli třída s přístupem k objektu `Game` požádat o poskytovatele této služby a využívat jej bez nutnosti mít jakýkoli vztah s jeho konkrétní třídou [34].

Hlavní rysy

- *High-level* zapouzdření knihoven DirectX 9.0
- Multiplatformní - snadná konverze mezi platformami PC, Xbox, Zune a Windows Phone 7 - většinou stačí velmi málo úprav.
- Integrace do Visual Studia - velmi usnadňuje a urychluje vývoj.
- Neobsahuje základní herní komponenty - např. graf scény, fyziku atd.
- Podporuje jen platformy kompatibilní s Windows.
- Velmi vysoká znovupoužitelnost kódu díky herním komponentám a službám.

Screenshoty



Obrázek 5.9: Screenshoty z her Schizoid a Culture

Obrázek 5.10: Screenshot ze hry Racing Starter Game

6 Tvorba enginu

Engine vznikal víceméně současně s hrou a reflektoval požadavky a potřeby, které při jejím vývoji vznikaly. Jako celek jej lze proto bez větších úprav použít jen pro tvorbu her stejného žánru - tedy *Real-time strategy* (dále jen RTS). Nicméně většinu jednotlivých komponent lze samostatně znovupoužít i v ostatních herních projektech s jiným žánrem.

Engine, který bude v následujících kapitolách popsán, je vystavěn nad platformou **XNA Game Studio 4.0**, jejíž použití je jednou z podmínek soutěže **ImagineCup**, kam byla hra přihlášena.

6.1 Návrh enginu

Před samotným vývojem hry by se měli vývojáři zamyslet a uvážit, zda není jejich hra natolik jednoduchá, že žádný složitý engine vlastně ani nepotřebuje. V takovém případě je lepším řešením vytvořit kód specifický jen pro danou hru a s vývojem obecného enginu neztrácet čas. Dalšími kriterii, která mohou ovlivnit toto rozhodnutí, jsou čas a prostředky, které má tým k dispozici. Prostředky se zde nemyslí jen ty finanční, ale například i počet a zkušenosti vývojářů.

Vzhledem k rozsáhlosti našeho projektu se nedalo o "bezenginové" variantě ani uvažovat. Problémem však byl čas, kterého nebylo nazbyt, a nechtěli jsme skončit jako mnoho ostatních týmů, které sice vytvořily velmi obecné a rozsáhlé enginy, avšak jim již nezbyl čas na vytvoření a doladění samotné hry.

Z těchto důvodů jsme zvolili kompromis mezi obecností a časovou náročností a vytvořili malý kompaktní engine, který byl vystavěn na míru naší hře. Kromě základních komponent, vznikala většina funkcí až podle aktuálních potřeb a požadavků při vývoji.

Dále jsme se rozhodli využít systém komponent a služeb, které XNA nabízí a nevytvářet centrální třídu, která by jako své členy měla jednotlivé části enginu. Místo toho je každá část zděděna z XNA komponenty nebo služby. Takto si může vývojář, používající náš engine, snadno a podle svých potřeb zvolit, které části využije a které ne. Stačí naši komponentu přiřadit hře jako aktivní. Má pak i větší volnost ve volbě pořadí updatování a vykreslování jednotlivých komponent.

6.1.1 SceneManager

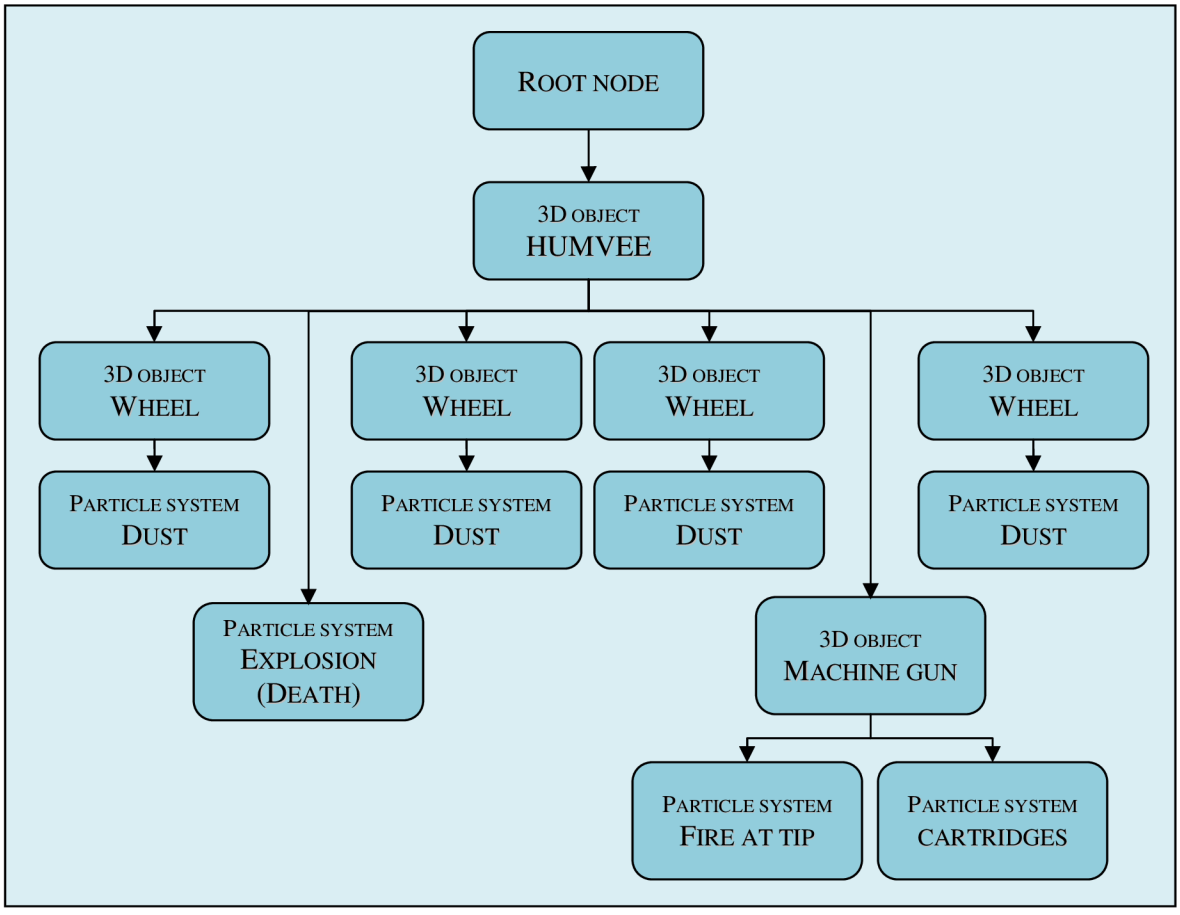
Velmi důležitou části každého herního enginu je komponenta, která reprezentuje takzvaný graf scény. Je to kolekce uzlů (anglicky *nodes*) uspořádaná do nějaké hierarchické struktury, nejčastěji však do grafu nebo stromu. Jednotlivé uzly jsou spojeny hranami a dohromady tak vytvářejí hierarchii scény. Uzly mohou reprezentovat různé typy entit, které se v každé hře mohou více či méně lišit. Existuje však několik základních typů, které jsou ve většině her stejné, např. předdefinované 3D objekty (krychle, válec…), libovolný animovaný/neanimovaný 3D objekt nebo různé zdroje světla [36].

Vyjma kořenového uzlu, který je vstupním bodem grafu scény, má každý uzel svého rodiče. Ty, které mohou mít potomky, nazýváme skupinové (*group*) a ty které nemohou, jsou listové (*leaf*). Při návrhu našeho grafu scény jsme však tuto možnost zanedbali a všechny naše uzly vytvořili jako skupinové. Tímto opatřením se implementace velmi zjednodušila a přitom byly zachovány všechny funkce grafu s listovými uzly.

Hierarchické uspořádání uzlů není zvoleno jen kvůli snadnějšímu a rychlejšímu hledání konkrétního uzlu, ale také proto, že jakákoli změna rodiče se projeví stejně i na jeho potomcích. Díky

tomuto lze snadno sdružovat části jednoho objektu do skupin, které lze měnit a ovládat jako celek, a přitom si zachovat kontrolu nad jednotlivými podobjekty.

Nejlépe to popisuje obrázek 6.1, na kterém vidíme jednoduchou stromovou hierarchickou strukturu, která popisuje vozidlo Humvee. Z obrázku je patrné, že se skládá z hlavního objektu, ke kterému jsou připojeny čtyři kola, kulomet a částicový systém, který je aktivován, pokud je vozidlo zničeno (exploze). Jednotlivé části mají také své potomky. Každé kolo má částicový systém generující za jízdy vozidla prach, kulomet má částicové systémy dokonce dva - jeden simuluje oheň u hlavně při střelbě a druhý generuje prázdné nábojnice. Toto rozdělení nám dává dostatečný stupeň kontroly nad jednotlivými částmi Humvee a zároveň můžeme ovládat vozidlo jako celek. Pokud například pootočíme kola tak, aby směřovala ve směru jízdy, upraví se i pozice a natočení připojeného částicového systému. Také je možné natočit kulomet požadovaným směrem bez toho, aby bylo nutné rotovat s celým vozidlem. Připojené částicové systémy budou samozřejmě rotovat s kulometem a zachovají si tak správnou pozici.

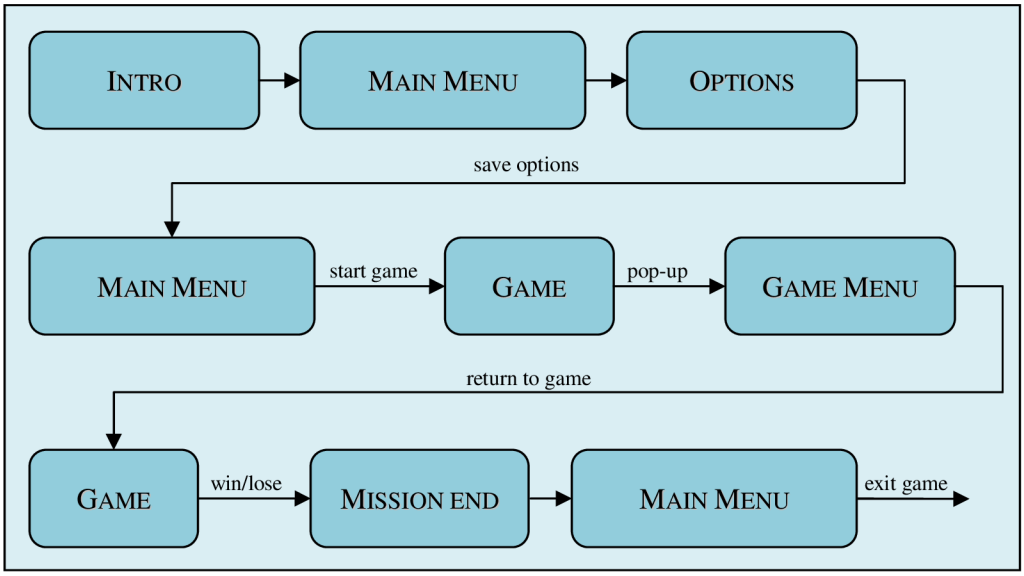


Obrázek 6.1: Ukázka jednoduchého grafu scény

6.1.2 ScreenManager

Většina her se skládá z různého počtu obrazovek, mezi kterými se hráč svou interakcí pohybuje. Pojem obrazovka zde není chápán jen jako hlavní menu nebo obrazovka s nastavením, ale může se jednat i samotnou hru - herní logika může být naprogramována přímo v jedné obrazovce. Tato komponenta byla navržena tak, aby umožňovala vývojáři takovéto obrazovky vytvářet, spravovat a

přepínat mezi nimi. Tento návrhový vzor jsme převzali z ukázkové aplikace společnosti Microsoft [37].



Obrázek 6.2: Ukázka hypotetického screen-flow diagramu

6.1.3 Input

I takto malý engine by měl být schopen poskytnout vývojáři snadný a intuitivní přístup k uživatelským vstupům. Ty je však potřeba zpracovávat na různých místech hry. Bylo by nepohodlné si tuto komponentu neustále předávat a ukládat v každé třídě zvlášť, a proto jsme se implementovali *Input* jako XNA službu. Dále jsme se rozhodli nepodporovat Xbox, a tudíž jsme se soustředili jen na myš a klávesnice.

Tato služba by měla být schopná zachytávat následující standardní události:

- Stisknutí, držení a uvolnění klávesy
- Stisknutí, dvojklik, držení a uvolnění tlačítka myši
- Získat a nastavit pozici myši
- Získat pootočení kolečka myši

Dále jsme chtěli vývojáři zajistit nějakou kontrolu nad pohybem myši, a proto jsme do návrhu zařadili i následující funkce:

- Uzamknutí pozice myši na určitém místě obrazovky
- Uzamknout pohyb myši v určité oblasti obrazovky - většinou v okně hry
- Zjištění, u kterého okraje/rohu obrazovky je myš - pro pohyb kamery v RTS hrách

6.1.4 GUIManager

Velmi důležitou součástí každé hry je grafické uživatelské rozhraní (*Graphical User Interface*, dále jen GUI), pomocí kterého hráč hru ovládá, případně mu hra skrze něj poskytuje různé informace o

stavu hry. Navrhli jsme proto komponentu *GUIManager*, která se o vše okolo GUI stará. Dopředu jsme si vytipovali prvky, které budeme bezpodmínečně potřebovat a zahrnuli je do návrhu:

- **Container** - nemá žádné grafické znázornění a slouží jen jako kontejner pro ostatní prvky
- **Button** - klasické tlačítko, jehož grafické znázornění se může měnit podle jeho stavu - *normal*, *hovered*, *disabled* a *clicked*
- **Image** - obrázek, lze jej použít například pro zobrazení statických částí GUI
- **Scrollbar** - posuvník, často používaný na posouvání čteného textu nebo pro nastavení hodnoty z nějakého rozsahu
- **Label** - jednoduché textové pole s možností standardního zarovnávání textu, včetně zarovnání do bloku
- **NinePatch** – libovolně roztažitelný obrázek, který se skládá z 9 menších obrázků (4 rohy, 4 hrany a vnitřní část)

Každý prvek je přímo, či nepřímo zděděn z obecného prvku. Tímto se eliminuje nutnost znovu a znovu opakovat tentýž kód pro parametry, které jsou u všech prvků totožné, např. pozice nebo velikost.

