**Vysoké učení technické v Brně**

Fakulta informačních technologií

# Ročníkový projekt

Daniel Výchopeň

2005

*Prohlašuji, že jsem tento ročníkový projekt vypracoval samostatně pod vedením Ing. Jana Pečivy. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.*

*Děkuji Ing. Janu Pečivovi za poskytnutí odborné pomoci při řešení projektu.*

V Brně dne ……  ……

Daniel Výchopeň

## Abstrakt a klíčová slova

## Abstrakt

Tato práce je zaměřena na vytvoření aplikace pomocí grafické knihovny Open Inventor. Zabývá se návrhem algoritmů pro jednoduchou 3D hru využívající zvuk, komunikaci po síti a fyziku těles. Důraz je kladen především na implementaci aplikace založené na navržených algoritmech. V samotné hře pak bude hráč ovládat pomocí klávesnice tank, se kterým bude jezdit po světě, který bude tvořen modelem bludiště. Cílem hry pak bude najít a zneškodnit nepřátelské tanky. Nepřátelské tanky budou ovládat hráči připojení přes síťové rozhraní. Ve hře jsou také řešeny kolize mezi tankem a bludištěm a vzájemně mezi tanky.

## Klíčová slova

Open Inventor, Coin3D, renderování, zvuk, kolize, síťová komunikace.

# Obsah

1 Úvod ..... 6
2 Vytvoření základní aplikace ..... 7
2.1 Vytvoření okna aplikace ..... 7
2.2 Ovládání pomocí klávesnice ..... 8
2.3 Vložení modelů do scény ..... 10
2.4 Pohyb tanku ve scéně ..... 12
3 Pokročilejší techniky ..... 14
3.1 Kontrola kolizí ..... 14
3.2 Animace střely ..... 17
3.3 Animace výbuchu ..... 19
3.4 Zvuky ..... 20
3.5 Síťová komunikace ..... 21
4 Závěr ..... 22

# 1 Úvod

V této práci se snažím popsat vytvoření aplikace demonstrující možnosti grafické knihovny Open Inventor. Konkrétně se jedná o 3D hru, ve které hráč jezdí s tankem po bludišti a snaží se vyhledat a zlikvidovat nepřátelské tanky. Nepřátelské tanky pak ovládají jiní hráči připojení přes síťové rozhraní.

Open Inventor je knihovna napsaná v C++ a je postavená nad OpenGL. Poskytuje rozsáhlé množství C++ tříd a posunuje tak programátora na vyšší úroveň, než je programování v OpenGL. Poskytuje tak vyšší komfort a umožňuje jednodušší a rychlejší tvorbu aplikací. Výsledná aplikace přitom může mít vyšší výkon než aplikace napsaná v OpenGL, protože Open Inventor umožňuje provádět nad daty scény optimalizace. Scéna v Open Inventoru se vytváří pomocí uzlů. Uzly mohou být různých typů. Můžou nést např. informace o geometrii tělesa, různých atributech a transformacích. Existují i uzly, které obsahují seznam jiných uzlů a pomocí nich pak lze vytvářet graf scény. V této práci používám knihovnu Coin [3], která je kompatibilní s Open Inventor API a je k dispozici pod GPL licencí. Existuje výborný tutoriál o Open Inventoru v češtině [1] a také kniha v angličtině [2].


*Obrázek 1.1: Vzhled aplikace*

# 2 Vytvoření základní aplikace

## 2.1 Vytvoření okna aplikace

Pro vykreslování okna v Open Inventoru se používají knihovny SoWin a SoQt. SoQt je okenní rozhraní pro aplikace založené na Qt a používá se pro zobrazování oken pod operačním systému Linux. SoWin je pak okenní rozhraní, které se používá pod operačním systémem Windows (32-bitové verze). Při inicializaci Inventoru tedy musíme rozlišit, jakou knihovnu vybrat. To zajistí následující kód, který tak rozhodne při kompilaci kódu podle toho, zda jsou definovány konstanty _WIN32 nebo __WIN32_ :

```
#if defined(_WIN32) || defined(__WIN32__)
    HWND window = SoWin::init(argv[0]);
#else
    QWidget *window = SoQt::init(argv[0]);
#endif
```

