Středoškolská odborná Činnost 2006/2007

Obor 10 - elektrotechnika, elektronika, telekomunikace a technická informatika

Matematické nástroje na řešení pohybu a kolizí objektů ve virtuální realitě

Autoři:
Vladimír Černý, Štěpán Vyterna
SPŠ a VOŠ Písek
Karla Čapka 402
397 11 Písek

Konzultant práce:
RNDr. Miroslav Procházka
(SPŠ a VOŠ Písek)

Písek 2007

Jížní Čechy

Prohlášení

Prohlašujeme tímto, že jsme soutěžní práci vypracovali samostatně pod vedením RNDr. Miroslava Procházky a uvedli v seznamu literatury veškerou použitou literaturu a další informační zdroje včetně internetu.

V Písku dne 12.3.2007

vlastnoruční podpisy autorů

Anotace

Využití analytické a deskriptivní geometrie k výpočtu kolizí mezi 3D objekty. Demonstrace multiplatformního využití jazyka C++.

Hlavními částmi práce jsou: určení pozice objektů (lineární transformace), výpočet kolizí mezi objekty, jejich využití k procházení prostředím, využití knihoven OpenGL k jejich vykreslování a ukládání do paměti grafické karty a multiplatformní programování.

Obsah

Kapitola 1

Úvod

1.1 Program

Program umožňuje procházet 3D prostředím. Vypočítávají se koliye s objekty a tudíž nemůže dojít k procházení stěnami. Díky použitému způsobu procházení je možné se pohybovat v jakémkoli scéně včetně vícepodlažních budov.

1.2 Cíle

Cílem bylo vytvořit co nejjednodušší a nejrychlejší engine[1], který by přesto umožňoval značnou univerzálnost a multiplatformnost.

[1] Program který zajišťuje zobrazování a pohyb objektů

Kapitola 2

Metodika

2.1 Analytická geometrie

Analytická geometrie slouží k matematickému (pomocí rovnic) vyjádření geometrických útvarů. Pomocí tohoto vyjádření lze poté vypočítávat umístění, vzájemné interakce a zobrazení útvarů.

V programu se analytická geometrie využívá hlavně k výpočtu kolizí.

2.2 Vyjádření geometrických útvarů

2.2.1 Bod

Základní jednotkou geometrických obrazců je vždy bod. V třírozměrném prostoru je bod určen třemi číselnými souřadnicemi.

$$\bar{A} = (A_x; A_y; A_z)$$

2.2.2 Vektor

Vektor určuje směr v prostoru. Vektor je určen třemi čísly. Vektor může být také určen dvěma body, potom se souřadnice vektoru vypočtou jako rozdíl těchto bodů (2.1).

$$\vec{V} = (V_x; V_y; V_z)$$

$$\begin{aligned} \vec{AB} &= \bar{B} - \bar{A} \\ \vec{AB} &= (B_x - A_x; B_y - A_y; B_z - A_z) \end{aligned} \tag{2.1}$$

2.2.2.1 Normovaný vektor

Často potřebujeme znát vektor pouze kvůli směru a ne velikosti, proto se takový vektor převádí na vektor normovaný (s velikostí jedna). Pro provedení tohoto přepočtu stačí vydělit všechny souřadnice velikostí vektoru (2.2).

$$\vec{V_n} = \frac{\vec{V}}{\left|\vec{V}\right|} \tag{2.2}$$

2.2.2.2 Skalární součin

Pokud je potřeba zjistit úhel mezi vektory, provádí se tzv. skalární součin. Hodnota skalárního součinu dvou normovaných vektorů (kapitola 2.2.2.1) odpovídá kosinu úhlu mezi vektory (2.3). Skalární součin je součet součinů jednotlivých souřadnic (2.4).

$$\begin{aligned} \cos\varphi &= \vec{V_1} \cdot \vec{V_2} && (2.3) \\ \vec{V_1} \cdot \vec{V_2} &= V_{1x}V_{2x} + V_{1y}V_{2y} + V_{1z}V_{2z} && (2.4) \end{aligned}$$

Protože se skalární součin rovná kosinu úhlu, tak nulový skalární součin (zde nezáleží na velikosti vektorů - můžou mít i jinou velikost než normovaný vektor) znamená kolmé vektory (2.5).

$$\vec{V_1} \cdot \vec{V_2} = 0 \tag{2.5}$$

Pokud je potřeba vypočítat vektor kolmý k jiným dvěma vektorům (například k rovině) vychází soustava dvou rovnic o třech neznámých (2.6). Po dosazení jedné souřadnice do druhé rovnice dostaneme rovnici o dvou neznámých (2.7). Jedním z možných řešení[1] je, že se souřadnice budou rovnat členu, který násobí druhou souřadnici (2.8).

$$\begin{aligned} V_{1x}V_{2x} + V_{1y}V_{2y} + V_{1z}V_{2z} &= 0 \\ V_{1x}V_{3x} + V_{1y}V_{3y} + V_{1z}V_{3z} &= 0 \end{aligned} \tag{2.6}$$

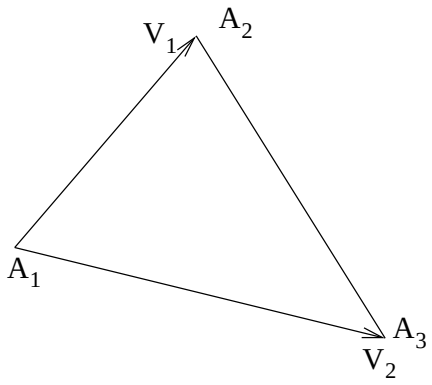
$$\begin{aligned} V_{1z} &= -\frac{V_{1x}V_{2x} + V_{1y}V_{2y}}{V_{2z}} \\ V_{1x}V_{3x} + V_{1y}V_{3y} - \frac{V_{1x}V_{2x} + V_{1y}V_{2y}}{V_{2z}}V_{3z} &= 0 \\ V_{1x}V_{3x}V_{2z} + V_{1y}V_{3y}V_{2z} - V_{1x}V_{2x}V_{3z} - V_{1y}V_{2y}V_{3z} &= 0 \\ V_{1x}\left(V_{3x}V_{2z} - V_{2x}V_{3z}\right) &= V_{1y}\left(V_{2y}V_{3z} - V_{3y}V_{2z}\right) \end{aligned} \tag{2.7}$$

$$\begin{aligned} V_{1x} &= V_{2y}V_{3z} - V_{3y}V_{2z} \\ V_{1y} &= V_{3x}V_{2z} - V_{2x}V_{3z} \end{aligned} \tag{2.8}$$

Po dosazení do první rovnice získáme způsob, jak ze dvou vektorů vypočítat vektor na ně kolmý (2.9).

[1] Řešení rovnice o dvou neznámých je samozřejmě nekonečně mnoho. Nás ale zajímá pouze směr vektoru a ten je stejný pro všechny řešení.

Obrázek 2.1: Rovina



$$
\begin{aligned}
V_{1z} &= -\frac{V_{2x}\left(V_{2y}V_{3z} - V_{3y}V_{2z}\right) + V_{2y}\left(V_{3x}V_{2z} - V_{2x}V_{3z}\right)}{V_{2z}} \\
V_{1z} &= -\frac{V_{2x}V_{2y}V_{3z} - V_{2x}V_{3y}V_{2z} + V_{3x}V_{2y}V_{2z} - V_{2x}V_{2y}V_{3z}}{V_{2z}} \\
V_{1z} &= \frac{V_{2x}V_{3y}V_{2z} - V_{3x}V_{2y}V_{2z}}{V_{2z}} \\
V_{1z} &= V_{2x}V_{3y} - V_{3x}V_{2y}
\end{aligned}
$$

$$
\begin{aligned}
V_{1x} &= V_{2y}V_{3z} - V_{3y}V_{2z} \\
V_{1y} &= V_{3x}V_{2z} - V_{2x}V_{3z} \\
V_{1z} &= V_{2x}V_{3y} - V_{3x}V_{2y}
\end{aligned}
\tag{2.9}
$$

2.2.3 Přímka

Přímka je definována bodem a vektorem. V parametrickém vyjádření (2.10) se libovolný bod na přímce vypočte jako součet bodu a vektoru vynásobeného parametrem t. Parametr tedy vlastně určuje vzdálenost od počátečního bodu přímky v násobcích délky vektoru.

$$\bar{p} = \bar{A} + t\vec{V} \tag{2.10}$$

2.2.4 Rovina

Rovina je parametricky určena podobně jako přímka bodem a dvěma vektory (2.11). Protože ale rovina bývá zadána třemi body, můžou se vektory vyjádřit pomocí rozdílu těchto bodů (2.12).

V grafice se většinou nezobrazuje celá rovina ale pouze část ohraničená trojúhelníkem. Parametry $(u, v, 1-u-v)$ v rovnici roviny (2.12) vyjadřují vzdálenost od bodů.Pokud chceme zjistit pouze prostor patřící do trojúhelníku ohraničeného body, nesmí být parametry záporné. Z toho vycházejí podmínky (2.13) pro které platí že bod je uvnitř trojúhelníka.

$$\bar{f} = \bar{A}_1 + u\vec{V}_1 + v\vec{V}_2 \tag{2.11}$$

$$\begin{aligned}
\bar{f} &= \bar{A_1} + u\left(\bar{A_2} - \bar{A_1}\right) + v\left(\bar{A_3} - \bar{A_1}\right) \\
\bar{f} &= \bar{A_1} + u\bar{A_2} - u\bar{A_1} + v\bar{A_3} - v\bar{A_1} \\
\bar{f} &= \left(1 - u - v\right)\bar{A_1} + u\bar{A_2} + v\bar{A_3}
\end{aligned} \tag{2.12}$$

$$\begin{aligned}
1 - u - v &\geq 0 \\
u + v &\leq 1 \\
u &\geq 0 \\
v &\geq 0
\end{aligned} \tag{2.13}$$

2.3 Kolize v 3D prostoru

Pokud mají objekty na sebe reagovat, je nutné zjistit kdy mezi nimi dochází ke kolizi. Ke kolizi dochází, když mají dva objekty společný jeden nebo více bodů.