Dalším důležitým bodem bylo navrhnout hierarchickou strukturu, ve které by byly prvky uloženy. Vytvořením takovéto struktury se velmi usnadní práce se skupinami navzájem souvisejících prvků, např. prvky jedné obrazovky. Abychom obrazovku skryli, museli bychom skrýt každý prvek zvlášť. V hierarchické struktuře však stačí skrýt jejich společného rodiče, což velmi zjednoduší a zpřehlední implementaci herního GUI. Princip je velmi podobný jako u grafu scény popsaného v kapitole 6.1.1.

Následně bylo potřeba vyřešit řízení pořadí vykreslování jednotlivých prvků. Standardní a nejjednodušší metoda je využití již existující hierarchické struktury a vykreslovat prvky v pořadí v jakém byly přidány. Změnu pořadí pak lze realizovat přesunem prvku na začátek, resp. na konec seznamu potomků (*bring to front*, resp. *send to back*).

Druhá varianta je nezávislá na pořadí přidávání, avšak je náročnější, a to jak z hlediska výkonu, tak z hlediska implementace. Každému prvku je při vytváření přiřazena hloubka, kterou však lze za běhu kdykoli změnit. Prvky jsou pak podle této hloubky seřazeny a následně vykresleny. I zde je však nutné brát v potaz hierarchii, a tedy hloubky rodičů. Nejjednodušší se ukázalo zvyšovat hloubku o konstantu s každou úrovní zanoření v hierarchické struktuře.

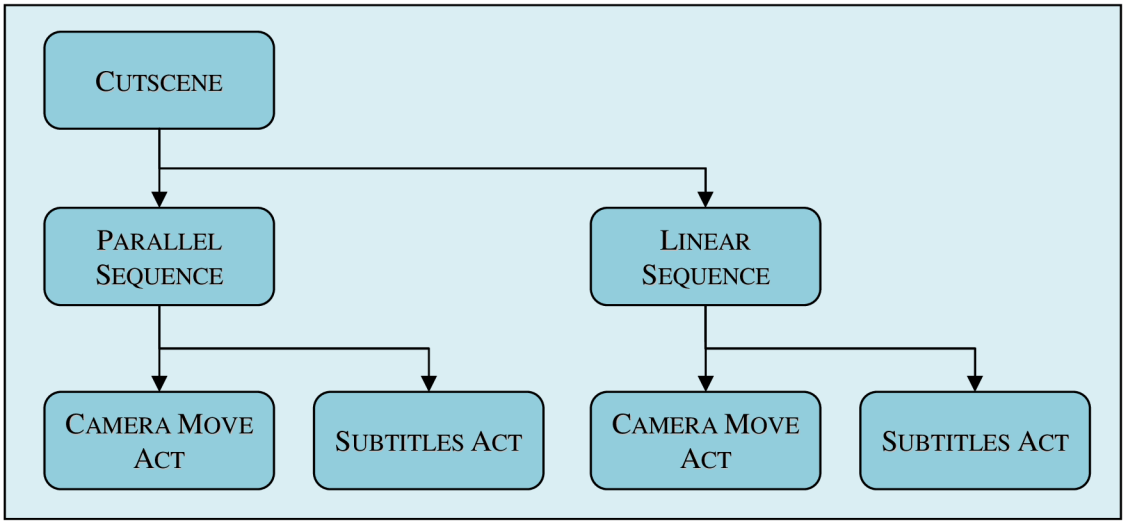
Je zřejmé, že herní GUI musí být interaktivní a zachytávat alespoň základní události, jako jsou *MouseDown*, *MouseMove*, *MouseUp*, *MouseWheel*, *MouseEnter* a *MouseLeave*. V některých případech je výhodné přesměrovat všechny zachycené události jen jednomu konkrétnímu prvku (*mouse capture*), např. při tažení scrollbaru se často stane, že myš opustí oblast posuvníku a tím se tažení přeruší. Toto lze vyřešit aktivací přesměrování do scrollbaru při začátku tažení a následného zrušení přesměrování při ukončení tažení.

6.1.5 CutsceneManager

Pro vytváření předem připravených in-game animačních sekvencí - cutscén - jsme navrhli komponentu nazvanou *CutsceneManager*. Tyto sekvence lez využít například k uvedení hráče do děje konkrétní mise nebo vytvoření tutoriálu.

Každá cutscéna je rozdělena do několika sekvencí, které se dále dělí na akty. Sekvence může být dvojího typu - lineární nebo paralelní. Pokud je lineární, jsou akty této sekvence vykonány

postupně za sebou, v případě paralelní sekvence jsou spuštěny všechny akty v jeden okamžik. Na obrázek 6.3 je první sekvence paralelní, a tedy se kamera začne pohybovat ve stejnou chvíli, v jaké se objeví titulky. Druhá sekvence je totožná s první až na to, že je lineární. To znamená, že titulky se zobrazí až po tom, co kamera ukončí svůj pohyb.



Obrázek 6.3: Ukázkové schéma cutscéna

Následující seznam ukazuje několik typů aktů, které jsou obecné a je možné je použít v kterémkoli žánru:

- **Move camera** - plynule přesune kameru na zadané souřadnice
- **Move camera look at** - plynule přesune kameru tak, aby se dívala na zadané souřadnice
- **Wait** - vyčká po zadaný časový úsek
- **Play sound** - přehraje zadaný zvuk

Vývojář si samozřejmě může velmi snadno vytvořit specifické akty, které jeho hra potřebuje. Například pro žánr RTS jsme navrhli tyto další akty:

- **Move RTS camera** - plynule přesune kameru na zadané souřadnice, přičemž zachová přiblížení RTS kamery
- **Show target marker** - na zadaných souřadnicích zobrazí na terénu značku
- **Wait for unit selection** - vyčká, dokud hráč neoznačí jednotku daného typu

6.1.6 Camera

Další nepostradatelná komponenta jakéhokoli enginu je bezesporu kamera. Bylo potřeba vytvořit abstrakci nad projekční a pohledovou maticí, neboť s těmito entitami se člověku velmi špatně pracuje a těžko si je představuje. Daleko přirozenější je pro něj kameru chápat jako bod s pozicí a rotací v prostoru. Všechny operace spojené s vytvářením a manipulaci s oběma výše zmíněnými maticemi pak komponenta udělá skrytě. Musí být také schopná počítat i další užitečné a často využívané parametry kamery, např. vektor, podél kterého se kamera dívá (*forward vector*), pohledové těleso (*viewing frustum*) atd.

Jelikož bude tato kamera používána většinou ve hrách, je nutné, aby s ní mohl hráč v prostoru pohybovat. K tomu je zapotřebí možnost nastavení maximální rychlosti pohybu a maximální rychlosti rotace. Nakonec je potřeba umožnit vývojáři nastavení kláves, které budou kameru ovládat.

Každý žánr, a někdy dokonce i dvě různé hry stejného žánru, mají na kameru různé požadavky. Z tohoto důvodu byla tato část enginu navržena co možná nejobecněji a nejjednodušeji, aby z ní bylo možné podle potřeby dědit a vytvářet tak specializované kamery. Prozatím jsme navrhli tyto:

- **Free camera** - kamera nemá žádná omezení a hráč ji může pomocí kláves a myši přesunout na libovolné místo prostoru
- **RTS camera** - pohyb kamery se ovládá pomocí kláves, ale také najetím myši k okraji okna. Otáčením kolečka myši je možné kameru přibližovat a po stisknutí prostředního tlačítka je možné kamerou rotovat okolo bodu, na který se právě dívá.

6.1.7 Cursor

V dnešních operačních systémech je běžné, že se kurzor myši mění podle situace a dává tak uživateli informaci o tom, co se právě děje, případně, co se při kliku stane či nestane. Ve hrách tomu není jinak, ba dokonce má tato funkce často důležitější úlohu než v desktopových aplikacích - lze například ihned poznat, kdo je nepřítel, či přítel, zda s předmětem pod kurzorem jde provést nějakou akci atd. V neposlední řadě je kurzor estetická záležitost a existuje jen málo her, které by neměly vytvořené své vlastní, sladěny s grafickým stylem hry.

Navrhli jsme tedy velmi jednoduchou službu, která by se o změny kurzoru starala a která by jej dokázala v případě potřeby i skrýt a znovu zobrazit.

6.1.8 Audio

Aby byla hra kvalitní a konkurenceschopná, musí kromě dobrého vizuálního vjemu poskytovat i kvalitní zvuky a hudbu. O to by se měla starat právě komponenta *Audio*, která musí být schopná pomocí jednoduchého rozhraní přehrávat zvuky, umisťovat je do trojrozměrného prostoru a v případě potřeby je přehrávat v nekonečné smyčce.

V XNA existují dvě možnosti, jak načíst a přehrát zvuk. První z nich se hodí na jednoduché přehrávání typu *fire-and-forget*, takže není velmi vhodná pro složitější hry, které vyžadují úplnou kontrolu nad přehráváním. Tato komponenta byla proto hned od začátku zamýšlena jako zjednodušení a zapouzdření poněkud kostrbatého a těžkopádného rozhraní knihovny *Cross-platform Audio Creation Tool* (dále jen XACT).

6.1.9 DebugOutput

Pro výpis pomocných textů a počtu snímků za sekundu (*Frames Per Second*, dále jen FPS) při ladění hry jsme navrhli třídu *DebugOutput*, která tyto informace zobrazuje do levého horního rohu obrazovky. Tato třída není XNA komponenta jako většina části našeho enginu, ale je zděděna z XNA služby. Výpis pomocných textů je totiž podobně jako přístup k uživatelským vstupům potřeba v mnoha místech enginu a hry.

6.2 Implementace enginu

V této kapitole naleznete konkrétní postupy při implementaci výše navržených komponent a služeb. U většiny naleznete napravo hned pod nadpisem tabulku, které představuje rozhraní dané komponenty. Obsahuje však jen důležité metody a položky. Měla by pomáhat v pochopení některých implementačních postupů.

Před samotnou implementací je vhodné dohodnout, jak budete kód strukturovat, rozdělit úkoly a vytvořit milníky. Dobrou praxí je zvolit vedoucího projektu, který se bude mimo jiné starat o administrativu okolo vývoje - plánování, rozdělování úkolů, komunikaci s exernisty atd.

Pro výměnu, synchronizaci a verzování kódu je dobré použít některou metodu *Source Code Managementu* - např. SVN nebo Mercurial.

Osvědčilo se také založení projektové wikipedie pro zapisování poznámek, nápadů a TODO listů. V případě, že vlastníte svůj server, existuje hned několik bezplatných wiki balíčků, např. *MediaWiki*. Pokud server nemáte, můžete využít některou z bezplatných online služeb a wiki si založit u nich, např. *Wikidot*.

6.2.1 SceneManager

I když se jedná o jednu z nejdůležitějších částí našeho enginu, je jeho základní implementace poměrně jednoduchá. Graf scény jsme se rozhodli implementovat jen jako jednoduchý strom, jelikož dostačoval pro účely naší hry. Nepotřebovali jsme například, aby měl jeden objekt více rodičů nebo aby byly uzly provázány jinak než vztahem rodič-potomek.

SceneManager

```
Node RootNode;

StaticMeshNode AddStaticMeshNode(Model
model, Vector3 position, Node parent);
BillboardNode AddBillboardNode(Texture2D
billboard, Node parent, Vector3 position);

A další…
```

K přidávání uzlů různých typů slouží metody, které začínají slovem `Add` a končí slovem `Node`, např. `AddStaticMeshNode`, která přijímá jako parametr 3D model a pozici. Další takovou metodou je `AddPointLightNode`, která navíc ještě přijímá parametry světla (barva, utlumení atd.). Každá tato metoda má navíc parametr, který určuje, ke kterému uzlu bude navázána jako potomek.

Při každém updatu je tento strom rekurzivně procházen a každý uzel je aktualizován. Vstupním bodem této rekurze je kořenový uzel, který není reprezentován žádným objektem, avšak hraje velmi důležitou roli, neboť každý uzel si udržuje pole svých potomků a tento není výjimkou. Slouží totiž jako rodič objektům nejvyšší úrovně scény. Každý uzel je při aktualizaci nejdříve transformován pozicí, rotací a škálou svého rodiče a teprve poté jsou k těmto transformacím přidány i jeho vlastní, které jsou však již relativní k rodiči. Výsledek je uložen do položky `AbsoluteMatrix`. Takto lze vytvářet i složité systémy provázaných objektů. Pro získání nebo změnu pozice, rotace a škály, relativní k rodiči, lze použít položky

Node

```
Vector3 Position;
Quaternion Rotation;
Vector3 Scale;
bool Visible;
Node Parent;
List<Node> Children;
Matrix RelativeMatrix;
Matrix AbsoluteMatrix;
BoundingSphere BoundingSphere;
BoundingBox BoundingBox;
bool IsCulled;

void AddChild(Node node);
bool RemoveChild(Node node);
void RemoveFromParent();
```

`Position`, `Rotation` a `Scale`, které má každý typ uzlu. U některých typ však některé položky nemají význam – např. rotace u všesměrového světla.

Bylo by zbytečné vykreslovat objekty scény, které nejsou vidět. Proto jsme implementovali velmi jednoduchý algoritmus, který přímo při vykreslování určuje, zda je objekt v záběru kamery, či nikoli. V XNA je implementace velice snadná, neboť zde existují předpřipravené struktury pro obalová tělesa (`BoundingShpere`, `BoundingBox` a další), která mají metody na zjištění vzájemného průniku a také, zda jedno těleso obsahuje druhé. Ke zjištění viditelnosti modelu pak již jen stačí tyto metody zavolat a jako parametry jim předat pohledové těleso kamery (viz obrázek 6.5) a obalové těleso modelu. Pokud jsou v prostoru pohledového tělesa, nebo jej jen protínají, jsou vykresleny běžným způsobem. V opačném případě je jejich vykreslování přeskočeno. Tato metoda se nazývá *view frustum culling*.