Dále vytvoříme kořen grafu scény. To se provede vytvořením objektu typu SoSeparator. SoSeparator je třída odvozená ze SoGroup, tedy od základní třídy udržující seznam jiných uzlů. Třídu SoSeparator používám místo SoGroup téměř vždy pro její speciální vlastnosti. Například proto, že dokáže scénu pod sebou předkompilovat do OpenGL display listu a tím urychlit proces renderování.

```
root = new SoSeparator;
```

Na vytvořený objekt se musí vytvořit reference. Každý objekt scény v Inventoru má totiž počítadlo referencí a pokud toto počítadlo klesne na nulu, objekt je automaticky uvolněn z paměti.

```
root->ref();
```

Nyní je vytvořen kořen grafu scény. Ten bude základem pro celou scénu. Pro vytvoření okna použijeme třídu SoWinRenderArea (SoQtRenderArea pro linux), která vytvoří jen jednoduché okno, do kterého se bude renderovat scéna. Aby Inventor věděl, z jakého směru a pozice scénu zobrazovat, musí se do scény vložit kamera. Tentokrát ale referenci na objekt `kamera` automaticky vytvoří metoda `addChild`.

```
kamera = new SoPerspectiveCamera;
root->addChild(kamera);
```

Ještě nastavíme přední a zadní ořezávací roviny u kamery. Objekty ležící mimo tyto roviny, nebudou zobrazeny.

```
kamera->nearDistance = 4;
kamera->farDistance = 4096;
```

Nyní můžeme vytvořit okno:

```
#if defined(_WIN32) || defined(__WIN32__)
  SoWinRenderArea *renderArea = new SoWinRenderArea(window);
#else
  SoQtRenderArea *renderArea = new SoQtRenderArea(window);
#endif
```

Nastavíme kořen grafu scény, titulek okna a povolíme zobrazení okna.

```
renderArea->setSceneGraph(root);
renderArea->setTitle("Tank Hunters");
renderArea->show();
```

Nyní necháme zobrazit okno a nastartujeme smyčku programu.

```
#if defined(_WIN32) || defined(__WIN32__)
    SoWin::show(window);
    SoWin::mainLoop();
#else
    SoQt::show(window);
    SoQt::mainLoop();
#endif
```

Tímto už je napsán veškerý kód, který je rozdílný pro operační systémy Windows a Linux. Na závěr programu se ještě přidají příkazy pro uvolnění paměti.

```
delete renderArea;
root->unref();
```

## 2.2 Ovládání pomocí klávesnice

Pro ovládání vytvářené aplikace bude sloužit klávesnice. Pomocí Coinu ale nemůžeme přímo zjistit, zda je nějaká klávesa právě stisknuta. Stisknutí a uvolnění klávesy vyvolá příslušnou událost. V Coinu si zaregistruji funkci `event_cb`, která bude tyto události zpracovávat:

```
SoEventCallback * cb = new SoEventCallback;
```

```
    cb->addEventCallback(SoKeyboardEvent::getClassTypeId(),
event_cb, NULL);
    root->insertChild(cb, 0);
```

Metoda `insertChild(cb, 0)` způsobí, že se volání funkce zařadí hned na začátek stromu grafu scény a nebude se tedy muset procházet celý strom při každém stisknutí nebo uvolnění klávesy.