Kolize se většinou počítá tak, že se řeší rovnice jednotlivých objektů jako soustava rovnic. Pokud má soustava řešení, objekty mají společný bod (body) a tedy dochází ke kolizi.

2.3.1 Druhy kolizí

Kolizi ve třírozměrném prostoru je možné zjišťovat různými způsoby. Základní rozdělení vychází z toho jaké druhy objektů kolidují.

- Koule - Koule
- Rovina - Rovina
- Přímka - Koule
- Přímka - Rovina

2.3.1.1 Koule - Koule

Kolize dvou koulí (obr. 2.2) je nejjednodušší případ kolize. Pro zjištění, jestli došlo ke kolizi, stačí zjistit vzdálenost středů koulí a porovnat ji se součtem poloměrů (2.14). Pokud je vzdálenost menší, došlo ke kolizi. Vzdálenost středů lze jako každou vzdálenost dvou bodů jednoduše vypočítat pomocí Pythagorovy věty (2.15). Umocnění probíhá v procesoru rychleji než odmocnění, proto je časově výhodnější porovnávat druhé mocniny (2.16).

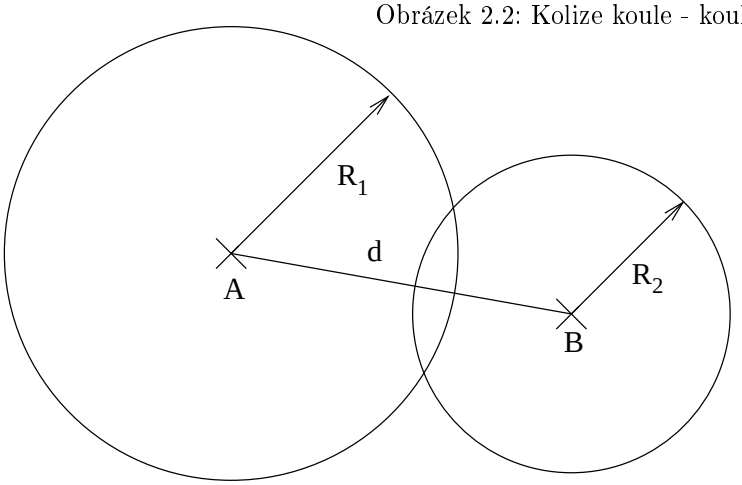
$$d < R_1 + R_2 \tag{2.14}$$

$$\sqrt{\left(A_x - B_x\right)^2 + \left(A_y - B_y\right)^2 + \left(A_z - B_z\right)^2} < R_1 + R_2 \tag{2.15}$$

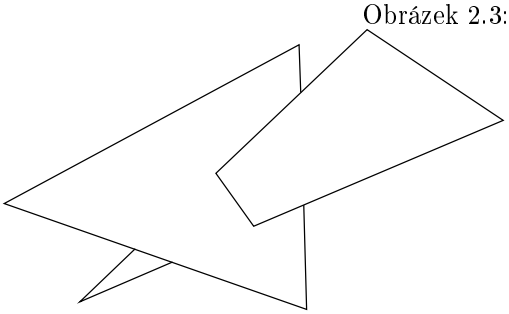
$$\left(A_x - B_x\right)^2 + \left(A_y - B_y\right)^2 + \left(A_z - B_z\right)^2 < \left(R_1 + R_2\right)^2 \tag{2.16}$$

Tento způsob lze použít pouze v případě že se tvar objektů blíží kouli nebo je možné sestavit objekty z více koulí.

Obrázek 2.2: Kolize koule - koule



Obrázek 2.3: Kolize rovina - rovina



2.3.1.2 Rovina - Rovina

Protože při zobrazování se všechny objekty skládají z rovinných trojúhelníků (face), jeví se výpočet kolize pomocí rovin (obr. 2.3) jako velmi výhodný. Nevýhodou tohoto řešení je, že je třeba pro zjištění kolize dvou objektů porovnat všechny kombinace, a to si vyžaduje procesorový čas. Další nevýhodou je že při pomalejších výpočtech může docházet k procházení objekty (kapitola 2.3.2).

2.3.1.3 Přímka - Koule

Přímka jako jeden z kolidujících objektů umožňuje například interakci myši s prostorem. Přímka potom vede z počátku kamery přes kurzor myši a vyhodnocuje se kolize s koulemi kolem jednotlivých objektů, na které je možno kliknout.

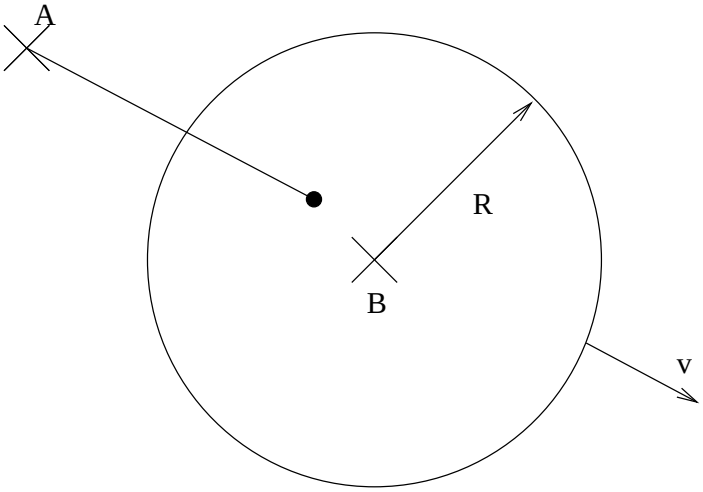
Pro zjištění této kolize je třeba vypočítat vzdálenost bodu od přímky.

2.3.1.4 Přímka - Rovina

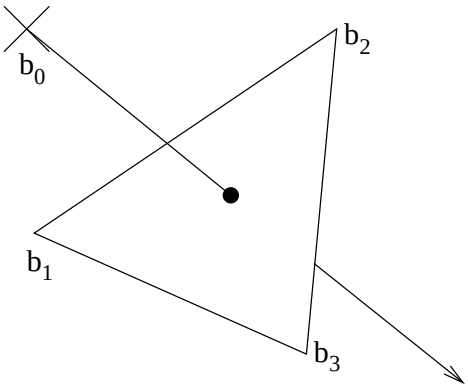
Kolizí přímky a roviny ze dosáhnout výpočtu kolizí, který není závislý na frekvenci výpočtů (kapitola 2.3.2).

Matematické vyjádření kolize získáme, když položíme rovnici přímky (2.17) a roviny (2.18) sobě rovny (2.19). Po úpravě dostaneme rovnici o třech neznámých (2.20). Pro větší přehlednost nahradíme konstantní výrazy konstantami (2.21). Protože se jedná o rovnici s prostorovými body můžeme rovnici rozepsat na tři rovnice (2.22) pro každou souřadnici zvlášť. Vznikne soustava tří rovnic o

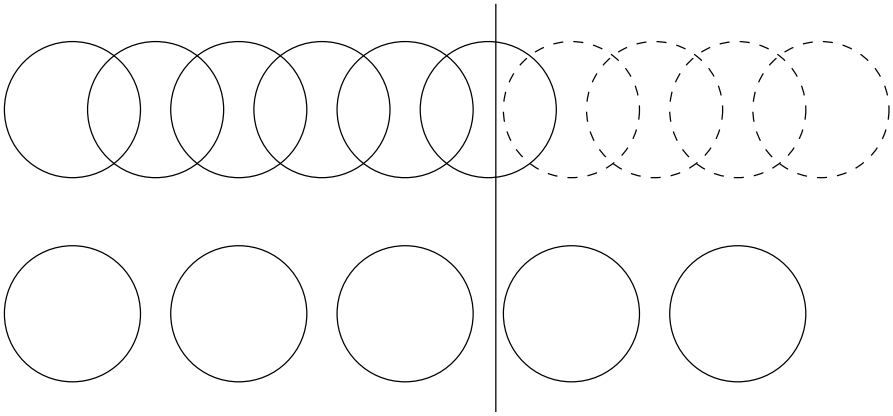
Obrázek 2.4: Kolize přímka - koule



Obrázek 2.5: Kolize přímka - rovina



Obrázek 2.6: Závislost výpočtu kolizí na rychlosti výpočtů



třech neznámých, kterou je možno řešit různými způsoby (v programu je použito řešení pomocí determinantů - kapitola 2.4).

$$\bar{p} = \bar{b_0} + t\vec{s} \tag{2.17}$$

$$\bar{f} = (1 - u - v)\bar{b_1} + u\bar{b_2} + v\bar{b_3} \tag{2.18}$$

$$\begin{aligned}
\bar{b_0} + t\vec{s} &= (1 - u - v)\bar{b_1} + u\bar{b_2} + v\bar{b_3} && (2.19)\\
\bar{b_0} + t\vec{s} &= \bar{b_1} - u\bar{b_1} - v\bar{b_1} + u\bar{b_2} + v\bar{b_3} \\
\bar{b_0} + t\vec{s} - \bar{b_1} + u\bar{b_1} + v\bar{b_1} - u\bar{b_2} - v\bar{b_3} &= 0 \\
u(\bar{b_1} - \bar{b_2}) + v(\bar{b_1} - \bar{b_3}) + t\vec{s} + (\bar{b_0} - \bar{b_1}) &= 0 \\
u(\bar{b_1} - \bar{b_2}) + v(\bar{b_1} - \bar{b_3}) + t\vec{s} &= \bar{b_1} - \bar{b_0} && (2.20)
\end{aligned}$$

$$u\bar{a_1} + v\bar{a_2} + t\bar{a_3} = \bar{a_4} \tag{2.21}$$

$$\begin{aligned}
ua_{1x} + va_{2x} + ta_{3x} &= a_{4x} \\
ua_{1y} + va_{2y} + ta_{3y} &= a_{4y} && (2.22)\\
ua_{1z} + va_{2z} + ta_{3z} &= a_{4z}
\end{aligned}$$

2.3.2 Závislost kolizí na rychlosti výpočtů

Běžný cyklus programu provede výpočet kolizí, pohyb objektů podle výsledků kolizí a vykreslení snímku. Rychlost provádění takového cyklu je závislá na výkonu a vytížení hardwaru. Protože je třeba, aby se objekty pohybovaly stále stejnou rychlostí, je vzdálenost pohybu za snímek závislá na rychlosti vykreslování[2].