Pro další zvýšení rychlosti by bylo možné implementovat jeden z algoritmů pro dělení 3D prostoru, například často používaný a oblíbený *octree*. Pro zvýšení přesnosti detekce kolizí lze použít hierarchii obalových těles (*Bounding Volume Hierarchy*).

6.2.2 ScreenManager

Tuto komponentu jsme implementovali podle návrhového vzoru převzatého z ukázkových příkladů k XNA Game Studio 4.0 [citace]. Její hlavní funkcí je spravovat obrazovky hry, mezi kterými se hráč pohybuje. Aktivní (viditelné) obrazovky jsou uloženy v seznamu `Screens`, který je každý update procházen, a jednotlivé obrazovky jsou takto aktualizovány. Před samotným procházením je nutné seznam `Screens` zkopírovat do pomocného seznamu `ScreensToUpdate`, neboť by mohlo dojít k narušení správného pořadí aktualizací, např. pokud by byla jedna obrazovka odebrána jinou.

ScreenManager

```
List<GameScreen> Screens;
List<GameScreen> ScreensToUpdate;

void AddScreen(GameScreen screen);
void RemoveScreen(GameScreen screen);
```

GameScreen

```
bool IsPopup;
bool OtherScreenHasFocus;
ScreenState ScreenState;

void HandleInput(GameTime time);
```

Metodou `AddScreen` lze obrazovky přidávat do seznamu `Screens` a tím je zařadit do pravidelné aktualizace. Odstranit je lze voláním metody `RemoveScreen`.

Aktualizace je provedena pro každou aktivní obrazovku, avšak metoda, ve které jsou zpracovávány vstupy od uživatele, `HandleInput`, se volá jen pro tu, která byla přidána jako poslední a je tedy v hierarchii obrazovek nejvýše. Takováto obrazovka má nastavenu položku `ScreenState` na hodnotu `Active`. Tím je zajištěno, že vstupy z klávesnice a myši dostává opravdu jen aktivní okno.

Obrazovce je při její aktualizaci zasílána také informace o tom, zda je celá překryta jiným oknem. Toto se neřeší opravdovou kontrolou překrývání, ale libovolné obrazovce je možné nastavit parametr `IsPopup`. Takováto obrazovka nepřekrývá obrazovky pod sebou. Je pak na vývojáři, jak s touto informací naloží a zda ji vůbec využije.

6.2.3 Input

Tuto službu lze implementačně rozdělit do tří částí. První, resp. druhá část obstarává zachytávání uživatelských vstupů z klávesnice, resp. myši. Poslední část zajišťuje vývojáři možnost omezení pohybu myši.

Klávesnice

Input
keyboard

```
bool KeyDown();
bool KeyDown(Keys key);
bool KeyHold(Keys key);
bool KeyUp(Keys key);
```

Platforma XNA dokáže pomocí třídy `Keyboard` a její metody `GetState` zjistit aktuální stav celé klávesnice v daném momentě. My ovšem potřebujeme zjistit stav pouze jedné klávesy a navíc rozlišit právě stisknutou klávesu (*KeyDown*), právě uvolněnou klávesu (*KeyUp*) a úsek mezi těmito dvěma událostmi (*KeyHold*).

Pro zjištění právě stisknuté klávesy se nám nabízí využít přímo metodu `GetState` a z vráceného seznamu zjistit, zda je v něm i námi požadovaná klávesa. Tento přístup však selže v případě nízkého FPS, kdy může hráč stisknout a uvolnit klávesu v časovém úseku mezi dvěma updaty a tak vůbec nedojde k rozpoznání události. Problém také nastane, pokud hráč klávesu podrží delší dobu, než trvá jeden update. *Input* totiž nemá žádnou informaci o tom, že klávesa již byla v minulém updatu stisknuta a událost *KeyDown* rozpozná znovu, přestože by mělo být rozpoznáno *KeyHold*. Je jasné, že tímto způsobem nelze ani zajistit správnou funkci události *KeyUp*.

Z těchto důvodů jsme implementovali velmi jednoduchou, avšak účinnou metodu, která minimalizuje tyto neduhy. Metoda si po provedení updatu uloží stav klávesnice, což jí v následujícím updatu umožní porovnat starý stav s tím aktuálním a vyhodnotit tak, která událost mezi updaty nastala. Vyhodnocení probíhá podle následujícího schématu:

	Je klávesa stisknuta?	
	Předchozí update	**Aktuální update**
KeyDown	NE	ANO
KeyHold	ANO	ANO
KeyUp	ANO	NE

Tabulka 4: Schéma vyhodnocování události klávesnice

Rozhraní je velmi intuitivní a jednoduché - vývojáři stačí jen zavolat metodu `KeyDown`, `KeyUp` nebo `KeyHold`, jako parametr jí předat klávesu, která jej zajímá a metoda vrátí, zda tato událost pro tuto klávesu nastala.

Ve hrách všech žánrů se často vyskytuje notoricky známá pobídka ke stisku libovolné klávesy. Aby vývojář nemusel testovat všechny klávesy přímo v kódu své hry, vytvořili jsme také bezparametrovou metodu `KeyDown`, která tuto práci udělá za něj.

Myš

Input
mouse

```
byte MouseButtonsPressed;
EdgeFlag EdgeFlags;
int EdgeWidth;
float DoubleClickTime;

Point GetMousePosition();
void SetMousePosition(int x, int y);
bool MouseDown();
bool MouseDown(MouseButtons button);
bool MouseHold(MouseButtson button);
bool MouseUp(MouseButtons button);
int GetWheelScroll();
bool MouseDoubleClick();
```

I pro práci s myší existuje třída `Mouse` s metodou `GetState`, která vrací aktuální pozici ukazatele a stav tlačítek. A i zde nastal stejný problém se správným rozpoznáváním událostí jako u klávesnice. Naštěstí lze tento problém vyřešit velmi podobným způsobem. Jediný rozdíl je v tom, že třída, kterou vrací metoda `Mouse.GetState`, má pro každé tlačítko zvláštní položku a neexistuje žádná metoda, jež by jako parametr brala například číslo tlačítka a vracela jeho stav. Naštěstí je tlačítek na myši poměrně málo a zkontrolovat všechny tyto položky není příliš náročné. Nechtěli jsme však toto nepohodlí přenášet i na uživatele našeho enginu, a proto jsme raději vytvořili výčtový typ (*enumeration*) `MouseButtons`, jehož hodnoty reprezentují jednotlivá tlačítka. Tyto hodnoty pak vývojář může předat jako parametr jedné z metod `MouseDown`, `MouseHold` nebo `MouseUp`, které se chovají stejně jako jejich klávesové ekvivalenty. I pro myš jsme se rozhodli implementovat bezparametrovou metodu `MouseDown`, která reaguje na libovolné tlačítko, a ulehčit tak vývojářům práci.

Každé otočení kolečka myši nahoru, resp. dolů změní hodnotu delta o +120, resp. -120. V případě, že uživatel vlastní myš, která umožňuje plynulé otáčení kolečka, nemusí být hodnota rovna násobku čísla 120. Podle MSDN [http://msdn.microsoft.com/en-us/library/system.windows.forms.control.mousewheel.aspx] může aplikace reagovat na jiné hodnoty než na násobky 120 a tím zjemnit reakci, ale měla by vždy zachovat poměr vůči tomuto číslu. Metoda `GetWheelScroll` tuto hodnotu nezměněnou vrací a tím nechává na vývojáři, jak s ní naloží.

Pro rozpoznání dvojkliku se ukládá, jaké tlačítko bylo stisknuto poslední a kdy bylo stisknuto. Při dalším stisku stejného tlačítka se odečte uložený časový údaj od aktuálního času. Zda je výsledek menší než `DoubleClickTime`, a tedy zda se jedná o dvojklik, může vývojář ověřit pomocí metody `MouseDoubleClick`. Hodnotu `DoubleClickTime` lze snadno měnit a tím přizpůsobit délku dvojkliku potřebám hry.

První tři položky vypsané v rozhraní jsou spíše pomocné a nejsou pro tuto komponentu stěžejní, avšak poskytují vývojáři další usnadnění práce a zvýšení komfortu. `MouseButtonsPressed` je bitové pole reprezentující stisknutá tlačítka. *Input* kontroluje, zda je myš u okraje okna a do `EdgeFlags` ukládá, který okraj či roh to je. Šířka okraje jde nastavit pomocí položky `EdgeWidth`.

Omezení pohybu myši

Input
mouse fix

```
void FixMouseBounds();
void FixMouseBounds(Rectangle b);
void UnfixMouseBounds();
void FixMouse();
void FixMouse(int x, int y);
void FixMouseAtCurrentPosition();
void UnfixMouse();
Point GetMoveFromFix();
```

Někdy je vhodné omezit pohyb myši jen po určitém prostoru, nejčastěji je tento prostor okno aplikace, ale může se jednat i o nějaký herní prvek a donutit tak uživatele zaměřit na něj svou pozornost. V případě omezení do okna je účel jasný - hráč tak v herním zápalu neopustí myší okno a nepřeruší tak hru. Toto omezení je vhodné aplikovat i v případě, že hra běží přes celou obrazovku (*fullscreen*), neboť pokud je k PC připojeno více monitorů může hráč myší opustit monitor, na kterém hru hraje. Metoda `FixMouseBounds` uzamkne myš v zadaném obdélníku, její bezparametrové přetížení myš uzamkne v okně hry. Uvolnit tento zámek lze pomocí metody `UnfixMouseBounds`.

Ve hrách jako *First-Person Shooter* (dále jen FPS) nebo leteckých simulátorech chceme, aby pohled kamery reagoval na pohyb myši. Avšak okno hry není nekonečné a časem bychom se dostali na jeho okraj a už bychom nemohli pokračovat dále v pohybu. Řešení jsou dvě, ale v praxi se používá jen jedno. První řešení při kontaktu myši s okrajem přesune kurzor na opačnou stranu obrazovky. Toto řešení se však skoro vůbec nepoužívá. Druhé, které implementuje náš engine, zafixuje myš na určitý bod obrazovky (běžně střed) a pro výpočet velikosti pohybu kamery využívá odchylky od tohoto bodu, které vznikají pohybem myši. Zafixovat myš na určitý bod lze pomocí metody `FixMouse`, případně `FixMouseAtCurrentPosition`. Bezparametrová verze metody `FixMouse` zafixuje pozici myši na střed obrazovky. Odchylky od fixního bodu lze získávat voláním `GetMoveFromFix`.

6.2.4 GUIManager

GUIManager

```
Control HoveredControl;
Control MouseCaptured;
float ToolTipDelay;
string ToolTipText;
bool UseShortcuts;
List<Shortcut> Shortcuts;

void CaptureMouse(Control c);
void ReleaseMouse();
Control GetControlById(int id);

Button AddButton(ButtonParams parms,
Control parent);
Image AddImage(ImageParams parms,
Control parent);

A další…
```

Pro prvky jakéhokoli GUI je charakteristické, že mají vždy velké množství parametrů. Některé parametry jsou důležitější než ostatní a některé chceme často ponechat ve výchozím nastavení. Pokud bychom k vytváření prvků použili metody, jejichž parametry by odpovídaly přesně parametrům prvku, museli bychom vždy uvést i ty, které pro nás v současné chvíli nejsou důležité. Případně bychom museli vytvořit mnoho přetížení jedné funkce a počítat se všemi eventualitami. Což by třeba u *Labelu*, s jeho 9 parametry, znamenalo vytvořit 510 přetížení (podle vzorce 6.1). Některá tato přetížení by samozřejmě nešla vytvořit z důvodu stejné typové signatury, avšak i po této redukci by jich zůstalo velmi mnoho. Toto je samozřejmě extrémní případ a výhodnější by bylo pokaždé vyplnit i nedůležité parametry. Chtěli jsme tím jen ukázat, že toto řešení není z hlediska komfortu a časové náročnosti myslitelné.

Pro každý typ GUI prvku jsme tedy vytvořili strukturu, která reprezentuje jeho parametry. Při jejím vytváření jsou v konstruktoru nastaveny všechny parametry na výchozí hodnoty. Vývojáři pak jen stačí změnit ty, které se od těch výchozích mají lišit. Po vytvoření parametrové struktury, je pro přidání prvku do scény nutné zavolat odpovídající metodu, která zkontroluje správnost hodnot ve struktuře a následně prvek vytvoří. Tyto metody vždy začínají slovem `Add` a pokračují názvem odpovídajícího prvku. Vracejí vytvořený a inicializovaný prvek, který si může vývojář uchovat a později jeho parametry měnit za běhu. Kromě obvyklých (`Position`, `Rotation`, `Scale`, `Size`, `Enabled` atd.) jsou zde i méně obvyklé, např. `Tag`, který může obsahovat jakýkoli objekt, který si vývojář zvolí. Jeho využití je zcela v jeho rukou. Dalšími položkami jsou delegáti metod, které zpracovávají události spojené s akcemi uživatele. Například pokud uživatel klikne na prvek, je zavolána metoda, jejíž delegát je uložen v položce `MouseDownHandler`.

ControlParams

```
Vector2 Position;
Vector2 RotationOrigin;
float Rotation;
Vector2 Scale;
Vector2 Size;
float Depth;
int Id;
object Tag;
bool Enabled;
bool ShowTooltip;
Tooltip Tooltip;

MouseEventHandler MouseDownHandler;
MouseEventHandler MouseMoveHandler;
MouseEventHandler MouseEnterHandler;
MouseEventHandler MouseLeaveHandler;
MouseEventHandler MouseUpHandler;
MouseEventHandler MouseWheelHandler;
```

$$combinations = \sum_{n=1}^{9} \binom{9}{n} \tag{6.1}$$

Stejně jako u *SceneManageru* jsou i zde prvky uloženy jako stromová struktura. Důvody jsou velmi podobné jako u 3D objektů. Proto má každý prvek uložen svého rodiče a seznam svých potomků. Toto uspořádání umožňuje snadnější tvoření celků z jednotlivých souvisejících prvků GUI.