Pro uložení stavu o stisknutých klávesách vytvoříme pole `klavesy`. Bude obsahovat položky třídy `Klavesa`, což je struktura, která má jako první prvek identifikátor klávesy. Druhý prvek je pak typ bool, který určuje, zda je klávesa právě stisknuta. Pole se bude indexovat pomocí výčtu, který je definován ve třídě `Klavesa` a přímo popisuje akci, ke které bude bude daná klávesa v aplikaci sloužit.

```
    struct Klavesa
    {
        enum {nahoru = 0, dolu, vlevo, vpravo, strela,
zvedniHlaven, sklonHlaven
        };
        SoKeyboardEvent::Key klavesa;
        bool stisknuta;
    } klavesy[]= {
                    {SoKeyboardEvent::UP_ARROW, false},
                    ...
                    {SoKeyboardEvent::N, false}
                };
```

Pokud se tedy budeme chtít dotázat, zda je stisknuta klávesa pro střelbu, učiníme tak příkazem `if(klavesy[Klavesa::strela].stisknuta)`. Pokud si přejeme, aby se provedla akce jen jednou při stisku klávesy, můžeme sami změnit stav klávesy na hodnotu `false`:

```
    klavesy[Klavesa::strela].stisknuta=false.
```

Vlastní funkce `event_cb` pak zjišťuje, zda se událost, která vyvolala spuštění funkce, rovná události stlačení nebo uvolnění některé klávesy z pole `klavesy` a podle toho pak mění stav kláves `stisknuta` na `true` nebo `false`.

```
    void event_cb(void * userdata, SoEventCallback * node)
    {
        const SoEvent * event = node->getEvent();
        for(int i=0; i<sizeof(klavesy); ++i)
```

```
        {
            if(SoKeyboardEvent::isKeyPressEvent(event, klavesy
[i].klavesa))
                klavesy[i].stisknuta=true;
            if(SoKeyboardEvent::isKeyReleaseEvent(event,
klavesy[i].klavesa))
                klavesy[i].stisknuta=false;
        }
        node->setHandled();
    }
```

Metoda `node->setHandled()` zde zajistí, že se událost označí jako zpracovaná a nebude tedy muset dál procházet celým grafem scény.

## 2.3 Vložení modelů do scény

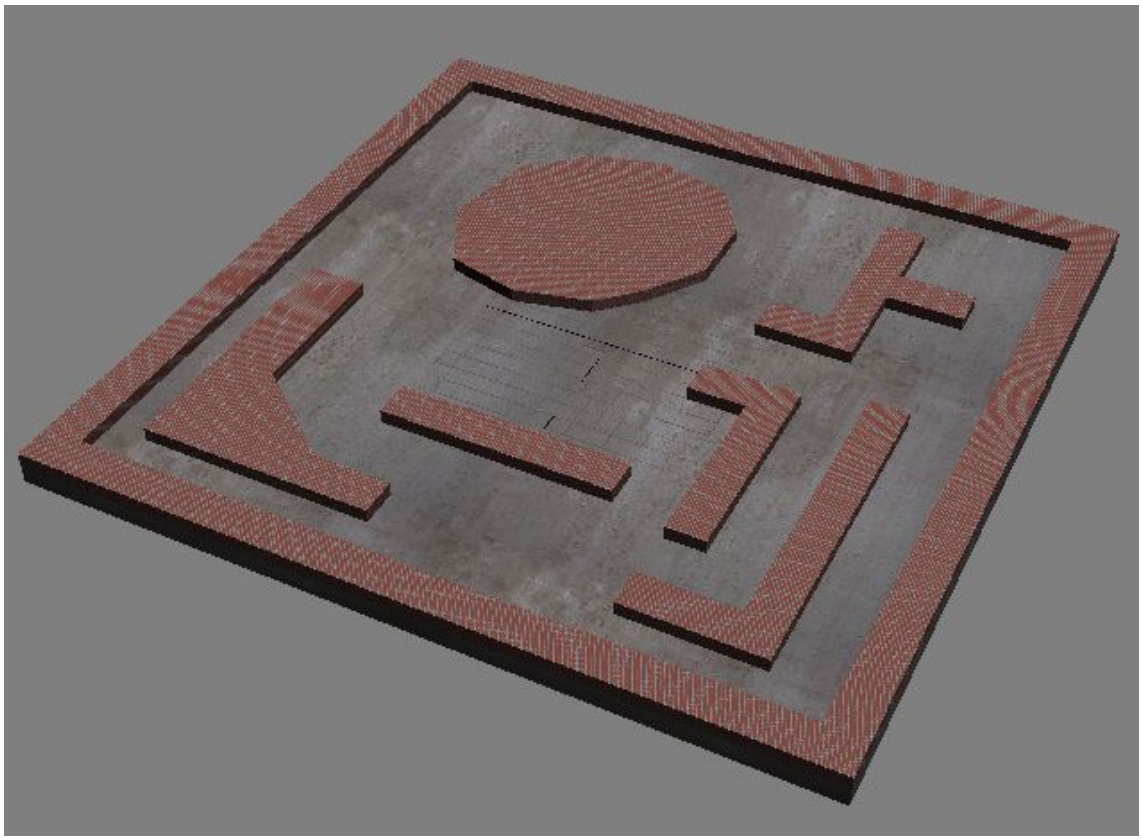
Ještě před vložením modelů do scény, je potřeba nastavit osvětlení, aby byly modely vůbec vidět. Můžeme tedy do scény vložit světla, která budou scénu osvětlovat. Použijeme směrové světlo, u kterého nastavíme směr záření a barvu.

```
SoDirectionalLight *svetlo1=new SoDirectionalLight;
svetlo1->direction.setValue(-1.0f, -1.0f, -1.0f);
svetlo1->color = SbVec3f(1.0f, 1.0f, 1.0f);
root->addChild(svetlo1);
```