Na pomalých nebo vytížených strojích se budou objekty pohybovat o příliš velké vzdálenosti. Při špatně navrhnutém kolizním řešení to může vest k tomu, že pohybující se objekty projdou jinými objekty bez toho aby byla zaznamenána kolize (obr. 2.6).

Tomuto problému lze předejít například tak, že se použije kolize přímka-rovina, kde přímka vychází z objektu ve směru pohybu.

[2]rychlost vykreslování se většinou udává ve snímcích za sekundu anglicky frames per second - FPS

2.4 Řešení soustavy rovnic pomocí determinantů

Při řešení soustavy rovnic pomocí determinantů se využívá Cramerovo pravidlo.

Algoritmus řešení Je zadána soustava rovnic pomocí matice (2.23). Vypočte se determinant pro matici na levé straně rovnice (2.24). Sloupec náležící proměnné, která se vypočítává zaměníme z maticí pravé strany (2.25). Pro výslednou matici se také vypočte determinant. Výsledná proměnná je pak rovna podílu determinantu z prohozené matice a matice levé strany (2.26). Ostatní proměnné se vypočítají obdobně (2.27, 2.28, 2.29, 2.30).

Pokud determinant levé strany vyjde roven nule, není možné zjistit jedno řešení. To znamená že řešení neexistuje nebo je řešení více. Při výpočtu kolizí to znamená že objekty, pro které se kolize počítá, jsou rovnoběžně.

$$\begin{aligned} ua_{1x} + va_{2x} + ta_{3x} &= a_{4x} \\ ua_{1y} + va_{2y} + ta_{3y} &= a_{4y} \\ ua_{1z} + va_{2z} + ta_{3z} &= a_{4z} \end{aligned}$$

$$\left.\begin{array}{ccc|c} a_{1x} & a_{2x} & a_{3x} & a_{4x} \\ a_{1y} & a_{2y} & a_{3y} & a_{4y} \\ a_{1z} & a_{2z} & a_{3z} & a_{4z} \end{array}\right. \tag{2.23}$$

$$D = \begin{bmatrix} a_{1x} & a_{2x} & a_{3x} \\ a_{1y} & a_{2y} & a_{3y} \\ a_{1z} & a_{2z} & a_{3z} \end{bmatrix} \tag{2.24}$$

$$u_d = \begin{bmatrix} a_{4x} & a_{2x} & a_{3x} \\ a_{4y} & a_{2y} & a_{3y} \\ a_{4z} & a_{2z} & a_{3z} \end{bmatrix} \tag{2.25}$$

$$u = \frac{u_d}{D} \tag{2.26}$$

$$v_d = \begin{bmatrix} a_{1x} & a_{4x} & a_{3x} \\ a_{1y} & a_{4y} & a_{3y} \\ a_{1z} & a_{4z} & a_{3z} \end{bmatrix} \tag{2.27}$$

$$v = \frac{v_d}{D} \tag{2.28}$$

$$t_d = \begin{bmatrix} a_{1x} & a_{2x} & a_{4x} \\ a_{1y} & a_{2y} & a_{4y} \\ a_{1z} & a_{2z} & a_{4z} \end{bmatrix} \tag{2.29}$$

$$t = \frac{t_d}{D} \tag{2.30}$$

2.5 Lineární transformace

V prostorové grafice je často potřeba provést změnu umístění jednotlivých objektů nebo kamery. Aby se nemuselo zasahovat do struktury objektů (měnit souřadnice jednotlivých bodů), tak se pro každý bod před vykreslením provede lineární transformace.

Lineární transformace spočívá ve vynásobení bodu (nebo vektoru) transformační maticí (2.31). Transformační matice je číselná matice o rozměrech 4×4. Bod (nebo vektor) je určen čtyrmi souřadnicemi - x, y, z a váhou w. Váha je pro bod rovna jedné a pro vektor je nulová.

Souřadnice výsledného bodu se získají váženým součtem bodu původního. Váhy v tomto součtu určuje právě transformační matice (2.32). U bodu je váha vždy rovna jedné (2.33). Aby i u výsledného bodu vyšlo $w = 1$ všechny souřadnice se vydělí w[3] (2.34).

$$\grave{B} = B \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \tag{2.31}$$

$$\begin{aligned} \grave{x} &= a_{11}x + a_{12}y + a_{13}z + a_{14}w \\ \grave{y} &= a_{21}x + a_{22}y + a_{23}z + a_{24}w \\ \grave{z} &= a_{31}x + a_{32}y + a_{33}z + a_{34}w \\ \grave{w} &= a_{41}x + a_{42}y + a_{43}z + a_{44}w \end{aligned} \tag{2.32}$$

$$\begin{aligned} \grave{x} &= a_{11}x + a_{12}y + a_{13}z + a_{14} \\ \grave{y} &= a_{21}x + a_{22}y + a_{23}z + a_{24} \\ \grave{z} &= a_{31}x + a_{32}y + a_{33}z + a_{34} \\ \grave{w} &= a_{41}x + a_{42}y + a_{43}z + a_{44} \end{aligned} \tag{2.33}$$

$$\begin{aligned} \grave{x} &= \frac{a_{11}x + a_{12}y + a_{13}z + a_{14}}{a_{41}x + a_{42}y + a_{43}z + a_{44}} \\ \grave{y} &= \frac{a_{21}x + a_{22}y + a_{23}z + a_{24}}{a_{41}x + a_{42}y + a_{43}z + a_{44}} \\ \grave{z} &= \frac{a_{31}x + a_{32}y + a_{33}z + a_{34}}{a_{41}x + a_{42}y + a_{43}z + a_{44}} \\ \grave{w} &= 1 \end{aligned} \tag{2.34}$$

Tímto způsobem je možné dosáhnout základních transformací jako je posunutí, otočení, zvětšení a zkosení. Výhodou tohoto řešení je, že při použití více transformací najednou (např. otočení a posunutí) stačí nejdříve vynásobit (kapitola 2.5.1) jednotlivé matice a bod násobit až výslednou.

Pokud se souřadnice bodů zachovávají použije se tzv. identická matice (2.35). Pro posun stačí nastavit poslední sloupec matice (2.36) který se násobí w, tedy 1 pro bod a nezapočítává se u vektoru (vektor má pouze směr, nemá pozici). Zvětšení je dáno čísly v úhlopříčce matice (2.37). Matice pro rotaci a zkosení jsou složitější, záleží na tom podél jaké osy se objekt rotuje (kosí).

$$\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix} \tag{2.35}$$

$$\begin{matrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{matrix} \tag{2.36}$$

[3] Někdy se tato operace nazývá perspektivní korekce

Ukázka kódu 2.1: Provedení lineárni transformace

```
//vynásobeni transformační maticí
ubod[0] = matice[0] * bod[0] + matice[0+4]*bod[1]
        + matice[0+8] * bod[2] + matice[0+12];
ubod[1] = matice[1] * bod[0] + matice[1+4]*bod[1]
        + matice[1+8] * bod[2] + matice[1+12];
ubod[2] = matice[2] * bod[0] + matice[2+4] * bod[1]
        + matice[2+8] * bod[2] + matice[2+12];
w = matice[3] * bod[0] + matice[3+4] * bod[1]
    + matice[3+8] * bod[2] + matice[3+12];
//provedení "perspektivní korekce"
w=1/w;
ubod[0]*=w;
ubod[1]*=w;
ubod[2]*=w;
```

$$\begin{matrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{matrix} \tag{2.37}$$

2.5.1 Násobení matic

Při provádění několika lineárních transformací je vhodnější mezi sebou matice vynásobit a potom provést transformaci výslednou maticí.

Hodnota výsledné buňky je součet násobků buněk ze stejné řádky jedné matice a sloupce druhé matice (2.38).

$$\begin{aligned} AB &= \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \\ AB &= \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix} \end{aligned} \tag{2.38}$$

2.5.2 Inverzní matice

Pokud chceme provést transformaci objektu přesně opačným směrem, jednou z možností je provést lineární transformaci pomocí matice inverzní k původní. Matice jsou vzájemně inverzní jestliže jejich vynásobením dostaneme identickou matici.

Výpočet Inverzní matice se vypočte tak že se k původní matici např. (2.39) přidá identická (2.40). Následně se provádí řádkové úpravy celé (původní i identické) matice tak aby na levé straně vyšla identická matice.

Nejprve se matice seřadí aby řádky začínající nulami byly pod řádky začínajícími jiným číslem. Potom se provádí řádkové operace pro každý řádek. Řádek se vydělí prvním nenulovým číslem (2.41), první číslo bude tedy jedna. Od všech ostatních řádků se odečte aktuální řádek vynásobený

Ukázka kódu 2.2: Vytvoření inverzní matice

```
float matrixP[32]; //pomocná matice
for(i=0;i<4;i++) //zkopirovaní do pomocné
  for(j=0;j<4;j++)
    matrixP[i+j*8]=matrix[i+j*4];
for(i=0;i<4;i++) //doplnění pomocné o identickou
  for(j=0;j<4;j++)
    matrixP[4+i+j*8]=(i==j?1:0);//všude nuly úhlopříčka jedničky
int r;
float pomocna;
for(i=0;i<4;i++){ //pro všechny řádky
  if (matrixP[i*9]==0.0) //hlavní pole nesmí být nula
    for(r=i;r<4;r++)
      if (matrixP[i+r*8]!=0.0) //nalezení nenulového radku pod nulovým;
        for(j=0;j<8;j++){ //prohození řadků
          pomocna=matrixP[j+i*8];
            matrixP[j+i*8]=matrixP[j+r*8];
            matrixP[j+r*8]=pomocna;
        };
  pomocna=matrixP[i*9];
  for(j=0;j<8;j++) //vyděleni řádku hodnotou hlavního pole
    matrixP[j+i*8]/=pomocna;
  for(r=0;r<4;r++)
    if (r!=i){ //vynulováni zbytku sloupce (všechny ostatní řádky)
      pomocna=matrixP[i+r*8];
      for(j=0;j<8;j++)
      matrixP[j+r*8]-=matrixP[j+i*8]*pomocna;
    };
};
for(i=0;i<4;i++) //zkopírovaní z pomocné matice
  for(j=0;j<4;j++)
    matrixI[i+j*4]=matrixP[4+i+j*8];
```

tak aby se odstranily číslice pod a nad jedničkou (2.42). Po zopakování postupu pro všechny řádky vyjde na levé straně identická matice (2.43).