GUIManager si musí neustále udržovat informaci o tom, který prvek je pod kurzorem myši. Řešení tohoto problému je poměrně triviální, ale lze aplikovat některá vylepšení, která by měla celý proces urychlit. Při každém updatu je rekurzivně procházen celý strom, pozice a rozměry každého prvku jsou porovnány s pozicí myši. V případě, že prvek leží pod myší, je přidán do seznamu. Takto je postupně vytvořen seznam všech prvků pod kurzorem. Ten je následně vzestupně seřazen podle hloubky (od nejbližšího, po nejvzdálenější) a první prvek je uložen do položky `HoveredControl`. V případě, že je seznam prázdný (pod myší není žádný prvek) je `HoveredControl` nastaven na nedefinovanou hodnotu (*null*).

Po zjištění a nastavení `HoveredControl` je již snadné správně zpracovávat události uživatelských vstupů. Všechny akce, které uživatel udělá, jsou vztaženy právě na aktivní prvek pod myší. Například pokud hráč pohne myší, je zavolána metoda, která je reprezentována delegátem v `HoveredControl.MouseMoveHandler`. U delegátů je vždy nutné ověřit, zda nějakou metodu vůbec obsahují. Vývojář ji totiž nemusel definovat, čímž pravděpodobně zamýšlel, že danou akci nechce zpracovávat. Při změně prvku v `HoveredControl` jsou voláni delegáti `MouseEnterHandler` a `MouseLeaveHandler` – na nově aktivovaný prvek je zavolána první jmenovaná a na původní ta druhá.

Zasílání událostí se chová poněkud jinak, pokud je aktivní přesměrování na jeden prvek (důvody pro toto chování jsou popsány v kapitole 6.1.4). Místo aby byly volány metody `HoveredControlu`, jsou volány metody prvku nastaveného v `MouseCaptured`. Ten lze nastavit pomocí metody `CaptureMouse`, která jako parametr přijímá prvek, do kterého se má přesměrovávat. Toto platí, dokud není přesměrování zrušeno voláním metody `ReleaseMouse`.

Následuje krátký popis některých základních a některých zajímavých prvků GUI:

Label

Základní prvek každého GUI. Obsahuje statický text, kterému je možné nastavit různé barvy, font a zarovnání. Celý text nemusí mít stejnou barvu – pomocí značky `<color=r,g,b>` je možné ji měnit jen u částí. Pro hledání značek a získání potřebných parametrů je použita třída `RegExp`, která je standardně obsažena v .NET.

Pro správné zarovnání textu je potřeba znát rozměry konkrétního textového řetězce. Naštěstí má třída `SpriteFont` nestatickou metodu `MeasureString`, která je toto měření schopna provést. Následný výpočet pozice pro požadované zarovnání je již triviální.

Button

Tlačítko může nabývat jednoho ze čtyř stavů – *normal*, *hovered*, *clicked* a *disabled*. Pro každý tento stav může vývojář určit tlačítku jinou podobu. Stav *disabled* lze zjistit přímo z položky `Enabled`, avšak ostatní stavy lze poznat jen porovnáním s odpovídajícími položkami v `GUIManageru`. Například tlačítko je ve stavu *hovered* jen tehdy, pokud je nastaven v položce `GUIManager.HoveredControl`.

Jako jediný prvek GUI může mít tlačítko přiřazenou klávesovou zkratku (*shortcut*). Aby nebylo potřeba při každém stisku klávesy procházet všechna tlačítka a zjišťovat, zda má některé tlačítko na klávesu reagovat, je už při přiřazení klávesy zaregistrována do zkratek, které udržuje *GUIManager*. Zkratku lze kdykoli zrušit nebo se zruší sama, pokud je prvek odebrán ze scény.

NinePatch

Jde o libovolně roztažitelný prvek, který je složen z 9 menších obrázků, kde každý reprezentuje jeden ze čtyř rohů, čtyř hran nebo středu (viz obrázek 6.4). Rozměry rohů nejsou měněny, hrany jsou roztaženy nebo opakovány jen podél jedné ose (vertikální nebo horizontální) a střed je vždy roztažen případně opakován v obou těchto osách.

Obrázek 6.4: Zvětšená ukázka rozdělení obrázku na NinePatch

6.2.5 CutsceneManager

V první fázi bylo nutné definovat, v jakém formátu budou jednotlivé cutscény uloženy. Kvůli vestavěné podpoře v .NET jsme se rozhodli pro XML. Pomocí `XmlTextReaderu` je načítání souborů velmi snadné a ušetří mnoho práce, která by vznikla při zavedení svého proprietárního formátu. Při načítání XML lze do jisté omezené míry kontrolovat jeho správnost, avšak pro naprostou přesnost bychom museli vytvořit *Document Type Definition* (dále jen DTD), která by definovala uspořádání všech entit a jejich atributy. Z časových důvodů jsme zvolili kompromis a kontrolujeme XML jen na nepřítomnost důležitých značek.

CutsceneManager

```
void LoadCutscene(string name);
void PlayCutscene(string name);
void StopCutscene(string name);
void ResetCutscene(string name);
void UnloadCutscene(string name);
```

O toto načítání se stará metoda `LoadCutscene`, která jako svůj parametr přijímá název XML souboru s cutscénou. Jak je soubor postupně načítán, je vytvářena stromová struktura tříd `Sequence` a `Act`, která popisuje cutscénu uloženou v souboru. Jako vstupní bod (kořen) každé této struktury slouží třída `Cutscene` (viz obrázek 6.3). Každá z těchto tříd má metodu `Load`, která načítá data a vytváří z nich své tělo. Takovéto rekurzivní využití objektů velmi zpřehlednilo zdrojový kód. Implementace veškerého načítání do jedné metody, např. `CutsceneManager.LoadCutscene`, by vedla ke zmatkům a nepřehlednosti. Vždy po načtení celého aktu, respektive sekvence se zavolá metoda `AddAct`, resp. `AddSequence` a tím se přiřadí svému rodiči.

Cutscene

```
void Load(XmlTextReader reader);
void AddSequence(Sequence seq);
bool HasNextSequence();
Sequence GetNextSequence();
void Unload();
void Reset();
```

Metoda `PlayCutscene` přijímá také jako parametr název XML souboru s cutscénou. Soubor však neotevírá a jen ověří, zda již byl načten metodou `LoadCutscene`. Tímto rozdělením jsme umožnili vývojáři oddělit časově náročnější načítání souboru od spuštění scény. Nevzniká tak žádná prodleva mezi zavoláním metody `PlayCutscene` a zahájením cutscény.

Sequence

```
void Load(XmlTextReader reader);
void AddAct(Act act);
bool HasNextAct();
Sequence GetNextAct();
void Unload();
void Reset();
```

Při spuštění jsou cutscéna a její první sekvence nastaveny jako aktivní. Pokud se jedná o paralelní sekvenci, jsou všechny její akty označeny jako aktivní. V opačném případě (sekvence je lineární) je takto označen jen první akt.

Act

```
bool Finished;

void Load(XmlTextReader reader);
void Unload();
void Reset();
```

V každém updatu cutscény jsou aktualizovány všechny aktivní akty. Ty přestávají být aktivní až ve chvíli, kdy splnily svůj účel, což indikuje položka `Finished` – např. přesun kamery byl dokončen, hráč splnil zadaný úkol atd. Jakmile nezbude žádný aktivní akt ke zpracování, je pomocí metody `HasNextAct` ověřeno, zda má aktivní sekvence ještě nějaké další akty, které nebyly vykonány. Pokud má, je následující získán voláním metody `GetNextAct` a je nastaven jako aktivní. Toto se opakuje, dokud má sekvence alespoň jeden nezpracovaný akt. Poté je, stejně jako u aktů, ověřeno, zda existuje ještě další sekvence, a pokud ano, je nastavena jako aktivní. Tentokrát však pomocí

metod `HasNextSequence` a `GetNextSequence`. Tento postup se opakuje, dokud v cutscéně existuje nějaká nezpracovaná sekvence.

Cutscénu je možné přerušit, znovuspustit nebo uvolnit z paměti metodami `StopCutscene`, `ResetCutscene` a `UnloadCutscene`. V posledních dvou případech jsou rekurzivně volány odpovídající metody sekvencí a aktů dané cutscény.

6.2.6 Camera

Ve scéně může být v jeden okamžik umístěn libovolný počet kamer, avšak jen jedna z nich může být aktivní. Složitější případ více aktivních kamer (např. hlavní obrazovka zobrazuje pohled na silnici před závodním vozem a menší obrazovka představuje zadní zrcátko) je nad rámec této diplomové práce. Implementovali jsme tedy položku `ActiveCamera`. Pro usnadnění přístupu k aktivní kameře odkudkoli z enginu nebo hry jsme ji vytvořili jako statickou. První vytvořená kamera je automaticky nastavena jako aktivní. Pokud chce vývojář aktivní kameru změnit, stačí, aby do položky `ActiveCamera` přiřadil jinou instanci a vše ostatní za něj udělá engine.

Camera

```
static Camera ActiveCamera;
Vector3 Rotation;
Vector3 Position;
BoundingFrustum ViewFrustum;
Vector3 Forward;
CameraControls Controls;
float MaxSpeed;
float MaxRotationSpeed;
Vector3 Velocity;
float RotationSpeed;
bool MoveEnabled;
Matrix Projection;
Matrix View;

Ray GetRayFromScreen(Point screenPos);
```

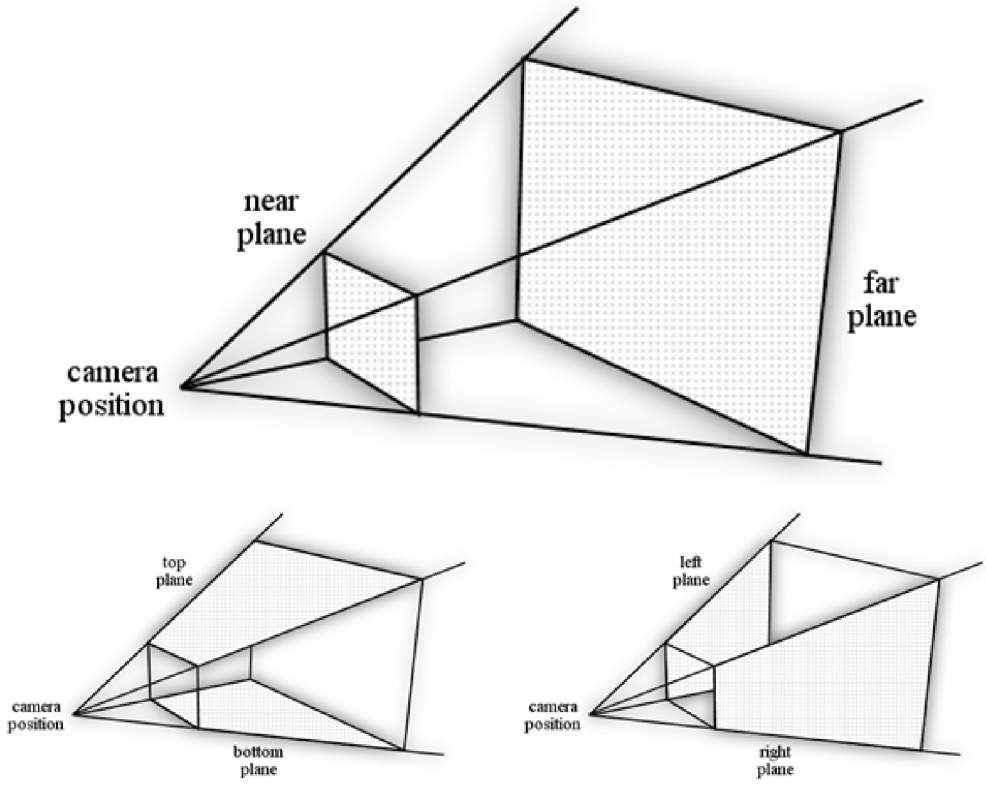
Při vytváření kamery je automaticky vytvořena projekční matice, neboť ta se na rozdíl od pohledové matice nemění v závislosti na pozici a rotaci kamery. K jejímu vytvoření je nutné uvést úhel zorného pole (*field of view*), poměr stran (*aspect ratio*) a vzdálenost přední a zadní ořezové plochy (*near/far clipping plane*). V našem enginu jsme se rozhodli neumožnit vývojáři tyto hodnoty měnit. Jsou použity hodnoty, které jsou běžné a které by měly být vyhovující pro většinu typů her. Pokud však přece jenom bude tuto změnu vyžadovat, může si tuto funkcionalitu velmi snadno doplnit. Aktuální projekční a pohledovou matici může vývojář získat dotazem na položky `Projection` a `View`.

Vektor rovnoběžný s pohledem kamery a pohledové těleso (viz obrázek 6.5) jsou počítány v každém updatu, neboť jsou závislé na pozici a rotaci kamery. Vektor rovnoběžný s pohledem kamery (*forward vector*) lze vypočítat následujícím vzorcem:

$$forward = \begin{pmatrix} \sin(-rotation.y) \cdot \cos(rotation.x) \\ \sin(rotation.x) \\ \cos(-rotation.y) \cdot \cos(rotation.x) \end{pmatrix} \tag{6.2}$$

Jedna možnost, jak získat pohledové těleso, je vypočítat průsečíky přední a zadní ořezové plochy s paprsky, které procházejí rohy obrazovky a objektivem kamery. Poté pomocí těchto bodů vyjádřit plochy, které těleso definují. Toto je však poměrně výpočetně náročné a především zbytečné, neboť XNA nám nabízí daleko jednodušší řešení. Obsahuje totiž třídu `BoundingFrustum`, která ve svém jediném konstruktoru přijímá jako parametr tzv. kombinovanou matici - součin pohledové a

projekční matice (v tomto pořadí). Takto vzniklý objekt obsahuje položky reprezentující všech šest ploch, které pohledové těleso tvoří (`Top`, `Bottom`, `Left`, `Right`, `Near`, `Far`). Také implementuje metody, které zjišťují, zda obsahuje nebo protíná některá další obalová tělesa (*bounding volumes*). Vývojář může pohledové těleso získat z položky `ViewFrustum`.