Pokud nechceme světla používat, což může být výhodné například pokud nemáme definovány normály u modelů, stačí nastavit osvětlovací model tak, aby při zobrazování používal přímo difúzní barvu materiálu daného modelu:

```
SoLightModel *lmodel = new SoLightModel;
lmodel->model.setValue(SoLightModel::BASE_COLOR);
root->addChild(lmodel);
```

Nyní již můžeme do scény vkládat modely objektů. Inventor podporuje svůj vlastní formát souborů s příponou .iv ve verzi 2.0 a 2.1 v binárním i textovém tvaru. Dále podporuje formát VRML a sice jeho novější verzi VRML97/VRML2. Podpora pro všechny verze formátu 3ds je zatím ve vývoji. Nejvýhodnější z hlediska rozšířenosti, je tedy asi používat formát VRML, který podporují snad všechny rozšířené modelovací programy. Na druhou stranu je vhodné převést modely do binárního formátu Inventoru, čímž se výrazně urychlí načítání modelů a odstraní se tak nepříjemná prodleva při spouštění aplikace.



*Obrázek 2.1: Model světa bludiste.wrl*

Do grafu scény tedy vložíme model bludiste.wrl (Obrázek 2.1), což je model světa, ve kterém se budou pohybovat naše tanky:

```
SoFile *model = new SoFile;
model->name.setValue("models/bludiste.wrl");
root->addChild(model);
```

Pro tank si uděláme třídu `Tank`, do které zapouzdříme kořenový uzel tanku `SoSeparator *root`, model tanku `SoFile *model`, uzel pro posuv modelu tanku ve scéně `SoTranslation *trans`, uzel pro natáčení modelu tanku ve scéně `SoRotation *rotace` a další atributy jako například rychlost tanku, sklon hlavně a počet zlikvidovaných nepřátelských tanků. Dále pak tato třída bude obsahovat metody pro inicializaci tanku, nastavení pozice a natočení tanku, vrácení rychlosti tanku atpod.