Matice na pravé straně (2.44) je k původní inversní. Implementace tohoto výpočtu je v ukázce 2.2.

$$\begin{matrix} 3 & 2 & 6 & 2 \\ 6 & 8 & 4 & 3 \\ 3 & 4 & 3 & 6 \\ 3 & 6 & 8 & 8 \end{matrix} \tag{2.39}$$

$$\left(\begin{array}{cccc|cccc} 3 & 2 & 6 & 2 & 1 & 0 & 0 & 0 \\ 6 & 8 & 4 & 3 & 0 & 1 & 0 & 0 \\ 3 & 4 & 3 & 6 & 0 & 0 & 1 & 0 \\ 3 & 6 & 8 & 8 & 0 & 0 & 0 & 1 \end{array}\right) \tag{2.40}$$

$$\left(\begin{array}{cccc|cccc} 1 & \frac{2}{3} & 2 & \frac{2}{3} & \frac{1}{3} & 0 & 0 & 0 \\ 6 & 8 & 4 & 3 & 0 & 1 & 0 & 0 \\ 3 & 4 & 3 & 6 & 0 & 0 & 1 & 0 \\ 3 & 6 & 8 & 8 & 0 & 0 & 0 & 1 \end{array}\right) \tag{2.41}$$

$$\left(\begin{array}{cccc|cccc} 1 & \frac{2}{3} & 2 & \frac{2}{3} & \frac{1}{3} & 0 & 0 & 0 \\ 0 & 4 & -8 & -1 & -2 & 1 & 0 & 0 \\ 0 & 2 & -3 & 4 & -1 & 0 & 1 & 0 \\ 0 & 4 & 2 & 5 & -1 & 0 & 0 & 1 \end{array}\right) \tag{2.42}$$

$$\left(\begin{array}{cccc|cccc} 1 & 0 & 0 & 0 & \frac{71}{234} & -\frac{11}{234} & \frac{35}{117} & -\frac{85}{234} \\ 0 & 1 & 0 & 0 & -\frac{43}{156} & -\frac{257}{156} & -\frac{19}{78} & \frac{35}{156} \\ 0 & 0 & 1 & 0 & \frac{3}{26} & -\frac{25}{26} & -\frac{2}{13} & \frac{3}{26} \\ 0 & 0 & 0 & 1 & -\frac{1}{39} & \frac{4}{39} & \frac{10}{39} & -\frac{1}{39} \end{array}\right) \tag{2.43}$$

$$\begin{matrix} \frac{71}{234} & -\frac{11}{234} & \frac{35}{117} & -\frac{85}{234} \\ -\frac{43}{156} & -\frac{257}{156} & -\frac{19}{78} & \frac{35}{156} \\ \frac{3}{26} & -\frac{25}{26} & -\frac{2}{13} & \frac{3}{26} \\ -\frac{1}{39} & \frac{4}{39} & \frac{10}{39} & -\frac{1}{39} \end{matrix} \tag{2.44}$$

2.5.3 Lineární transformace v openGL

OpenGL provádí lineární transformaci vždy, když vykresluje nějaký bod. Proto má uloženou aktuální matici podle které se transformace provádí. Tuto aktuální matici je možné nastavit (glLoadMatrix), vynásobit ji jinou pro provedení více různých transformaci (glMultMatrix), načíst identickou (glLoadIdentity) a provádět základní operace jako je zvětšení posun a rotace (glScale, glTranslate, glRotate).

Tyto operace provádí OpenGL s podporou hardware, takže pokud to grafická karta umožňuje je tato operace prováděna přímo v ní. Takže procesor není zatěžován.

2.6 Display-listy

2.6.1 Co to je display-list

Display-listy prostředí OpenGL jsou navrženy tak, aby optimalizovali výkon při síťovém zobrazování, ale také aby nikdy nezvyšovaly nároky na výkon při běhu na lokálním počítači.

Pro optimalizaci výkonu je OpenGL display-list spíše zásobárna příkazů, než dynamická databáze. Jinými slovy, pokud jednou display-list vytvoříte, nelze jej později změnit. Pokud by byly display-listy modifikovatelné, výkon by byl velmi zredukován, protože by bylo potřeby mnohem více času na procházení display-listů a hledání změn v nich.

Spouštění display-listu netrvá déle než spouštění jednotlivých příkazů jeden po druhém. Jistě, je zde malá prodleva při skoku do display-listu a poté při skoku zpět, ale pokud display-list dobře optimalizujete a také jej použijete na vhodném místě, je velmi výhodné jej používat, protože se tím nejen zpřehlední kód, ale také zde nastává podstatné zrychlení při vykreslování složitějších scén, protože objekt je většinou již uložen v paměti grafické karty a pouze se umístí.

2.6.2 Na co lze display-listy použít

- **Maticové operace**
  - Většina maticových operací vyžaduje OpenGL pro výpočet inverze. Vypočtená matice a její inverzní matice tak mohou být uloženy každá ve svém vlastním display-listu.
- **Rastrové bitmapy a obrázky**
  - Formát, kterým specifikujeme rastrová data není pravděpodobně nejpřijatelnější pro hardware. Pokud je display-list jednou „vykompilován," OpenGL jej může transformovat do podoby, která je pro hardware nejpřijatelnější. Tato skutečnost se nejradikálněji projevuje při vykreslování rastrových písem, protože jeden znak většinou obsahuje sérii malých bitmap.
- **Světla, vlastnosti materiálů a světelné modely**
  - Pokud vykreslujeme scénu s komplexními světelnými podmínkami, museli bychom měnit nastavení materiálu zvlášť pro každý objekt na scéně, což může být při významějších výpočtech velmi zpomalující. Pokud však umístíme specifikaci materiálu do display-listu, tak tyto výpočty nebudeme muset provádět pokaždé, když změníme materiál, protože budeme ukládat pouze výsledek výpočtu.
- **Textury**
  - Pokud budeme schopni uložit texturová data do display-listu, dosáhneme opravdu velkého zrychlení, protože formát uložení dat v hardwaru se pravděpodobně bude lišit od formátu uložení dat v OpenGL, takže konverzi bude třeba provádět jen jednou, při vytváření display-listu.

2.6.3 Jak vytvářet display-listy

Pro zahájení záznamu do display-listu se používá příkazem *void glNewList()*, pro ukončení záznamu příkaz *void glEndList()*. Do příkazu *void glNewList(GLuint list, GLenum mode)* zadáváme dva parametry. První parametr (*GLuint list*) je celočíselný a jedinečný identifikátor vytvořeného display-listu. Pomocí tohoto identifikátoru můžeme display-list provést. Druhý parametr (*GLenum mode*) určuje, zda se má display-list pouze vytvořit (hodnota GL_COMPILE), nebo vytvořit a hned také provést (hodnota GL_COMPILE_AND_EXECUTE).

Zavolání display-listu (tedy vyvolání příkazů uložených v display-listu) provádíme funkcí *void glCallList(GLuint list)*, v jejímž parametru list je uložen identifikátor dříve vytvořeného display-listu.

Ukázka kódu 2.3: Display-list

```
#define MUJ_KRUH 1
udelejKruh () {
 GLint i;
 GLfloat cosinus, sinus;
 glNewList (MUJ_KRUH, GL_COMPILE);
   glBegin (GL_POLYGON);
    for(i = 0; i < 100; i++){
      cosinus = cos (i * 2 * PI / 100.0);
      sinus = sin (i * 2 * PI / 100.0);
      glVertex2f (cosinus, sinus);
    }
   glEnd ();
 glEndList ();
}
```

2.6.4 Příklad využití display-listu

Algoritmus ukazuje, jak lze vytvořit jednoduchý display-list, který vykreslí kružnici o 100 segmentech. Celý tento display-list se přeloží a nahraje do paměti grafické karty, takže pokud jej chceme provést, stačí zavolat funkci

```
glCallList(MUJ_KRUH);
```

2.7 Použité knihovny

Při tvorbě projektu byl kladen důraz na možnost multiplatformního použití. Použité knihovny jsou volně dostupné včetně zdrojových kódů.

2.7.1 OpenGL

2.7.1.1 Co je OpenGL

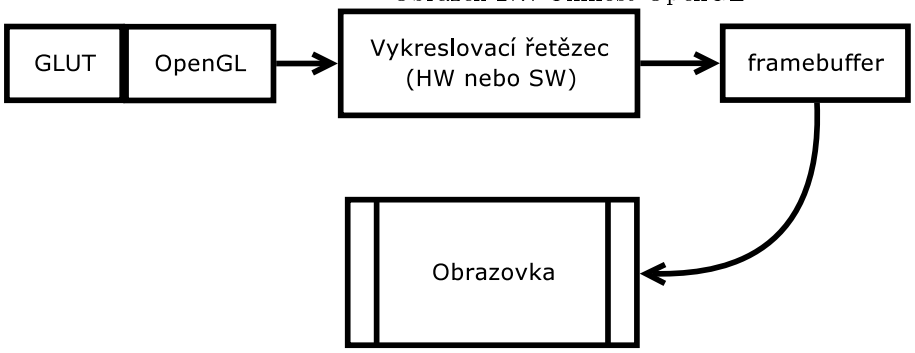
OpenGL je otevřená grafická knihovna (*Open Graphics Library*), která byla vytvořena firmou SGI (*Silicon Graphics Inc.*) jako API[4] k akcelerovaným grafickým kartám. Knihovna OpenGL byla vytvořena tak, aby se dala použít na různých typech grafických karet a aby ji bylo možno využívat také v případě, že na nějaké platformě žádný grafický akcelerátor nainstalován není. V tomto případě se použije softwarová simulace. Nyní je knihovna OpenGL použitelná na různých verzích UNIXových systémů (např: Linux), OS/2 a také ji lze použít na platformách systému Microsoft Windows.