Obrázek 6.5: Pohledové těleso s označenými hlavními plochami

Často je potřeba vybrat pomocí myši nějaký objekt před kamerou. Tomuto úkonu se anglicky říká *picking*. V trojrozměrném prostoru se většinou provádí tak, že se vypočítá paprsek (polopřímka), který začíná v objektivu kamery a prochází bodem, na který hráč klikl. Poté se zkontroluje, zda paprsek protíná některý objekt nebo jeho obalové těleso. Abychom však tento bod získali, musíme provést tzv. *unprojection*, což je převod obrazových souřadnic (*screen space coordinates*) do souřadnic v trojrozměrném prostoru (*world space coordinates*). Pak již jen stačí od takto získaného bodu odečíst pozici kamery a výsledný vektor normalizovat. V našem enginu lze tento paprsek získat zavoláním metody `GetRayFromScreen`.

CameraControls

```
public Keys Forward;
public Keys Backward;
public Keys Left;
public Keys Right;
```

Aby bylo možné ovládat pohyb kamery pomocí klávesnice, implementovali jsme abstraktní třídu `CameraControls`. Každé nové kameře je přiřazena instance této třídy inicializovaná výchozími klávesami. Pokud se vývojář rozhodne, že chce kameru ovládat jinými klávesami, může vytvořit vlastní instanci, inicializovat ji svými klávesami a přiřadit ji kameře jako položku `Controls`. Výhoda toto přístupu je, že není potřeba pro každou klávesu vytvářet v kameře položku. Pro základní kameru, kde jsou jen čtyři klávesy (vpřed, vzad, doleva a doprava) by to nebyl tak velký

problém, ale vývojář může chtít vytvořit kameru, která těchto kláves má daleko více. Takto mu stačí vytvořit potomka třídy `CameraControls` a implementovat si vlastí inicializaci a položky odpovídající jednotlivým klávesám.

Free camera

Pro testování a ladění je výhodné, aby kamera nebyla omezena žádnými limity a vývojář se tak mohl podívat na kteroukoli část scény z kteréhokoli úhlu. Kamera má i novou dvojici tlačítek, která ovládá její pohyb nahoru a dolů, což dále zvyšuje její stupeň volnosti.

RTS camera

Jako rozšíření standardní kamery jsme implementovali kameru, jejíž primární použití je v RTS hrách. Nejdůležitější věcí, která přibyla, je propojení s terénem, jelikož kamera potřebuje při maximálním přiblížení kopírovat terén a nesmí se dostat pod něj. Terén implementuje metodu `GetTerrainHeight`, které stačí předat pozici kamery nad terénem, a ona vrátí maximální výšku, do které se kamera může přiblížit.

CameraRTS

```
Vector3 MinBounds;
Vector3 MaxBounds;
ITerrain Terrain;
Vector2 Position2D;

void LookAt(Vector2 target);
```

V RTS nás velmi často zajímá pozice nějakého objektu (v tomto případě kamery), avšak nezajímá nás jak je vysoko. Pro tento případ je zde položka `Position2D`, která reprezentuje pozici kamery, ovšem bez třetí složky – výšky.

Jednotlivé herní úrovně v RTS hrách jsou často čtvercového nebo obdélníkového půdorysu. Za okraje toho prostoru by se kamera neměla nikdy dostat. Z toho důvodu jsme do tohoto typu kamery přidali omezení reprezentované dvěma položkami `MinBounds` a `MaxBounds`, které označují nejzazší body, kam se kamera může dostat.

Do ovládání kamery přibylo tlačítko pro rotaci kamery a tlačítka, která ovládají přibližování a oddalování terénu.

6.2.7 Cursor

Při vývoji jsme zkusili implementovat dvě varianty této služby a porovnat jejich výhody a nevýhody. Varianta B se ukázala jako výrazně lepší řešení, a proto byla zařazena do našeho enginu.

Varianta A - kurzor jako sprite

Varianta spočívá ve vykreslování *spritu* (malý dvojrozměrný obrázek), který představuje kurzor, na pozici myši. XNA má velmi dobrou podporu *spritů*, a proto se tato varianta sama nabízí. Má však poměrně velké množství nevýhod, a tudíž jsme se rozhodli ji neimplementovat.

Výhody:

- Transformace (rotace, změna rozměrů…) jsou možné přímo za běhu programu.

Nevýhody:

- Při nižším FPS je pohyb kurzoru trhaný nebo zpomalený, neboť nastavování jeho pozice probíhá ve stejném vlákně jako zbytek hry. Tento problém by samozřejmě šlo eliminovat přesunem na samostatné vlákno, avšak to by bylo zbytečně časově náročné.

- Potřeba definovat další třídu pro reprezentaci různých kurzorů, která by uchovávala všechny parametry kurzoru - rozměry, animaci, aktivní bod (anglicky *hotspot*, což je bod na kurzoru, který je použit při kliku; tento bod může být u každého kurzoru jiný, viz obrázek 6.6)

Obrázek 6.6: Ukázky různých hotspotů u různých kurzorů

Varianta B - systémový kurzor

Tato varianta využívá třídu `Cursor`, která je součástí .NET a dokáže přímo manipulovat se systémovým kurzorem Windows. Její použití je velmi snadné a vytvořit kurzor ze souboru je otázka jednoho řádku kódu.

SystemCursor

```
Cursor Cursor;
bool Visible;

Cursor LoadCursor(string file);
void Hide();
void Show();
```

Má ovšem jednu zásadní nevýhodu a to, že konstruktor, který jako parametr přijímá cestu ke kurzoru, nedokáže načíst soubory ve formátu *ani* (animovaný kurzor) [39].Naštěstí existuje alternativní řešení - pomocí *PInvoke* importovat *WinAPI* funkci `LoadCursorFromFile`, která jako svůj jediný parametr přijímá cestu ke kurzoru a po načtení vrací ukazatel do paměti. S tímto ukazatelem už si konstruktor třídy `Cursor` poradí a bez problémů načte animovaný kurzor.

Naše služba tuto funkcionalitu zapouzdřuje a nepřenáší toto nepohodlí na vývojáře. Stačí jen kurzory načíst pomocí metody `LoadCursor` a položku `Cursor` nastavit na jednu z vrácených hodnot. Metody `Hide` a `Show` slouží k jednoduchému nastavení vlastnosti `Visible`.

Výhody:

- Poté, co jsme zjistili, jak obejít omezení při načítání *ani* kurzorů, byla implementace daleko jednodušší a rychlejší než u varianty A.
- Pro pohyb kurzoru je využíváno zachytávání zpráv, které je přímo vestavěné ve formulářích systému Windows a které navíc běží na zvláštním vlákně. Tím se odstranil problém se zpožděním pohybu, jenž vzniká při nízkém FPS.
- Existuje velké množství bezplatných, ale přesto velmi mocných, aplikací pro tvorbu *ani* kurzorů. Lze takto snadno oddělit vývoj hry a vytváření grafického designu kurzorů.
- Není potřeba implementovat žádné další třídy. O animaci, rozměry, hotspot a další se stará přímo operační systém.

Nevýhody:

- Nelze za běhu aplikace měnit rozměry a rotaci. To však lze částečně nahradit vytvořením více kurzorů s různou animací.

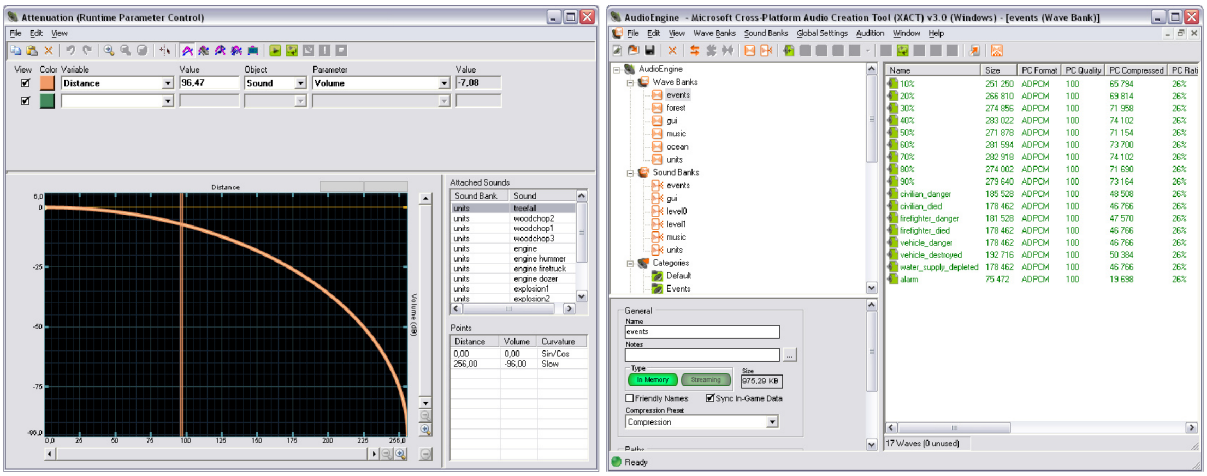
6.2.8 Audio

Pro jednoduché přehrávání typu *fire-and-forget*, které se hodí jen do velmi jednoduchých her, obsahuje XNA třídu `SoundEffect`. Vytváří se načtením zvuku z obsahu hry. Poté již jen stačí zavolat metodu `Play`, která se postará o přehrání zvuku. Tímto však nad ním vývojář ztrácí veškerou kontrolu a nemůže jej zastavit, posunout nebo změnit hlasitost.

Tato metoda tedy není vhodná pro větší projekty, kde vývojář potřebuje mít absolutní kontrolu nad parametry zvuku v reálném čase. Naštěstí XNA podporuje i komplexnější práci se zvukem pomocí utility *Cross-platform Audio Creation Tool* (XACT). Tu Microsoft navrhl, aby oddělila práci programátora od práce zvukaře. XACT umožňuje snadnou organizaci zvuků do tzv. *wave bank*, což je v podstatě jeden soubor obsahující více zvuků (podporované formáty jsou wav, aiff a wma). *Wave banky* lze komprimovat metodou ADPCM na Windows a XMA na Xbox 360, jejich kompresní poměr je však poměrně nízký, jen 4:1.

Další úrovní organizace jsou *sound banky*, které odkazují na zvuky v jedné nebo více *wave bancích*. K těmto odkazům přidávají další doplňující informace o zvuku, jako je například hlasitost nebo výška (*pitch*).

Posledním stupněm dělení jsou *cues*. Názvy těchto front jsou rozhraním mezi XACTem a hrou (jsou používány přímo ve zdrojovém kódu). Může obsahovat více zvuků ze *sound banky* a také je možné určit pořadí, v jakém se budou přehrávat. Například *cue* s názvem „kaboom“ může obsahovat čtyři mírně rozdílné zvuky výbuchu, jako pořadí přehrávání zvolíme náhodu bez opakování stejného zvuku dvakrát po sobě. Nyní každé spuštění *cue* „kaboom“ přehraje jeden náhodný zvuk ze seznamu. Tato náhoda dodá hře na větší realističnosti, než kdyby byl jeden zvuk přehráván stále dokola.



Obrázek 6.7: Ukázka aplikace XACT

XNA samozřejmě implementuje API pro práci se soubory vytvořenými XACTem. Toto rozhraní má však pár omezení, která je možné překonat jen implementací vlastní obálky. Jedno z těchto omezení spočívá v tom, že nejsme schopni vytvořit *cue* tak, aby byla přehrávána stále dokola ve smyčce. Nekonečné přehrávání lze nastavit jen u konkrétního zvuku dané *cue*, což není přesně to, co potřebujeme. Představme si kácení lesa, přičemž vývojář chce, aby sekyra při dopadu udělala pokaždé trochu jiný zvuk. Zvukař tedy vytvoří *cue* se zvuky sekyry. Vývojář však nemá možnost přehrávat tuto *cue* ve smyčce, dokud je les kácen.

Nezbývá než při každé aktualizaci kontrolovat, zda již *cue* skončila, a v případě, že ano, ji znovu spustit. Toto nepohodlí bylo důvodem, proč jsme vytvořili komponentu *Audio*, která se o tyto kontroly měla starat bez přílišné náročnosti pro vývojáře. Prvním krokem je pomocí metod `AddWaveBank` a `AddSoundBank` přidat potřebné banky. Aby nemusel vývojář při dalším odkazování na tyto banky neustále zadávat kompletní cestu k souboru, může si bank pojmenovat nějakým krátkým výstižným jménem. Pak již jen stačí zavolat metodu `PlayCue` a předat ji, ve kterém *sound banku* se nachází *cue*, kterou chceme přehrát, její jméno a zda se má přehrávat ve smyčce. Varianta `PlayCue3D` navíc přijímá pozici odkud má zvuk vycházet a vytváří tak prostorový zvuk.

Audio

```
SoundBank AddSoundBank(string name,
string filename);
WaveBank AddWaveBank(string name,
string filename);
SoundCue PlayCue(string soundBank,
string cue, bool repeat);
SoundCue3D PlayCue3D(string soundBank,
string cue, IPosition pos, bool repeat);

void Pause();
void Resume();
```

6.2.9 Helpers

I když je XNA velmi mocný nástroj pro tvorbu počítačových her, nachází se v něm pár nelogičností a nedodělků – jako v každém takto rozsáhlém projektu. Naštěstí C# (a také Visual Basic) poskytuje možnost vytvořit takzvané rozšiřující metody (*extension methods*), což umožňuje přidat metodu již existujícímu typu bez nutnosti vytvářet zděděný typ, rekompilovat nebo jinak měnit původní typ. Z pohledu vývojáře není ve volání obvyklých a rozšiřujících metod žádný rozdíl [37].

Následuje pár příkladů těchto metod, které jsme v našem enginu implementovali, aby vývojáři usnadnily práci.