Vlastní vložení modelu tanku do grafu scény pak provedeme v metodě `void init (SoSeparator *root, SoSeparator *rootKolize)`, kde první parametr je uzel grafu scény, ke kterému budeme připojovat kořenový uzel tanku. Druhý parametr bude sloužit

pro sestavení zjednodušené scény k řešení kolizí. V metodě `init` tedy vytvoříme kořenový uzel tanku a připojíme ho ke grafu scény:

```
this->root = new SoSeparator;
root->addChild(this->root);
```

Vytvoříme uzly pro posouvání a otáčení modelu tanku a připojíme je ke kořenovému uzlu tanku:

```
this->trans = new SoTranslation;
this->root->addChild(this->trans);
this->rotace = new SoRotation;
this->root->addChild(this->rotace);
```

A nakonec připojíme vlastní model tanku tank.wrl:

```
this->model = new SoFile;
this->model->name.setValue("models/tank.wrl");
this->root->addChild(this->model);
```

Teď můžeme vytvořit objekt `tank` třídy `Tank` a inicializovat ho.

```
tank.init(root, rootkolize);
```

## 2.4 Pohyb tanku ve scéně

Abychom mohli přepočítávat scénu v čase, musíme si vytvořit funkci, kterou nám bude Inventor spouštět po každém překreslení scény. K tomu využijeme třídu `SoOneShotSensor`, pomocí které si zaregistrujeme funkci `sensorCallback`:

```
SoOneShotSensor * sensor = new SoOneShotSensor
(sensorCallback, NULL);
```

A následně naplánujeme její spuštění:

```
sensor->schedule();
```

Funkce `sensorCallback` pak bude vypadat následovně:

```
void sensorCallback(void *data, SoSensor *sensor)
{
    ...
    sensor->schedule();
}
```

Na konci těla funkce je příkaz `sensor->schedule()`, kterým tak naplánujeme další spuštění této funkce. Inventor pak funkci zavolá hned jak bude moci (po překreslení scény,

obsloužení klávesnice, atpod.).

Na začátek této funkce pak přidáme kód, který nám zjistí, kolik času uběhlo od předchozího volání funkce:

```
static double casNovy=SbTime::getTimeOfDay().getValue(),
    casStary;
casStary=casNovy;
casNovy=SbTime::getTimeOfDay().getValue();
double uplynulyCas=casNovy-casStary;
```

Dále zde umístíme volání funkce `prepocitejScenu`, ve které bude kód pro přepočítání scény.

```
prepocitejScenu(casNovy, uplynulyCas);
```

Do funkce `prepocitejScenu` už pak můžeme vložit kód pro rozpohybování tanku. Podle stisknutých kláves vypočítáme novou rychlost. Pokud je například stisknuta klávesa pro pohyb tanku dopředu, vypočítá se nová rychlost podle vzorce:

$$novaRychlost = Rychlost + zrychleniTanku * uplynulyCas$$

Rychlost tanku musí být samozřejmě omezena, k tomu slouží konstanty `maxv` a `minv`, kde `minv` má zápornou hodnotu a znamená tedy maximální rychlost při couvání tanku. Pokud není stisknuta klávesa pro pohyb dopředu, ani pro pohyb dozadu, rychlost tanku se zpomalí podle konstanty `zpomaleni`.

Když máme vypočítanou novou rychlost tanku, můžeme následně vypočítat dráhu ujetou tankem za přírůstek času:

$$s = (Rychlost + novaRychlost)/2 * uplynulyCas$$

Podle směru tanku se pak vypočítají přírůstky v jednotlivých osách a tankem se posune na nové místo. Podobným způsobem je řešeno otáčení tanku a sklápění hlavně tanku.