Logo a název OpenGL^{TM} je registrovaná známka firmy Silicon Graphics Inc.

Knihovna OpenGL je vyvinuta tak, aby byla použitelná v jakémkoliv programovacím jazyce, přičemž primárně je k dispozici hlavičkový soubor pro jazyky C a C++. V tomto souboru jsou deklarovány nové datové typy, které využívá knihovna, některé konstanty a soubor také obsahuje obsahuje asi 120 příkazů, které specifikují objekty a operace potřebné pro vytváření interaktivních trojrozměrných aplikací. Takovéto hlavičkové soubory samozřejmě existují i pro jiné programovací

[4] Aplication Programming Interface = Aplikační programové rozhraní neboli rozhraní pro tvorbu aplikací

Obrázek 2.7: Činnost OpenGL



jazyky, například Object Pascal, Java nebo Fortran. Tyto soubory se většinou generují z hlavičkových souborů pro jazyk C.

OpenGL bylo vyvinuto a aby pracovalo efektivně i když počítač, který grafiku zobrazuje není počítačem, který grafiku vykresluje. Tato skutečnost může být důležitá v případě, že bychom měli náročnou grafickou aplikaci a několik počítačů vzájemně spojených do sítě. Tato aplikace tak může být zpracovávána celou sítí počítačů dohromady a zobrazována pouze ne jednom klientském stroji. Knihovna byla vyvinuta tak, aby byla naprosto nezávislá na použitých grafických ovladačích, operačním systému či na používaném Window Manageru[5], což je také důvod, proč knihovna neobsahuje žádné funkce pro práci s okny. Na podporu těchto funkcí je třeba použít buď přímo používaného správce oken (čímž ale přijdeme o naprostou kompatibilitu a nezávislost) nebo můžeme využít některou z nadstaveb. My budeme využívat nadstavbovou knihovnu GLUT, které se budeme věnovat dále.

2.7.1.2 Jak OpenGL funguje

Základní funkcí OpenGL je vykreslování do tzv. framebufferu[6]. Tento buffer uchovává veškeré informace, které potřebuje grafický akcelerátor pro zobrazení na obrazovce monitoru či LCD. Jsou zde uloženy informace o barvě a jasu každého pixelu na obrazovce. Činnost OpenGL programu znázorňuje obrázek .

Aby se dosáhlo ještě větší nezávislosti na využívané platformě, zavádí OpenGL vlastní datová primitiva (např: GLint nebo GLdouble).

2.7.2 GLUT

2.7.2.1 Co je to GLUT

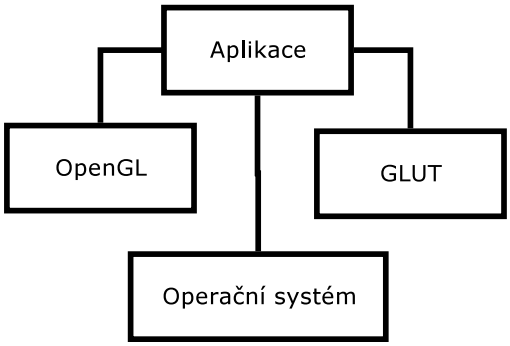
GLUT neboli *OpenGL Utility Toolkit* tvoří doplněk grafické knihovny OpenGL. Základem této nadstavby je hlavně podpora práce s písmem, vyskakovacími menu a také podpora pro práci s okny včetně zpracovávání jejich událostí.

GLUT je naprogramován v jazyce C a je kompatibilní se systémy Linux, Unix, OS/2 a také Microsoft Windows. Vzhledem ke skutečnosti, že je GLUT vytvořen tak, že z volání funkcí se vracejí pouze primitivní datové typy, je vcelku jednoduché vytvořit rozhraní pro další programovací jazyky, které mají podporu práce s dynamickými knihovnami. Tato ovládací rozhraní jsou již hotova pro jazyk Fortran, Object Pascal a také pro Python.

Hlavičkový soubor GLUTu se vetšinou vkládá přes příkaz include:

[5] správce oken
[6] obrazový rámec

Obrázek 2.8: Začlenění GLUT a OpenGL v aplikaci




```
#include <GL/glut.h>
```

Podpora GLUTu se spouští přes příkaz

```
glutInit();
```

Tento příkaz inicializuje podporu knihoven GLUTu a vytvoří relaci s používaným systémem oken. Při běhu tohoto procesu může docházet k chybovým hlášením, pokud nemohl být GLUT správně inicializován. Příkladem takovéto situace může být chybějící podpora OpenGL nebo chybné volání příkazové řádky.

2.7.2.2 Jak GLUT funguje

Knihovna GLUT začleňuje mezi knihovny systému a mezi vlastní aplikaci novou funkční vrstvu, takže z aplikace je možné volat nejen knihovny GLUTu, ale také knihovny OpenGL a knihovny operačního systému, což ukazuje obrázek . K zachování co možná největší kompatibility, kterou poskytuje jak OpenGL tak i GLUT je dosti nevhodné volat přímo systémové knihovny operačního systému. Díky skutečnosti, že se knihovna GLUT implementuje do výsledné aplikace tímto způsobem, lze také samozřejmě stále využívat standardních knihoven jazyka C, jimiž jsou například *stdlib*, *stdio*, *string* nebo *math*. Funkce zahrnuté v těchto knihovnách jsou v dnešní době bez problémů podporovány prakticky na všech platformách a jsou také popsány v normě jazyka C.

Pro celkové zjednodušení programování jsou rutiny GLUTu roztříděny do několika sub-API, které se rozdělují podle jejich funkčnosti. Tyto sub-API jsou:

- *Inicializace*
  - Zpracovávání příkazové řádky
  - Systémová inicializace oken
  - Vytváření počátečních oken a jejich řízení
- *Počáteční zpracovávání událostí*
  - Tyto rutiny vstupují do zpracovávacích událostí GLUTu
  - Tyto rutiny nikdy nic nevrací, pouze předávají volání návratovým funkcím GLUTu
- *Práce s okny*
  - Tyto rutina vytváří a ovládá okna

- *Práce s vrstvami*
  - Tyto rutiny zavádějí a řídí vrstvy pro okna
- *Práce s menu*
  - Tyto rutiny vytvářejí a ovládají pop-up menu
  - V těchto rutinách GLUT pouze využívá funkcionalitu použitého *Window Manageru*, takže se nemusí vždy jednat o pop-up menu (např.: pull-down) a také vzhled menu nemusí být jednotný
- *Registrace návratových funkcí*
  - Tyto rutiny zaregistrují návratové funkce pro zpětná volání GLUTu
- *Ovládání barevných map*
  - Tyto rutiny dovolují manipulaci s indexy barev v *colormapách* pro jednotlivá okna
- *Zjišťování stavu*
  - Tyto rutiny dovolují aplikaci přijímat stavové hlášení GLUTu
- *Renderování písma*
  - Tyto rutiny umožňují vykreslování čárových i bitmapových fontů
- *Renderování obrazců*
  - Tyto rutiny umožňují vykreslování 2D i 3D geometrických objektů včetně koulí, kuželů, dvacetistěnů a „čajových konvic“[7]

2.7.3 LibJPEG

LibJPEG je knihovna pro práci z obrázky ve formátu JPEG. V programu se tento formát používá pro načítání textur. LibJPEG je vlastně poměrně jednoduchý kód v C. Protože nevyužívá žádných služeb systému, měl by jít přeložit takřka kdekoliv.

Knihovna LibJPEG je použita ve zvláštní funkci pro načítání obrázků, aby bylo možné jednoduše přidat podporu pro jiné obrázkové formáty.

2.8 Použitý software

Veškerý použitý software je volně dostupný včetně zdrojových kódů pod licencí GPL nebo kompatibilní.

[7] v některých 3D grafických editorech se pro reprezentaci složitého objektu používá již předdefinovaný objekt „čajová konvice“

2.8.1 Vim

Vim (neboli Vi improved) je textový editor, který vychází z editoru Vi. Jeho autorem je Bram Moolenaar. Cílem tohoto editoru není vzhled ani uživatelská přívětivost ale efektivita práce. Proto je editor optimalizovaný tak aby uživatel při editaci textů stiskl co nejméně kláves a příliš nepřesouval ruce.

Vim je možné spouštět z textového terminálu, takže může pracovat například přes ssh nebo telnet. Další výhodou je snadná konfigurovatelnost a možnost přidávat makra pomocí konfiguračních souborů.

Vim byl použit pro editaci zdrojových kódů a jiných textových souborů.

2.8.2 CVS

Protože program byl vyvíjen ve dvou členném týmu, bylo potřeba mít k dispozici stále aktuální verzi zdrojových kódů. CVS (Concurrent Versions System) zajišťuje právě správu zdrojových kódů. CVS spravuje repozitář zdrojových kódů, ve kterém jsou uloženy všechny změny, ke kterým došlo. Vývojář si vždy stáhne aktuální (nebo jakoukoliv předešlou) verzi, upraví ji a nahraje zpět do repozitáře.

Pro účel našeho projektu byl zřízen repozitář pomocí služby cvsdude.com.