Převod Vector2 na Vector3 a naopak

Tento převod je poměrně často potřeba (zvláště u RTS) a v XNA neexistuje žádný snadný a elegantní způsob, jak to provést. Proto jsme pro tříprvkový vektor vytvořili metodu `ToVector2` a pro dvouprvkový metodu `ToVector3`.

Ukázka implementace první zmíněné metody. Je zde uvedena, neboť je tento způsob rozšiřování málo známy, a přitom je velmi jednoduchý a účinný. Povšimněte si především klíčového slova `this`, které určuje, jaký typ budeme rozšiřovat.

```
public static Vector2 ToVector2(this Vector3 v)
{
     return new Vector2(v.X, v.Z);
}
```

Zdrojový kód 6.1: Implementace rozšiřující metody ToVector2

Přidání bodu do BoundingBoxu

Takováto základní operace asi vývojářům XNA unikla a nám nezbylo nic jiného, než že jsme si ji museli vytvořit sami. Opět následuje ukázková implementace (ze stejných důvodů jako výše).

```
public static BoundingBox AddPoint(this BoundingBox bb, Vector3 v)
{
    bb.Min = Vector3.Min(bb.Min, v);
    bb.Max = Vector3.Max(bb.Max, v);

    return bb;
}
```

Zdrojový kód 6.2: Implementace rozšiřující metody AddPoint

7 Tvorba hry

Praktickým výsledkem této diplomové práce je RTS **FireFighters: Whatever it takes**, která byla přihlášena do soutěže **Imagine Cup**, pořádanou společností Microsoft a to konkrétně do kategorie game design. V době psaní této práce již **FireFighters** postoupili do semifinále mezinárodního kola a vyhráli první místo v národním kole. Tato kapitola popisuje návrh a implementaci hry, která je vytvořena na enginu popsaném výše.

7.1 Návrh hry

Jak již bylo zmíněno výše, byla tato hra zapsána do soutěže Imagine Cup, jejíž motto a zároveň zadání zní "*Imagine a world where technology helps solve the toughest problems*". Čili hra by měla šířit povědomí o jednom z tzv. *Millennium Development Goals* (dále jen MDG) a snažit se u hráče zvýšit zájem o řešení tohoto problému.

Z těchto důvodů jsme navrhli hru, kde hráč koordinuje jednotky hasičů a snaží se rychle a efektivně uhasit lesní požár a tím zachránit co možná největší plochu lesa a životy ohrožených lidí. Zaměřuje se tedy na MDG číslo 7 - *Ensure Environmental Sustainability*. Logo tohoto MDG je na obrázku vpravo.

7.1.1 Reprezentace lesa

V první fázi bylo potřeba navrhnout způsob vytváření a uložení lesa. Chtěli jsme, aby les splňoval tyto požadavky:

- Potřebovali jsme, aby bylo možné vytvořit jakýkoli les, přesně na požadované místo terénu.
- Stromy musí být rozmístěny realisticky, čili nesmí být příliš blízko sebe, natož aby se jejich modely překrývaly, a u okrajů lesa by měly být menší a méně husté.
- Les musí být v dané misi při každém spuštění stejný, takže nepřichází v úvahu náhodné generování až při spuštění mise. Vše musí být vypočítáno předem a nějakým způsobem uloženo.

7.1.2 Šíření ohně

Vytvořit šíření ohně na základě skutečných fyzikálních modelů není kvůli jejich náročnosti dost dobře možné. V případě, že by se nám to přece jenom podařilo, bylo by řešení všech souvisejících diferenciálních rovnic velmi výkonnostně a paměťově náročné. Toto si samozřejmě ve hře nemůžeme dovolit, a proto jsme navrhli jednodušší variantu, která je však stále velmi realistická. Není evidentně tak přesná, jako fyzikální model, ale pro naši hru je více než dostačující.

Návrh počítá s tím, že si každý strom uchovává informace o svém stavu - velikost, množství paliva, vlhkost a produkované teplo. Množství paliva a vlhkost lze také označit jako zdraví a štít stromu - v analogii k běžným RTS hrám.

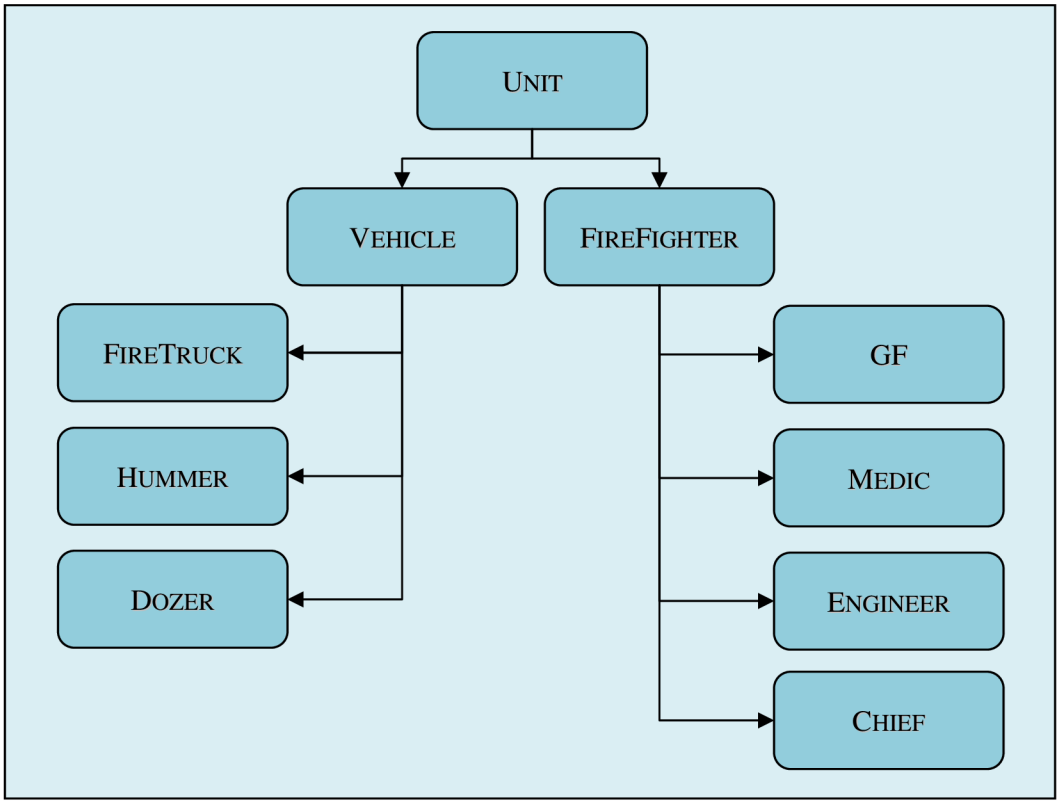
7.1.3 Herní obrazovky

Navrhli jsme také jaké obrazovky (ve smyslu popsaném v kapitole 6.1.2) budeme potřebovat a jakému účelu budou sloužit:

- **VideoScreen** - Zobrazení loga týmu a hry při spuštění hry.
- **MainMenuScreen** - Hlavní menu hry i s nastavením, výběrem misí, titulky a dalšími podobrazovkami.
- **GameplayScreen** - Nejdůležitější obrazovka celé hry. Zde je implementována podstatná část logiky samotné hry. Stará se sloučení jednotlivých komponent hry - jednotky, les, oheň, upgrady a další.
- **GameMenuScreen** - Jedná se menu, které může hráč zobrazit během hraní. Při zobrazení musí zapauzovat hru.
- **OverviewScreen** - Před každou misí si zde hráč nakupuje vybavení pro své hasiče. V druhé části této obrazovky si tyto hasiče také rekrutuje.
- **MissionEndScreen** - Obrazovka, která se objeví na konci každé mise. Informuje hráče o úspěchu, či neúspěchu mise

7.1.4 Jednotky

Jako v každé RTS, i v té naší je velmi důležité mít k dispozici pestrou škálu jednotek. Museli jsme navrhnout, jak tyto jednotky reprezentovat, jak uložit jejich vlastnosti a jak s nimi vykonávat různé akce. Následující obrázek ukazuje navrženou stromovou strukturu jednotek:



Obrázek 7.1: Stromová struktura jednotek

Tato hierarchie je výhodná, neboť každá jednotka musí obsahovat informace o svém zdraví, rychlosti, odolnosti vůči ohni a mít uložen svůj model. Vozidla, na rozdíl od hasičů, mají informaci o volném počtu míst, zda jsou zdroji vody a jestli je řídí autopilot. Hasiči mají také specifické vlastnosti, např. poloměr hašení, síla hašení, nebo zda je v dosahu *Chiefova* povzbuzování. Avšak navíc má každá jednotka další speciální vlastnosti podle svého typu. Například *Firetruck* musí mít informaci o tom, kolik mu v nádrži ještě zbývá vody nebo *Medic* jak rychle a účinně dokáže léčit.

Hasiči (firefighters)

General Firefighter
Nejodolnější a umí nejlépe hasit, ale je pomalý. Jako jediná pěší jednotka umí provádět výsek.

Medic
Nejrychlejší pěší jednotka a umí léčit ostatní hasiče. Jako jediný dokáže evakuovat civilisty.

Engineer
Může opravovat poškozená vozidla a zvyšuje rychlost vozidel, která řídí.

Chief
Zvyšuje schopnosti ostatních hasičů v jeho okolí. Jednou za čas dokáže přivolat letadlo s retardanty.

Vozidla (vehicles)

Firetruck
Nejdůležitější vozidlo, neboť obsahuje hlavní zásoby vody pro ostatní hasiče.

Hummer
Malý ale rychlý, vhodný pro převážení pěších jednotek. Po upgradu může mít vlastní zásobu vody a po krátkou dobu pomáhat s hašením.

Dozer
Velmi pomalá a velmi odolná jednotka sloužící pro rozsáhlý výsek lesní plochy v oblastech blízko požáru.

Zvláštním typem jednotek jsou civilisti. Ti nejsou přímo hráčem ovladatelní, ale je potřeba je evakuovat pomocí medika. Počet přeživších civilistů je jedním z kriterií úspěšného ukončení mise.

7.1.5 Databáze znalostí

Chtěli jsme do hry implementovat databázi znalostí o lesních požárech obecně a o různých způsobech boje s nimi. Také měla obsahovat popis jednotek, misí a herních strategií - tedy herní manuál Proto jsme museli navrhnout způsob reprezentace jednotlivých článků a jejich zařazení do kategorií. Opět jsme se rozhodli pro použití XML, neboť .NET nabízí velmi propracovanou podporu pro tento formát. Rozhodli jsme se rozdělit definici struktury encyklopedie od obsahu jednotlivých článků. Proto vznikly dva souborové formáty - *cyc* a *tex*.

Souborový formát *cyc*

Tento typ reprezentuje celou encyklopedii a její strukturu. Na následující ukázce je encyklopedie, která obsahuje dvě kategorie *Causes* a *Prevention*. První je dále rozdělena na dvě podkategorie *Nature* a *Human*. Podkategorie *Nature* obsahuje dva odkazy typu text (tedy články), každý odkaz musí mít nadpis, v tomto případě *Lightning* a *Fuel*. Druhá podkategorie však neobsahuje článek, ale přímý odkaz na obrázek.

```
<encyclopedia>
    <category title="Causes">
      <category title="Nature">
        <item type="text" title="Lightning" url="lightning.tex" />
        <item type="text" title="Fuel type" url="fueltype.tex" />
      </category>
      <category title="Human">
        <item type="image" title="Arson" url="arson.png" />
      </category>
    </category>
    <category title="Prevention">
      <item type="text" title="Prevention" url="prevention.tex" />
    </category>
</encyclopedia>
```

Zdrojový kód 7.1: Ukázka souborového formátu *cyc*

Souborový formát *tex*

Potřebovali jsme navrhnout souborový formát, který by byl schopný dělit článek na odstavce s nepovinným nadpisem a zároveň ke každému odstavci přiřadit libovolné množství obrázků. Obrázky by měly být textem obtékány, ale vývojář musí mít možnost určit alespoň základní zarovnání (doleva, doprava).

```
<textitem>
  <p title="Back burning">
    <img url="controlledburn00" align="right" />
    Suspendisse purus tellus, scelerisque eu interdum nec,
    pellentesque vel justo
  </p>
</textitem>
```

Zdrojový kód 7.2: Ukázka souborového formátu *tex*

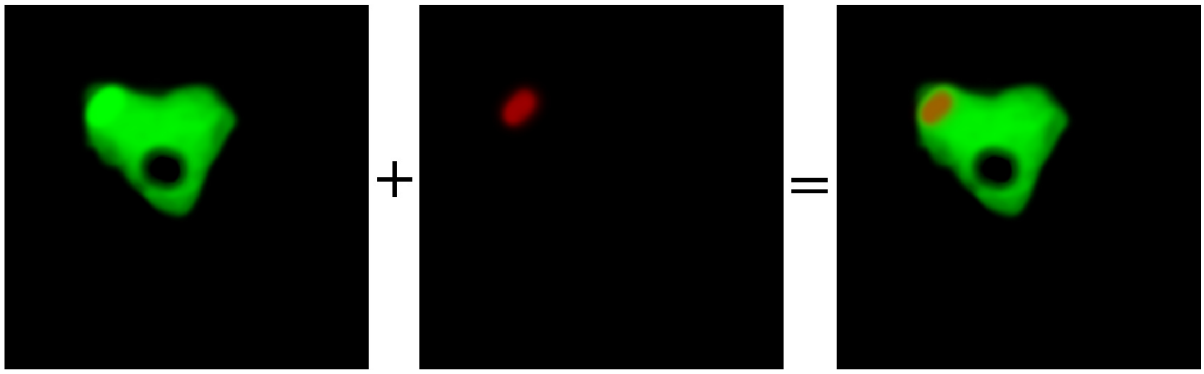
7.2 Implementace hry

V této kapitole se zaměříme především na části, které jsou specifické pro naši hru - reprezentace lesa, šíření ohně. Nebudeme vysvětlovat, jak vytvořit GUI, jak naprogramovat *pathfinding* pomocí algoritmu A* apod. Řešení těchto problémů již zkušený čtenář určitě zná a pro méně zkušeného by k vysvětlení nestačil rozsah této práce.