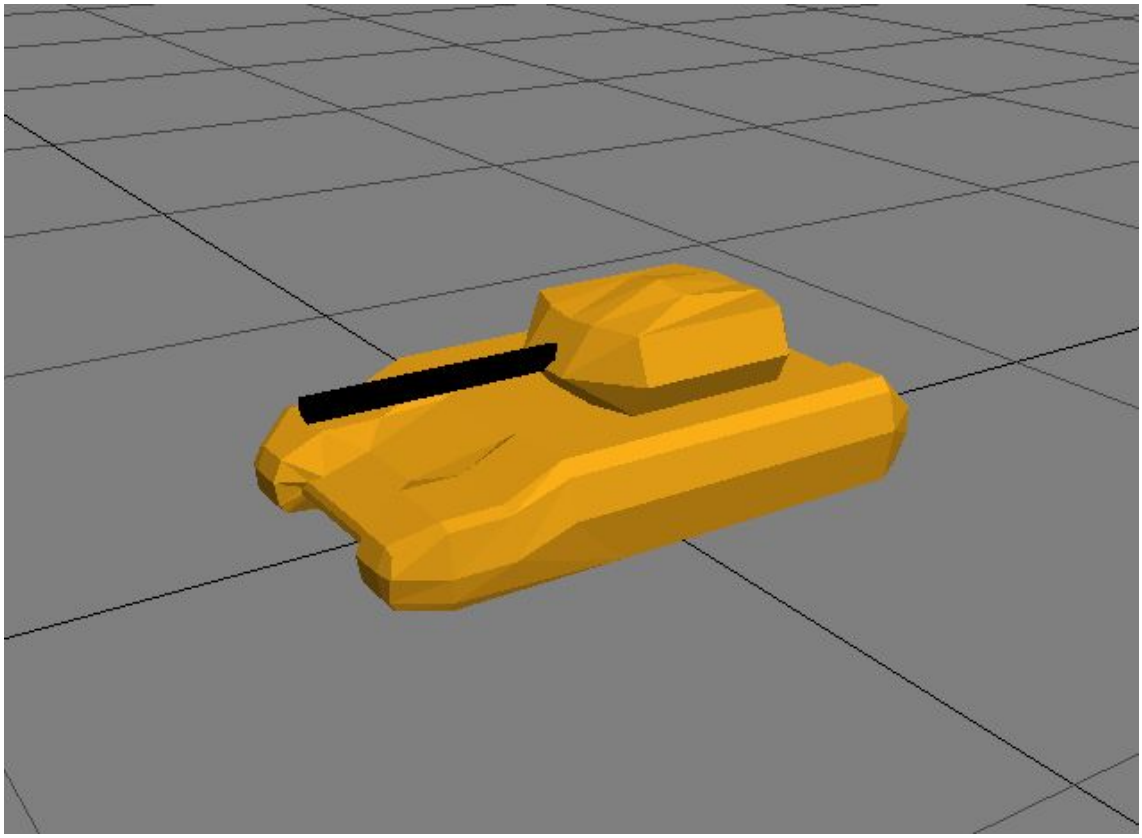
Nakonec nastavíme ještě pozici a směr kamery. Kamera bude umístěna kousek za tankem, ale při couvání se trochu oddálí, aby bylo lépe vidět na cestu. Pro otočení a sklopení kamery si vytvoříme dva objekty třídy `SoSFRotation`. Požadovaný směr kamery pak získáme vynásobením transformačních matic těchto objektů:

```
otockameru.setValue(SbVec3f(0, 1, 0), natoceni);
sklopkameru.setValue(SbVec3f(1, 0, 0), sklopeni);
kamera->orientation=sklopkameru.getValue()*
otockameru.getValue();
```