2.8.3 GCC

GCC (the GNU Compiler Collection) je sada kompilátorů patřící pod GNU. Verze pro Linux je přeložena pomocí g++ kompilátoru pro C++ z této sady.

2.8.4 GDB

GDB (the GNU project debuger) je program pro ladění programů. Umožňuje zastavit běžící program, krokovat ho po řádkách, číst a zapisovat data v paměti a zjišťovat důvody proč došlo k pádu programu. GDB se standardně ovládá pomocí své vlastní příkazové řádky. Pro některé úkony bylo použito grafické uživatelské rozhraní DDD, které umožňuje zobrazit graficky strukturu dat.

2.8.5 GNU Make

Make je program pro sestavování projektů z více částí. Sestavuje vždy soubor z jiných souborů. Make kontroluje čas změny souborů a vždy aktualizuje jen soubory, které jsou starší než soubory, na kterých závisí.

Používá se například při kompilaci programů. Jednotlivé zdrojové soubory zkompiluje na binární soubory, které následně linkuje do výsledného programu. Programátor tedy nemusí hlídat, které soubory změnil, ani se zdržovat kompilací souborů, které se nezměnily.

2.8.6 Autotools

Je sada nástrojů pro vytváření balíčků se snadno zkompilovatelnými zdrojovými kódy. Podrobnější popis v kapitole 3.7.1.

2.8.7 LyX

WYSIWYM[8] editor pro LaTeX. To znamená, že jde o něco mezi WISIWIG[9] editory, kde je vidět v reálném čase to samé co bude ve výsledku (např OpenOffice Writter) a mezi přímou editací zdrojového kódu LaTeXu pomocí textového editoru.

LaTeX je balík maker TeXu napsaný Leslie Lamportem, který představuje systém pro zpracování dokumentu. LaTeX dovoluje popsat strukturu dokumentu pomocí značkování tak, aby uživatel nebyl nucen přemýšlet o výsledném vzhledu [7].

TeX je sázecí systém vytvořený Donaldem E. Knuthem. TeX umožňuje pomocí maker popsat sazbu dokumentu.

LyX byl použit pro napsání tohoto dokumentu.

2.8.8 Ostatní

GIMP (GNU Image Manipulation Program) je program pro práci z rastrovou grafikou. V projektu byl použit pro tvorbu textur.

Blender 3D modelovací program. Pro projektu byl použit na vytváření modelů objektů.

Doxygen Program pro tvorbu dokumentace ze zdrojových kódů

Adobe Flash 8 Proffessional IDE pro tvorbu Flash aplikací

Dia Program pro tvorbu diagramů

DOT Program pro generování diagramů (např. obr. 3.6)

CodeBlocks IDE[10] pro jazyk C++. Byl použit pro práci s kódy ve windows.

[8] What You See Is What You Mean - vidíte to jak to myslíte
[9] What You See Is What You Get - vidíte to co dostanete
[10] Integrated Development Environment = Vývojové prostředí

Kapitola 3

Výsledky

3.1 Struktura

Datová struktura programu je nejlépe vidět na UML diagramu (obr. 3.1). Základem programu je třída Player (hráč).

3.1.1 Player

Třída Player obsahuje veřejné metody, které se volají z obsluhy událostí:

kamera zajistí vykreslení scény z pohledu hráče.

pohyb slouží k přepočtení polohy hráče.

mys slouží k ovládání pohybu.

rozmeryOkna volá se pří změně rozměrů vykreslovaného prostoru, zajišťuje správné poměry zobrazení a nastavuje reakce na pohyb myši[1].

Aby bylo možné zobrazovat scénu a vypočítávat kolize je součástí třídy Player třída Mapa, která obsahuje informace o geometrii prostředí ve kterém se hráč pohybuje.

3.1.2 Mapa

Třída Mapa obsahuje informace o scéně. Konstruktor třídy mapa má jediný parametr a to název souboru[2], ze kterého se mapa načítá. Mapa poskytuje pouze dvě veřejné metody:

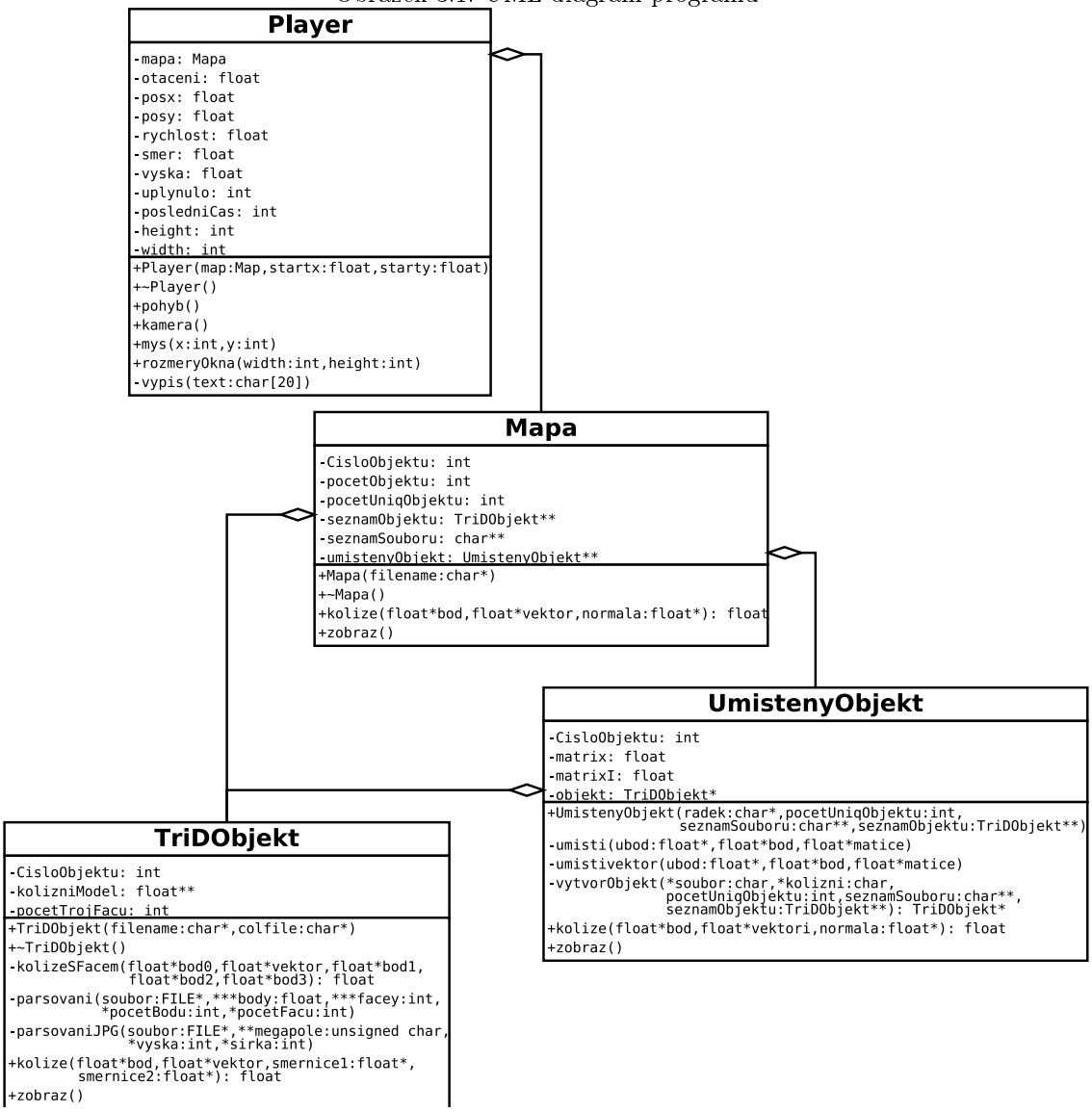
zobraz zobrazí scénu.

kolize slouží ke zjištění kolize přímky z mapou.

Informace o tvaru mapy jsou uloženy v poli umístěných objektů (třída UmistenyObjekt).

[1] Souřadnice myši se zasílají absolutně v pixelech, ovládání se ale odvíjí od relativního umístění kurzoru v okně.
[2] soubor je typu *.world který vznikl speciálně pro tento projekt

Obrázek 3.1: UML diagram programu



3.1.3 UmistenyObjekt

Třída UmistenyObjekt obsahuje transformační matici (i inversní matici) a ukazatel na třídu TriDObjekt.

Díky tomu že je třída UmistenyObjekt oddělena od třídy TriDObjekt je možné, aby více instancí třídy UmistenyObjekt obsahovalo ukazatel na tu samou instanci třídy TriDObjekt. Toho lze využít v případě, že mapa obsahuje na různých místech[3] tvarově stejné objekty.

3.1.4 TriDObjekt

Třída TriDObjekt obsahuje informace o tvaru objektu a textuře.

Konstruktor této třídy má dva parametry. První parametr - *filename* - je název souboru, ze kterého se má načítat výsledný objekt a druhý parametr - *colfile* - je název „kolizního“ objektu (kapitola 3.3).

parsuj Slouží pro nalezení potřebných informací o objektu v souboru formátu DirectX a jejich následnému zapsání do struktury programu tak, aby mohly být následně využity pro kompilaci display-listů.

parsujJPG slouží k rozparsování *.jpg souboru na jednotlivé barvy a jejich následné poskládání zpět do struktury programu pro další využití při texturování pomocí OpenGL .

kolize Počítá vlastní kolizi mezi polopřímkou a jednotlivými rovinami, ze kterých se objekt skládá.

3.2 Načítání a parsování objektů

Objekty se načítají jeden po druhém, tak jak jsou napsány v souboru mapy včetně jejich transformační matice, která je důležitá nejen pro výslednou pozici objektu, ale také pro jeho natočení nebo velikost.

Postup načítání informací je dán způsobem uložení dat v souboru s modelem pro DirectX (*.x) a nejlépe je vidět z vývojového diagramu (obr. **??**).