7.2.1 Reprezentace lesa

Abychom splnili všechny požadavky, které jsme si definovali v kapitole 7.1.1, museli jsme přijít s jednoduchým způsobem, jak označit zalesněnou část mapy. Důležité také bylo nějak určit, jak moc je daná část zalesněna a jak velké stromy v této oblasti rostou. Nejjednodušší se nakonec ukázalo vytvořit obrázek o stejné velikosti, jako výšková mapa, kde hodnota jednoho jeho kanálu (v našem případě zeleného) určovala velikost stromů v daném místě (viz obrázek 7.2). Takto lze snadno určit, kde mají být stromy menší a kde naopak větší. Lehce tak splníme požadavek na to, aby při okrajích lesa byly menší stromy.

Bylo potřeba také určit, kde oheň započne. Tuto informaci jsme přidali do jiného kanálu obrázku (my se rozhodli pro červený). Hodnota určuje, jaký je počáteční žár na tomto místě. Zelený a červený kanál a jejich kombinaci ukazuje následující obrázek.



Obrázek 7.2: Zelený kanál určuje velikost stromů, červený počáteční žár, kombinace je vstupem pro ForestCreator

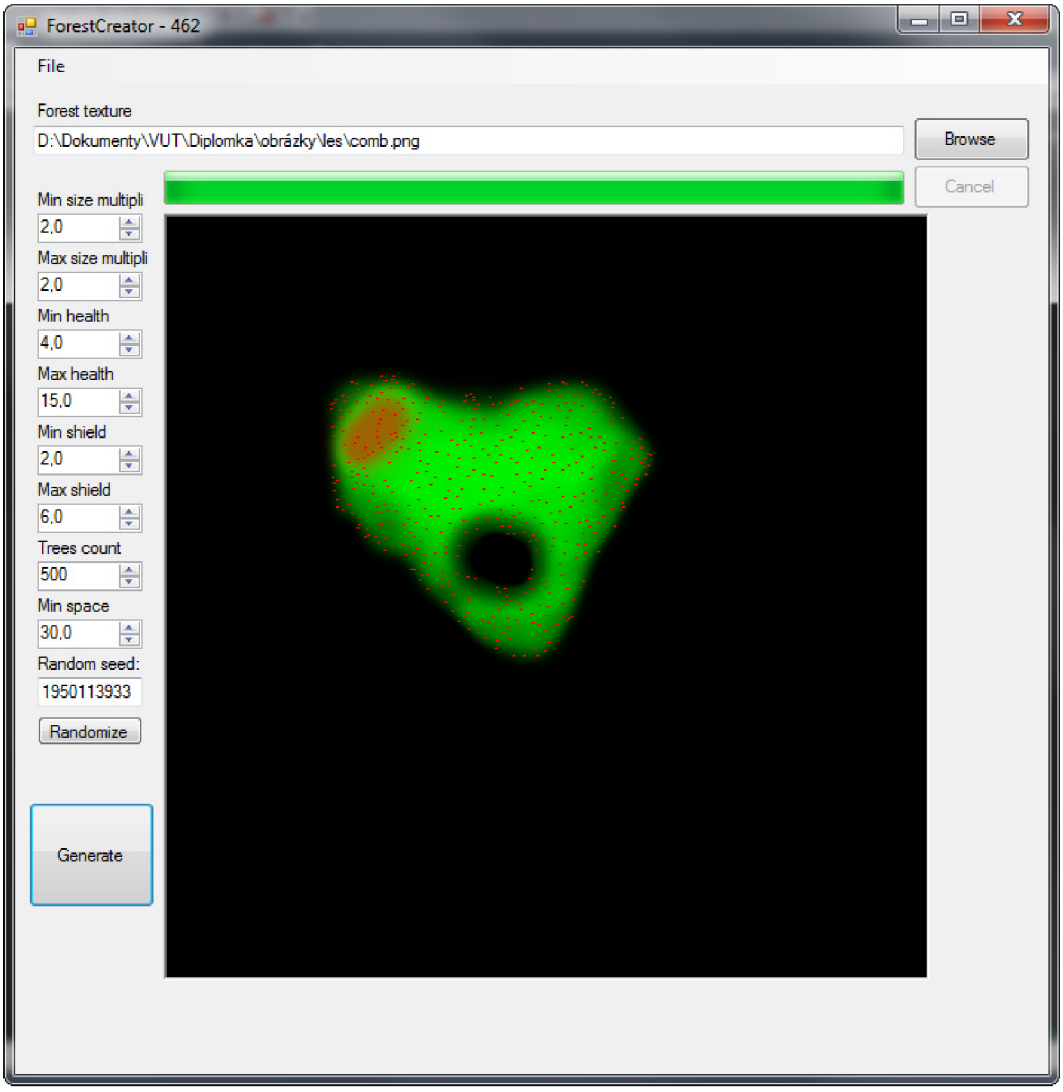
Takto už bychom byli schopni před každou misí vytvořit les na správném místě a se správnou velikostí stromů. Avšak tento les by pokaždé vypadal jinak. To by šlo samozřejmě vyřešit tak, že by měla každá mise uloženo semínko (*seed*) a generátor náhodných čísel by jím byl vždy inicializován. To by však bylo poměrně časově náročné a hráč by tak musel na každé načtení hry dlouho čekat, zvláště na pomalejších strojích.

Vytvořili jsme proto program *ForestCreator*, který vše potřebné předpočítá a uloží v binární podobě do souboru (viz obrázek 7.3). Generování stromů probíhá následovně:

1) Vygeneruj náhodně bod v prostoru obrázku.
2) Zkontroluj, zda není bod příliš blízko některému stromu, pokud ano, jdi zpět na bod 1), jinak pokračuj.

3) Zjisti hodnotu v kanálu s velikostí stromu. Pokud je tato hodnota nižší než práh, pokračuj na bod 1), jinak vytvoř podle této hodnoty odpovídající strom. Přidej jej do seznamu vygenerovaných stromů.
4) Zjisti hodnotu v kanálu s počátečním žárem. Tuto hodnotu nastav stromu.
5) Pokračuj na bod 1) dokud není vygenerován požadovaný počet stromů a nebo již došlo k velkému množství marných pokusů.

Rozpracovaný projekt lze uložit a vrátit se k němu později, vývojář si tak nemusí zapisovat parametry při dolaďování finální podoby lesa.



Obrázek 7.3: Rozhraní programu ForestCreator s vygenerovaným lesem

Soubor vytvořený *ForestCreatorem* je pak při načítání mise otevřen a uloženými daty jsou naplněny třídy reprezentující jednotlivé stromy. Každá mise má seznam modelů stromů a každý strom má normalizovaný rozsah velikosti, při kterém je použit. Například model *bush* má rozsah od 0,0 po 0,6 a model *tree* od 0,5 po 1,0. To znamená, že pro všechna místa, kde je velikost v rozsahu 0,0 až 0,5 budou jen modely *bush*, v rozsahu 0,6 až 1,0 jen *tree* a v překryvu 0,5 až 0,6 bude buď *bush*, nebo *tree*. Tímto lze zajistit odpovídající velikost pro správné modely. Překryvem je pak vytvořen plynulý přechod mezi těmito modely.

7.2.2 Šíření ohně

Pro naši hru bylo velmi důležité, abychom vytvořili realistické a uvěřitelné šíření ohně, které by však zároveň bylo výpočetně nenáročné. Jak již bylo vysvětleno v kapitole 7.1.2, použití skutečných fyzikálních modelů není možné. Proto má každý strom uloženo své palivo, vlhkost, produkovaný žár a žár, který je pod stromem. Pokud strom hoří, tak produkuje žár, který je přímo úměrný množství paliva, které stromu ještě zbývá. Tento žár předává do okolí a jeho intenzita klesá s druhou mocninou vzdálenosti od stromu. Všechny stromy v tomto okolí jsou žárem ovlivněny. Nejdříve začínají vysychat a snižuje se jim vlhkost (štít), teprve až dosáhne tato hodnota nuly, začne strom hořet. Hořením samozřejmě produkuje další žár a ubývá mu palivo (zdraví).

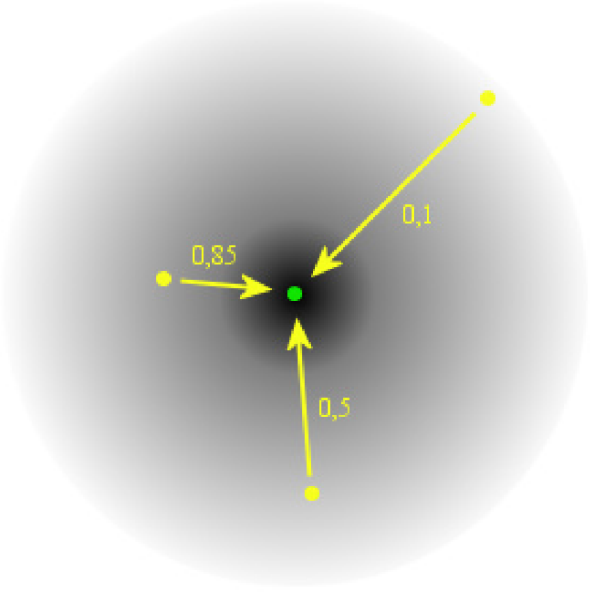
Tree

```
short X, Y;
float BaseHealth;
float Health;
float InitShield;
float Shield;
float Size;
float Heat;
float HeatUnderTree;
List<HeatInfluence> HeatInfluences;
```

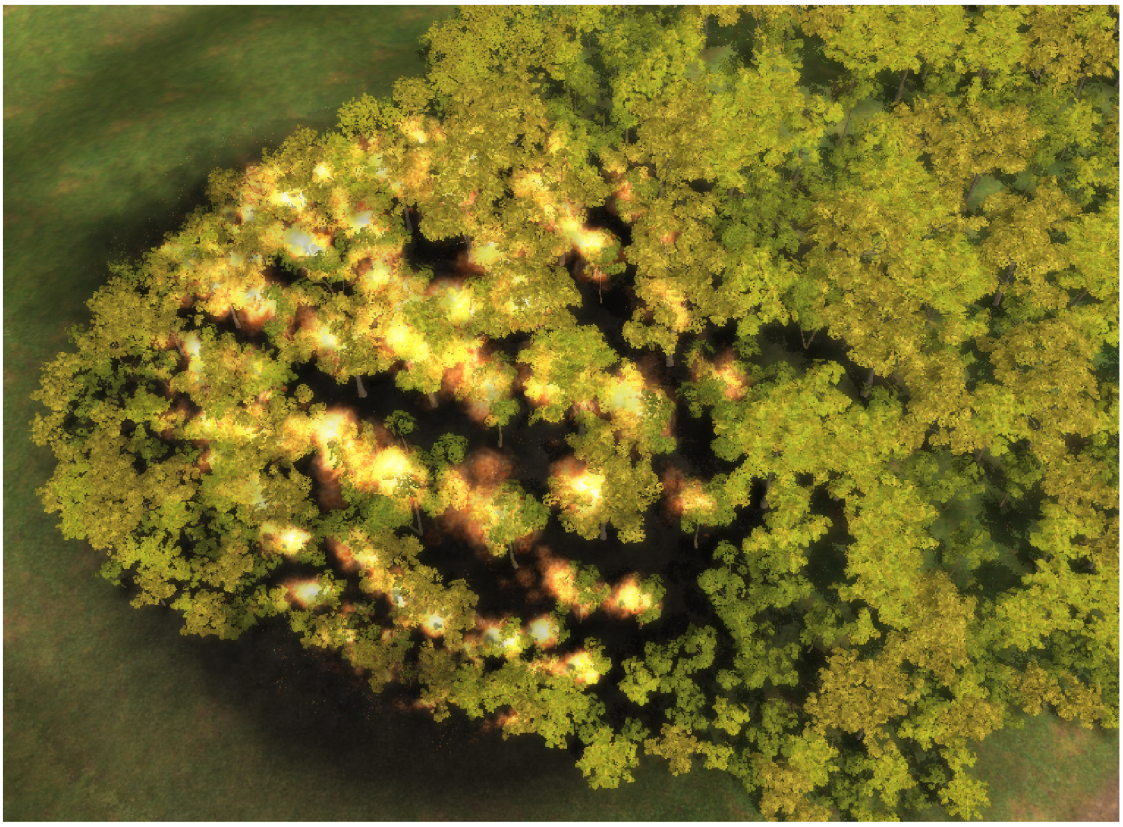
Bylo by velmi nepraktické a pomalé procházet pro každý strom všechny ostatní stromy na mapě. Proto jsme provedli optimalizaci, která vychází z předpokladu, že některé stromy se nemohou navzájem ovlivnit ani tehdy, pokud produkují maximální možný žár. Každý strom má seznam dalších stromů, které jej ovlivnit mohou. Tento seznam je vytvořen už při načítání mise, což je možné díky tomu, že stromy v průběhu času nemění svou pozici. Každá položka tohoto seznamu obsahuje kromě ovlivňujícího stromu i koeficient určující míru ovlivnění (viz obrázek 7.4), což odstranilo nutnost opakovaného výpočtu vzdálenosti mezi těmito dvěma stromy.