Nejprve se tedy načítají informace o meshi, tedy globálně o tvaru celého objektu. V souboru .x jsou uloženy nejprve informace o jednotlivých vertexech, tedy o bodech, každé stěny. Nejprve je zde údaj o tom, kolik budů bude daný objekt mít. Každý bod je pak reprezentován trojicí souřadnic v pořadí x, y, z. Následují informace o tom kolik má objekt stěn a z jakých bodů je každá stěna spojena.

Poté, pokud se u objektu vyskytuje, se načte název souboru textury, který se předá funkci *parsujJPG* .

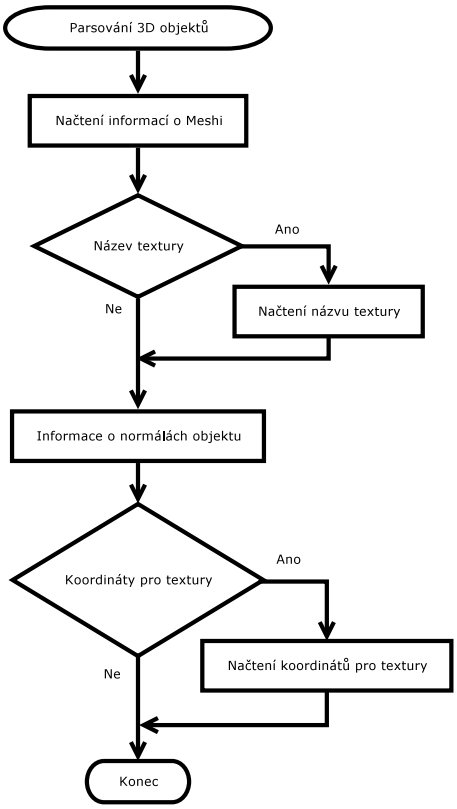
Následují informace o normálách objektu. Normály jsou vektory, které jsou kolmé na danou stěnu objektu. OpenGL tyto vektory využívá pro vykreslování při osvětlení a pro zaoblování objektů. Normála zpravidla bývá stejný počet jako faců.

Nakonec, opět pokud je objekt obsahuje, se načtou jednotlivé koordináty pro mapování textury. Koordinátů pro texturu je stejný počet jako počet vertexů. Každý koordinát je zde reprezentován souřadnicemi x a y. Souřadnice z se nepoužívá, protože se jedná jen o dvourozměrné textury.

Nakonec se vytvoří display-list (kapitola 3.7), ve kterém se uloží nejprve texturová data, pokud je objekt obsahuje. Tato data se musí vkládat na začátku, protože podle validní syntaxe nesmějí být mezi *glBegin* a *glEnd*. Následují dva výše zmíněné příkazy a mezi nimi se již cyklicky vypisují veškeré body (vertexy), normály a také texturové koordináty k jednotlivým facům.

[3] Nemusí se lišit pouze místem ale obecně transformační maticí (rotací, velikostí ...).

Obrázek 3.2: Parsování objektů



3.3 Kolizní model

Při výpočtu kolizí není potřeba model, který se skládá z velkého množství plošek (polygonů). Proto je možné použít jiný model pro výpočet kolizí než pro zobrazování. To má výhodu v rychlejším výpočtu kolizí. Nevýhoda je potřeba vytvořit zvlášť kolizní model. V případě jednoduchých objektů je možné načítat jako kolizní stejný model jako pro zobrazení.

Je možné zobrazovat polygony které mají více než tři body (například krychle složená ze čtverců). S těmito polygony je ale velmi obtížné počítat kolize. A tak se polygony pro kolizní model vždy převádí na tříbodové. To je provedeno tak, že se vybere první bod jako hlavní (je ve všech trojúhelnících) a pak se postupně přidávají další body (druhý v jednom trojúhelníku je první v dalším (obr. 3.3).

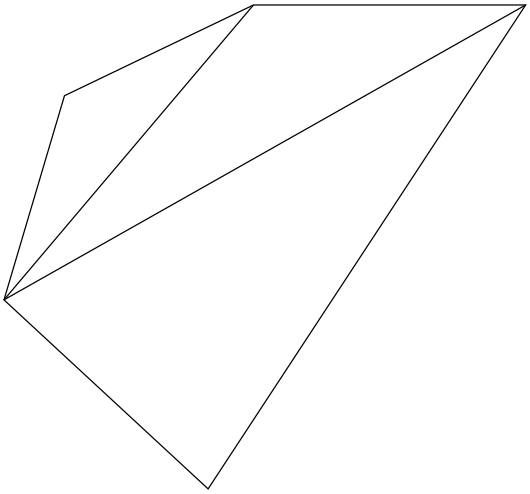
3.4 Výpočet kolizí

Výpočet kolizí začíná ve třídě Mapa, kde se postupně volá kolize v jednotlivých umístěných objektech. Pokud se zjistí kolize porovnává se vzdálenost a vrací se nejbližší.

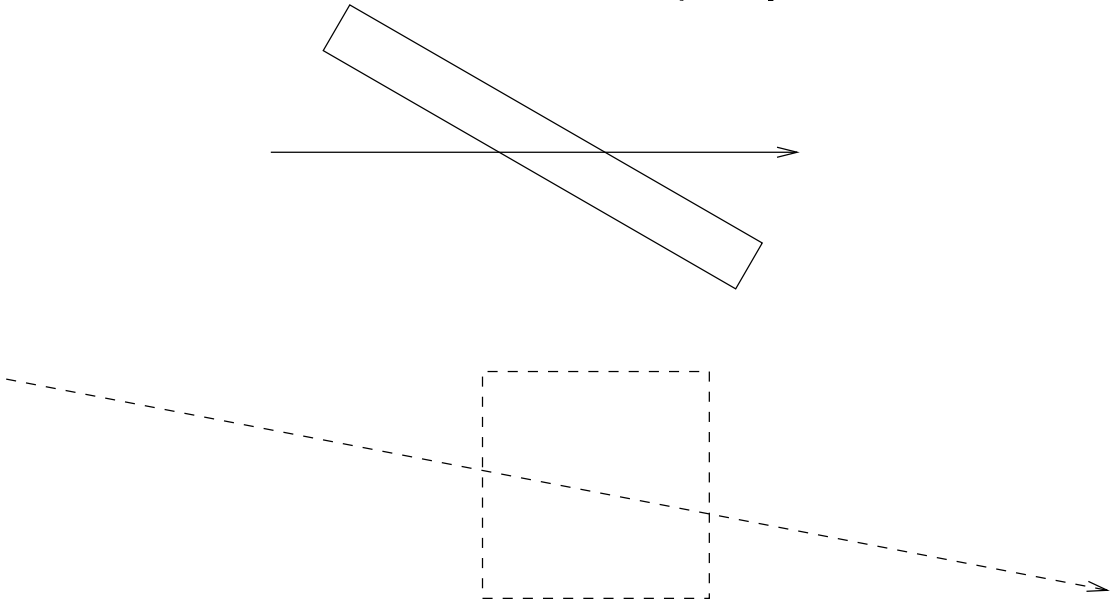
Třída UmistenyObjekt obsahuje ukazatel na třídu TriDObjekt, ve které je uložen tvar neumístěného objektu.

Aby bylo možné provést výpočet kolize mezi přímkou, která je určena globálními souřadnicemi a objektem, který je určen lokálními souřadnicemi, je potřeba provést lineární transformaci. Ta se však neprovádí, tak jak by to bylo na první pohled zřejmé přepočtem všech bodů objektu (stejně jako se provádí zobrazení) ale přepočtením přímky pomocí inverzní matice (kapitola 2.5.2). Takže

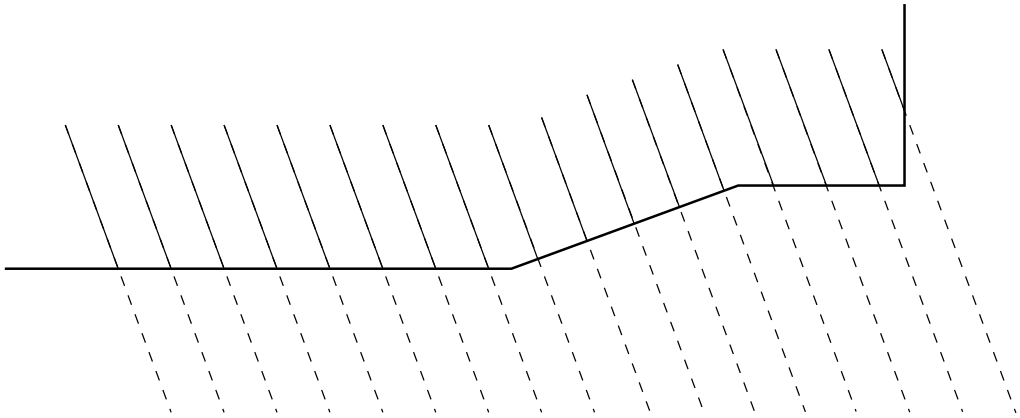
Obrázek 3.3: Převod na trojúhelníky



Obrázek 3.4: Kolize umístěného objektu s přímkou



Obrázek 3.5: Pohybující se hráč (pohled z boku)



ve výsledku se provádí výpočet kolize mezi přímkou a objektem v základním (neumístěném) tvaru (obr. 3.4).

Protože výsledek kolize, vzdálenost od počátku přímky k místu kolize, je určen v násobcích velikosti vektoru (parametr t v kapitole 2.3.1.4), není nutné přepočítávat výsledek zpět podle původní matice. Toto řešení má výhodu, že se provádí lineární transformace pouze pro jedem bod a vektor a ne pro desítky až stovky bodů ze kterých se skládá objekt.

Další možností by bylo ukládat souřadnice umístěných bodů. To by ale vyžadovalo více operační paměti, také by bylo mnohem složitější rozšíření o pohyblivé objekty.

Aby bylo možné snadno odlišit zeď od podlahy zjišťuje se současně s kolizí i normála[4] roviny. To se provádí pomocí skalárního součinu jak je popsáno v kapitole 2.2.2.2.