HeatInfluence

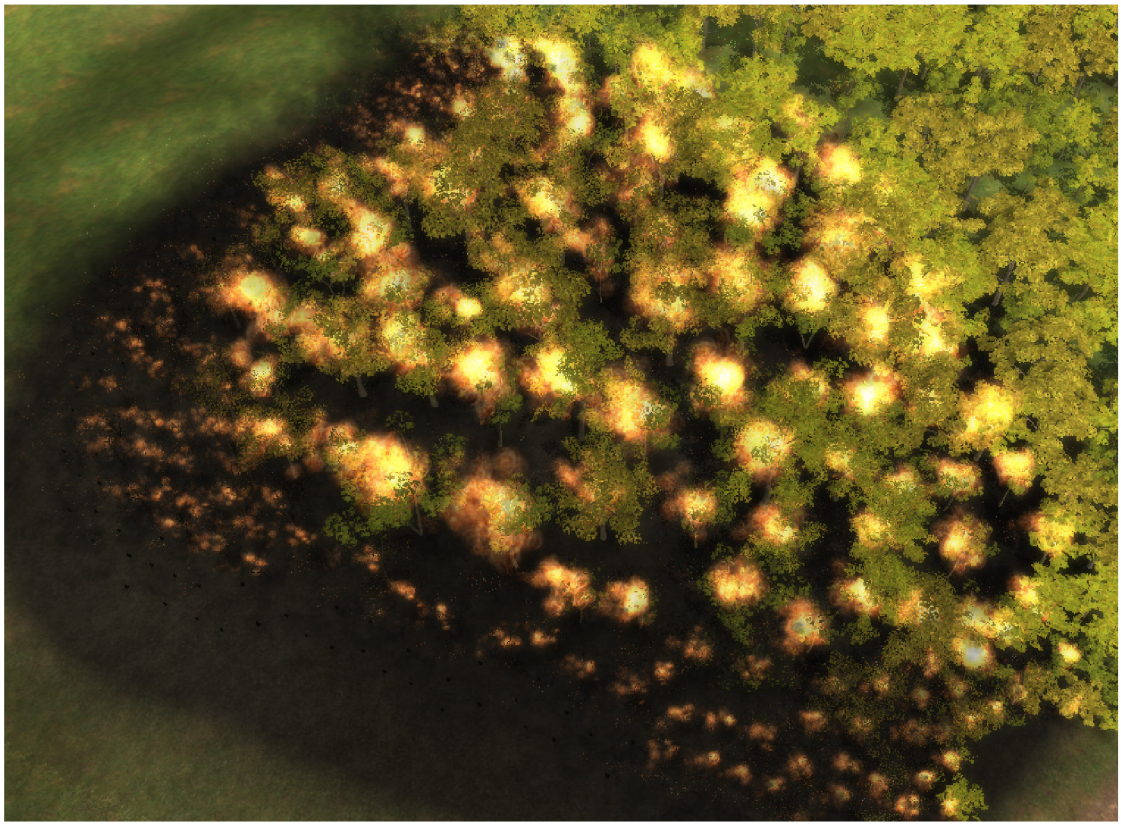
```
Tree Tree;
float Influence;
```



Obrázek 7.4: Míra ovlivnění stromu (zelený) stromy v okolí (žluté)



Obrázek 7.5: Šíření ohně - ranější fáze



Obrázek 7.6: Šíření ohně - pozdější fáze

7.2.3 Jednotky a jejich akce

Každá jednotka je reprezentována vlastní třídou, která obsahuje vlastnosti specifické jen pro ni. Vlastnosti, které jsou stejné pro stejnou skupinu jednotek (vozidla nebo hasiči) jsou implementovány v obecných třídách `Vehicle` a `Firefighter`. Tyto dvě třídy jsou ještě zděděny ze třídy `Unit`, ve které jsou nejobecnější vlastnosti, jež jsou společné pro všechny jednotky.

Unit

```
float BaseHealth;
float Health;
float Speed;
float CurrentMoveSpeed;
float RotationSpeed;
bool IsDead;
float FireResistance;
Task CurrentTask;
Queue<Task> NextTasks;
```

Pokud hráč označí jednotku, zpřístupní tak seznam akcí, které s ní může provádět - pohyb, hašení, zastavení akce a další. Každá zvolená akce je přidána jednotce do jejího seznamu akcí - `NextTasks`. Pokud je seznam prázdný, je ihned nastavena jako aktivní a při prvním updatu je hned aktualizována. Aktivní akce je uchovávána v položce `CurrentTask`. Akce může skončit úspěchem, nebo neúspěchem, např. jednotka nemůže na dané místo dojít, v cisterně už není dostatek vody k hašení apod. V případě, že skončí úspěchem, je vybrána další akce ze seznamu a je nastavena jako aktivní. Avšak pokud nastane neúspěch, jsou všechny následující akce přerušeny.

Většina akci je složena z více podakcí. Často to bývá pohyb následovaný jinou specifičtější akcí. Kdyby musel hráč s jednotkou pokaždé dojít na místo, kde akci může vykonat, velmi by se snížila hratelnost hry. Tedy například pokud hráč jednotce určí, že by měla začít hasit na druhé straně mapy, tak sama dojde na nejbližší možné místo, odkud může akci započít a následně ji započne.

Obrázek 7.7: Screenshot ze hry Firefighters ukazující všechny typy jednotek

8 Závěr

Cílem této diplomové práce bylo čtenáře seznámit s nedávno vzniklým fenoménem nezávislého vývoje počítačových her, který dnes zažívá rapidní rozvoj. Stručně čtenáři popsala historii počítačových her jako celku, vysvětlila základní historické pojmy jako zlatý věk videoher a krach herního průmyslu roku 1983. Vysvětlila samotný pojem *indie game*, co vedlo k jeho vzniku a jakými způsoby se tyto hry nejčastěji distribuují. V přehledné tabulce stručně popsala některé rozdíly mezi vývojem nezávislého a komerčního herního projektu.

V následujících kapitolách popsala návrh a implementaci jednoduchého herního enginu založeného na platformě *XNA Game Studio 4.0*. Vysvětlila jak navrhnout a implementovat graf scény, grafické uživatelské rozhraní, získávání uživatelských vstupů, různé druhy kamer, in-game animační sekvence a v neposlední řadě, jak vytvořit komponentu spolupracující s *Cross-platform Audio Creation Tool*.

Nakonec se zaměřila na důležité součásti námi vyvíjené hry *FireFighters: Whatever it takes* - reprezentace lesa, šíření ohně, herní obrazovky, systém jednotek a jejich akce. Tato hra v době odevzdávání práce vyhrála národní kolo a postoupila do světového kola soutěže *Imagine Cup*, z čehož je zřejmé, že i malý tým dokáže v krátkém čase vytvořit konkurence schopný produkt, pokud je pro jeho tvorbu opravdu zapálen.

Literatura

[1] Bellis, Mary: *Computer and Video Game History* [online]. [cit. 2010-12-27]. Dostupné na URL: http://inventors.about.com/library/inventors/blcomputer_videogames.htm

[2] Donovan, Tristan: *Replay: History of video games*. Yellow Ant, Lewes, 2010, 138 stran. ISBN 978-0-9565072-2-8.

[3] Kent, Steven L.: The *Ultimate History of Video Games: From Pong to Pokémon*. Three Rivers Press, 2001. ISBN 0761536434.

[4] Stahl, Ted: *Timeline of Video Games* [online]. Aktualizováno 2010-03-17 [cit. 2011-01-09]. Dostupné na URL: http://www.thocp.net/software/games/games.htm

[5] Kolektiv autorů: *Pong-story* [online]. Aktualizováno 2011-01-04 [cit. 2011-01-05]. Dostupné na URL: http://www.pong-story.com/intro.htm

[6] Šulc, Tomáš: *Rating závadnosti počítačových her - jeho vývoj a dnešní praxe* [online]. Aktualizováno 2007-07-19 [cit. 2011-01-05]. Dostupné na URL: http://pctuning.tyden.cz/multimedia/hry-a-zabava/9102-rating_zavadnosti_pocitacovych_her-jeho_vyvoj_a_dnesni_praxe

[7] McMillen, Edmund: *Indie Game Development: 16 Do's and Don'ts* [online]. Aktualizováno 2010-04-22 [cit. 2011-01-05]. Dostupné na URL: http://www.gamepro.com/article/features/214865/indie-game-development-16-dos-and-donts/

[8] Hall, Steve: *What Are Indie Games?* [online]. Aktualizováno 2008-10-31 [cit. 2010-11-15]. Dostupné na URL: http://www.indiegamereviewer.com/what-are-indie-games/

[9] Kolektiv autorů: *History of Independent Games* [online]. Aktualizováno 2009-11-16 [cit. 2010-11-15]. Dostupné na URL: http://tig.wikia.com/wiki/History_of_Independent_Games

[10] Kolektiv autorů: *Game Maker* [online]. Aktualizováno 2009-11-16 [cit. 2010-11-15]. Dostupné na URL: http://tig.wikia.com/wiki/Game_Maker

[11] Kolektiv autorů: *Uplink game features* [online]. [cit. 2010-11-15]. Dostupné na URL: http://www.introversion.co.uk/uplink/

[12] Kolektiv autorů: *Introversion Software Limited* [online]. [cit. 2010-11-15]. Dostupné na URL: http://www.giantbomb.com/introversion-software-limited/65-543/

[13] Franta, Dalibor: *Recenze Tron Evolution - Perský princ v latexu*. Level, 2010, 199, s. 14-17.

[14] Cai, Yuanzhe: *Games-on-Demand: the Reality and Future* [online]. 2004, 11 stran. Dostupné na URL: http://www.parksassociates.com/free_data/dofwnloads/parks-games_on_demand.pdf

[15] Gril, Juan: *The State of Indie Gaming* [online]. Aktualizováno 2008-04-30 [cit. 2011-01-05]. Dostupné na URL: http://www.gamasutra.com/view/feature/3640/the_state_of_indie_gaming.php

[16] Langley, Ryan: *In-Depth: Xbox Live Indie Games Sales For 2009, Plus Some Perspective* [online]. Aktualizováno 2010-01-25 [cit. 2011-01-05]. Dostupné na URL: http://www.gamerbytes.com/2010/01/indepth_xbox_live_indie_games.php

[17] Kolektiv autorů: *To become an Authorized Developer for Wii, WiiWare and/or Nintendo DS/DSi* [online]. Aktualizováno 2011-01-09 [cit. 2011-01-05]. Dostupné na URL: http://www.warioworld.com/apply/

[18] Kohler, Chris: *Nintendo Taps U.S. Talent in Search of WiiWare Hits* [online]. Aktualizováno 2008-05-10 [cit. 2011-01-05]. Dostupné na URL: http://www.wired.com/gamelife/2008/05/for-wiiware-nin/

[19] List of Game Development Tools & Game Engines. Dostupné na URL: http://indiegametools.com/ [cit. 2011-01-06]

[20] Geršl, Vladimír: *Game Design: Pro počítačové a mobilní hry* [online]. Aktualizováno 2010-06-05 [cit. 2011-01-08]. Dostupné na URL: http://herakles.zcu.cz/education/ph/slide/PH-10-Gamedesign.ppt

[21] Michael, David: *The Indie Game Development Survival Guide*. Charles River Media, 2003. ISBN: 1-58450-214-2

[22] Juuso: *What Are AAA Titles?* [online]. Aktualizováno 2006-05-26 [cit. 2011-01-06]. Dostupné na URL: http://www.gameproducer.net/2006/05/26/what-are-aaa-titles/

[23] Kršek, P., Španěl, M.: *Základy počítačové grafiky: OpenGL* [online]. Aktualizováno 2008 [cit. 2010-12-20]. Přístupné studentům FIT VUT v Brně.

[24] Mungler, Steven: *Just what is DirectX?* [online]. Aktualizováno 2004-09-21 [cit. 2011-01-05]. Dostupné na URL: http://www.d-silence.com/feature.php?id=254

[25] Herout, Adam: *Srovnání OpenGL a Direct3D* [online]. Aktualizováno 2006 [cit. 2010-12-26]. Přístupné studentům FIT VUT v Brně.

[26] Kajan, Rudolf: *DirectX, Direct3D* [online]. Aktualizováno 2010 [cit. 2010-12-26]. Přístupné studentům FIT VUT v Brně.

[27] Herout, Adam: *Historie a Standardizace OpenGL a dalších…* [online]. Aktualizováno 2007 [cit. 2010-12-26]. Přístupné studentům FIT VUT v Brně.

[28] Wright, Richard: *OpenGL SuperBible*. Sams, 2004, 1173 stran. ISBN: 978-0672326011.

[29] Domovská webová stránka Torque Game Enginu. Dostupné na URL: http://www.torquepowered.com

[30] Preisz, Eric: *November Update* [online]. Aktualizováno 2010-11-11 [cit. 2011-01-05]. Dostupné na URL: http://www.torquepowered.com/community/blogs/view/20495

[31] Kolektiv autorů: *Unreal Development Kit* [online]. Aktualizováno 2010-11-27 [cit. 2011-01-05]. Dostupné na URL: http://developer.nvidia.com/object/udk.html

[32] Domovská webová stránka Unreal Technology. Dostupné na URL: http://www.unrealtechnology.com

[33] Sweeney, Tim: *Unreal Engine 4.0 aims at next-gen console war* [online]. Aktualizováno 2008-03-12 [cit. 2011-01-05]. Dostupné na URL: http://www.tgdaily.com/business-and-law-features/36436-tim-sweeney-part-3-unreal-engine-40-aims-at-next-gen-console-war

[34] Kolektiv autorů: *XNA Game Studio 4.0* [online]. Dostupné na URL: http://msdn.microsoft.com/en-us/library/bb200104.aspx

[35] Ewald, M.: *Game Components and Game Services* [online]. Aktualizováno 2006-11-01 [cit. 2011-01-04]. Dostupné na URL: http://www.nuclex.org/articles/4-architecture/6-game-components-and-game-services

[36] Walsh, Aaron E.: *Understanding Scene Graphs* [online]. Aktualizováno 2002-07-01 [cit. 2011-05-19]. Dostupné na URL: http://drdobbs.com/java/184405094

[37] Kolektiv autorů: *Extension Methods (C# Programming Guide)* [online]. Aktualizováno: 2010-10-01. Dostupné na URL: http://msdn.microsoft.com/en-us/library/bb383977.aspx

[38] Kolektiv autorů: *Game State Management* [online]. Aktualizováno 2007-04-26 [cit. 2011-04-12]. Dostupné na URL: http://create.msdn.com/en-US/education/catalog/sample/game_state_management

[39] Kolektiv autorů: *Cursor Constructor* [online]. Aktualizováno 2008-01-06 [cit. 2011-04-11]. Dostupné na URL: http://msdn.microsoft.com/en-us/library/kkw8k45d.aspx