3.5 Pohyb

Pozice hráče je určena třemi souřadnicemi (pata hráče) a číslem určujícím směr otočení. Pohyb hráče se vyhodnocuje mezi každým překreslením snímků. Nejprve se podle stavu ovládání nastaví správné otočení hráče. Potom se vyhodnocuje kolize mezi přímkou a mapou.

Přímka je vedena z bodu nad místem kde hráč stojí (výška je nastavena pomocí direktivy VYSKAKROKU) šikmo dolů. úhel pod kterým přímka směřuje k zemi závisí na aktuální rychlosti hráče a rychlosti vykreslování. Pokud nedojde ke kolizi s neschůdným povrchem (rozeznává se podle úhlu normály), hráč se přesune na místo, kde došlo ke kolizí (obr. 3.5).

Kamera se vykresluje nad pozicí hráče (jak vysoko je určeno direktivou VYSKAOCI).

Tímto způsobem je možné zjišťovat kolizi se stěnou a zároveň zjišťovat výšku terénu.

3.6 Ovládání

Program se ovládá pomocí dvou proporcionálních[5] hodnot. Kvůli co největší kompatibilitě je nastaveno ovládání na myš. Jednou osou (zleva doprava) se ovládá rychlost otáčení a druhou osou rychlost pohybu vpřed (vzad). Protože jde v základu o závodní hru neovládá se přímo směr ale rychlost otáčení. Díky tomu je možné provádět veškeré pohyby bez zvednutí myši z podložky.

[4] kolmý vektor

[5] mají více možných stavů (z ovladačů např. myš, joystick, volant...)

Ukázka kódu 3.1: Pohyb hráče

```
bod[0]=-posx;
bod[1]=vyska+VYSKAKROKU;   //nastavení výchozího bodu přímky
bod[2]=-posy;
vektor[0]=sin(smer*PI/180)*rychlost*uplynulo/10;
vektor[1]=-VYSKAKROKU;   //nastavení vektoru ve směru pohybu šikmo dolů
vektor[2]=-cos(smer*PI/180)*rychlost*uplynulo/10;
float t=mapa->kolize(bod,vektor,normala);   //zjištení vzdalenosti kolidujícího bodu
float delka=sqrt(normala[0]*normala[0]+normala[1]*normala[1]+normala[2]*normala[2]);
normala[0]/=delka;
normala[1]/=delka;   //normalizace normaly (pro snažší určení úhlu)
normala[2]/=delka;
if ((normala[1]>STRMOST)||(normala[1]<-STRMOST)){ //jestli není stěna příliš prudká
    posx -=vektor[0]*t;   //posun na místo kolize
    posy -=vektor[2]*t;
    if ((VYSKAKROKU-VYSKAKROKU*t)<(-RYCHLOSTPADU*uplynulo)) //není-li příliš vysoko
        vyska -=RYCHLOSTPADU*uplynulo; //změnit výšku na místo kolize
    else
        vyska += (VYSKAKROKU-VYSKAKROKU*t);  //padat dolů
    };
```

3.7 Multiplatformnost

Program je možné přeložit na různých platformách beze změn ve zdrojovém kódu. Toho je docíleno použitím co nejmenšího počtu knihoven. Program používá pouze tři knihovny (kapitola 2.7), které všechny fungují na mnoha platformách.

Pro univerzální distribuci programu jsou nejvhodnější zdrojové kódy. Aby nebylo pro uživatele složité si zdrojové kódy přeložit používá se balíček vytvořený pomocí Autotools. Tento způsob distribuce je obvyklý hlavně v Unix-like[6] systémech. Ve Windows je potřeba pro přeložení balíčku potřeba funkční unixový shell[7]. Ten je možno získat v balíčku CYGWIN nebo MSYS. Proto je verze pro Windows přeložena zvlášť a pro distribuci je určena binární verze pomocí překladače ze sady Mingw.

Program byl testován na platformách Windows XP (home i proffessional) a Gentoo Linux.

3.7.1 Autotools

Instalace ze zdrojových kódů v Linuxu (případně jiných Unixových systémů) se provádí pomocí takzvané „svaté trojice“ což je posloupnost příkazů:

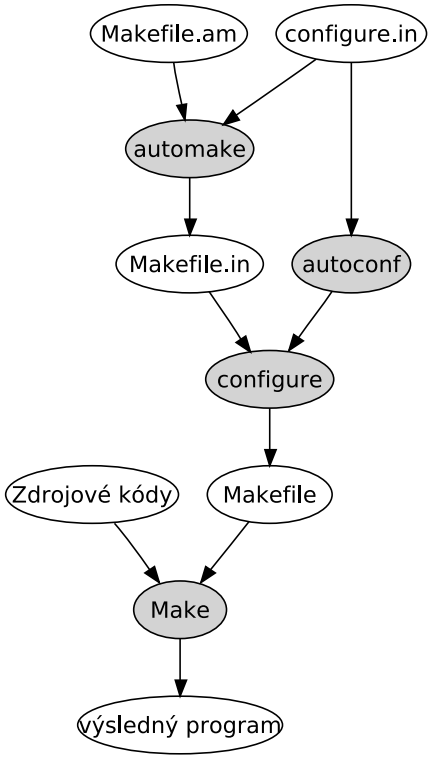
```
./configure
make
make install
```

Nejprve skript configure zjistí jestli je v systému vše potřebné pro kompilaci, poté vytvoří soubor Makefile obsahující informace pro program make. Make (kapitola 2.8.5) potom provede nejdříve vlastní kompilaci a následně i instalaci programu.

[6] systémy vycházející z Unixu

[7] Prostředí příkazové řádky. Umožňuje používat jednoduché programovací příkazy a tak psát skripty.

Obrázek 3.6: Tvorba balíčku pomocí Autotools



Skript configure musí zjistit jestli je v systému nějaký kompilátor, co lze použít, jestli zde jsou všechny potřebné knihovny a kde, jaké jsou systémové cesty a kam se program bude instalovat. Skriptu je možno předávat parametrem upřesňující informace. Dále musí skript podle zjištěných informací vytvořit Makefile srozumitelný pro make. Dále může generovat i hlavičkový soubor, který umožní automatické upravení zdrojového kódu podle systému.

Protože skript configure není jednoduchý ale bývá většinou pro různé projekty podobný, je vhodné tento skript generovat. To se provádí pomocí programu autoconf, který skript generuje podle souboru configure.in. Ten obsahuje už jednoduše formulovaná data o tom co má skript zjišťovat. Výsledný skript potom vyplní zjištěné informace do šablony Makefile.in.

Šablonu Makefile.in lze podobně jako skript configure generovat programem autoconf podle souboru Makefile.am. Soubor Makefile.am obsahuje pouze informace o názvu programu a seznam zdrojových kódů případně i seznam dalších souborů které je třeba distribuovat. Celý postup je vidět na obrázku 3.6.

Make nemusí vytvářet pouze výsledný program případně jej instalovat. Například příkazem „make dist“ se vytvoří archiv se vší potřebným pro kompilaci.

Kapitola 4

Závěr

4.1 Využití

Ačkoliv hlavním záměrem byla hlavně vizualizace architektury, umožňuje program mnohem širší využití. Protože hlavní myšlenkou je procházení prostředím, které lze snadno vytvářet, je možné použít program například jako základ pro počítačovou hru.

4.2 Rozšířitelnost

Program je tvořen objektově s ohledem na možnosti budoucího rozšíření. Například je možné poměrně snadno přidat podporu pro pohybující se objekty ve scéně nebo podporu jiných obrazových formátů.

Jednoduchým způsobem je také možné měnit základní účel projektu jak je popsáno výše.

Seznam obrázků

Seznam ukázek kódu

Literatura

[1] NEIDER, Jackie, DAVIS, Tom, WOO, Mason. OpenGL Programming Guide [online].Release 1. 1994 [cit. 2007-01-05]. Eng. Dostupný z WWW: <http://www.rush3d.com/reference/opengl-redbook-1.1>. ISBN 0-201-63274-8.

[2] OpenGL Reference Manual [online]. 1994 [cit. 2007-01-05]. Eng. Dostupný z WWW: <http://www.rush3d.com/reference/opengl-bluebook-1.0>. ISBN 0-201-63276-4.

[3] KILGARD, Mark J.. The OpenGL Utility Toolkit (GLUT) Programming Interface API [online]. Version 3. c1994-1996 [cit. 2007-02-10]. Dostupný z WWW: <http://www.opengl.org/resources/libraries/glut/spec3/spec3.html>.

[4] FERNANDES, António Ramires. 3D Maths for CG. Lighthouse3d.com [online]. 2005 [cit. 2007-02-17]. Dostupný z WWW: <http://www.lighthouse3d.com/opengl/maths>.

[5] TIŠNOVSKÝ, Pavel. Grafická knihovna OpenGL. Root.cz [online]. 2003 [cit. 2007-02-14]. Dostupný z WWW: <http://www.root.cz/serialy/graficka-knihovna-opengl>.

[6] KAŠPÁREK, Tomáš. GNU - pomoc při tvorbě programů [online]. 2001 [cit. 2007-02-13]. Dostupný z WWW: <http://www.root.cz/serialy/gnu-pomoc-pri-tvorbe-programu>.

[7] HUDEC, Tomáš, ŠKARVADA, Libor. často kladené otázky o TEXu a odpovědi na ně [online]. c1997-2003 , Poslední aktualizace: 26.07.2006 [cit. 2007-03-04]. Dostupný z WWW: <http://www.fi.muni.cz/cstug/csfaq>.

[8] OLŠÁK, Petr. Lineární algebra [online]. 2000-2006 [cit. 2007-03-05]. Dostupný z WWW: <http://math.feld.cvut.cz/olsak/linal.html>.

[9] VYTERNA, Štěpán. Maturitní práce. [s.l.], 2007. 26 s. Maturitní práce.

[10] ČERNÝ, Vladimír. Maturitní práce. [s.l.], 2007. 29 s. Maturitní práce.