# 3 Pokročilejší techniky

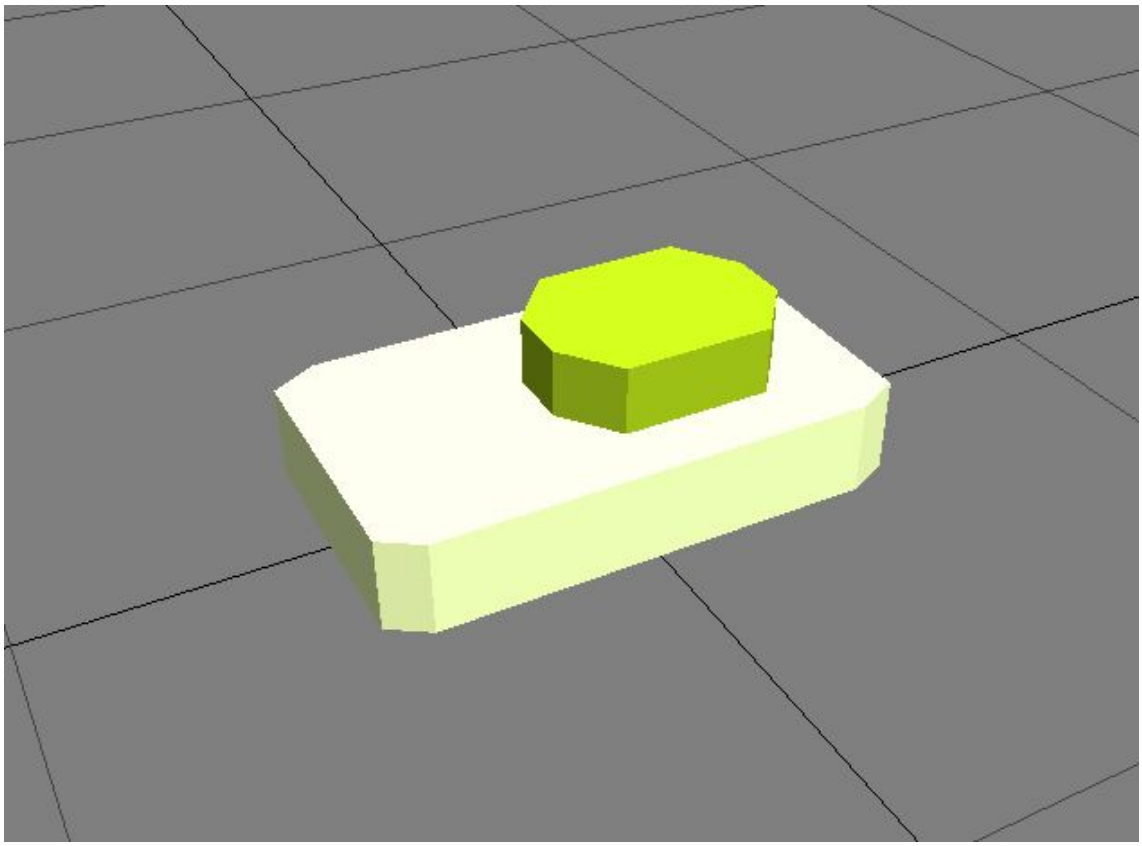
## 3.1 Kontrola kolizí

Výpočet kolizí je výpočetně náročný. Z toho důvodu si vytvoříme ještě jednu scénu, ve které však budou zjednodušené modely. Například místo složitého modelu tanku s mnoha trojúhelníky (Obrázek 3.1), použijeme zjednodušený model (Obrázek 3.2).



*Obrázek 3.1: Model tanku bez textur*

Při počítání kolizí Inventor zabalí objekty do obalových těles (kvádrů) a nejprve zjišťuje, zda nastala kolize mezi nějakým obalovým tělesem. Teprve až taková kolize nastane, tak počítá kolize mezi trojúhelníky těchto objektů. U modelu bludiště (Obrázek 2.1) je stěna která tvoří hranici bludiště tvořena jedním objektem. Celá stěna se pak zabalí do jednoho obalového tělesa, které bude vždy kolidovat s obalovým tělesem tanku jezdícím uvnitř bludiště. To pak zbytečně zatěžuje procesor a pro scénu pro počítání kolizí proto upravíme model tak, aby byla okrajová stěna tvořena čtyřmi samostatnými objekty (Obrázek 3.3).



*Obrázek 3.2: Zjednodušený model tanku pro výpočet kolizí*

Pro výpočet kolizí mezi objekty grafu scény slouží v Inventoru třída `SoIntersectionDetectionAction`. Umožňuje zaregistrovat funkci `filterCB`, což je filtr objektů mezi kterými se zpracovávají kolize. Dále pak funkci `intersectionCB`, která je zavolána vždy, když nastane kolize.

```
ida = new SoIntersectionDetectionAction;
ida->setFilterCallback(filterCB, NULL);
ida->addIntersectionCallback(intersectionCB, NULL);
```

Do těla funkce `filterCB` pak vložíme kód, který zaručí, že se nebudou počítat kolize mezi objekty vlastního bludiště, ale jen mezi tankem a bludištěm.

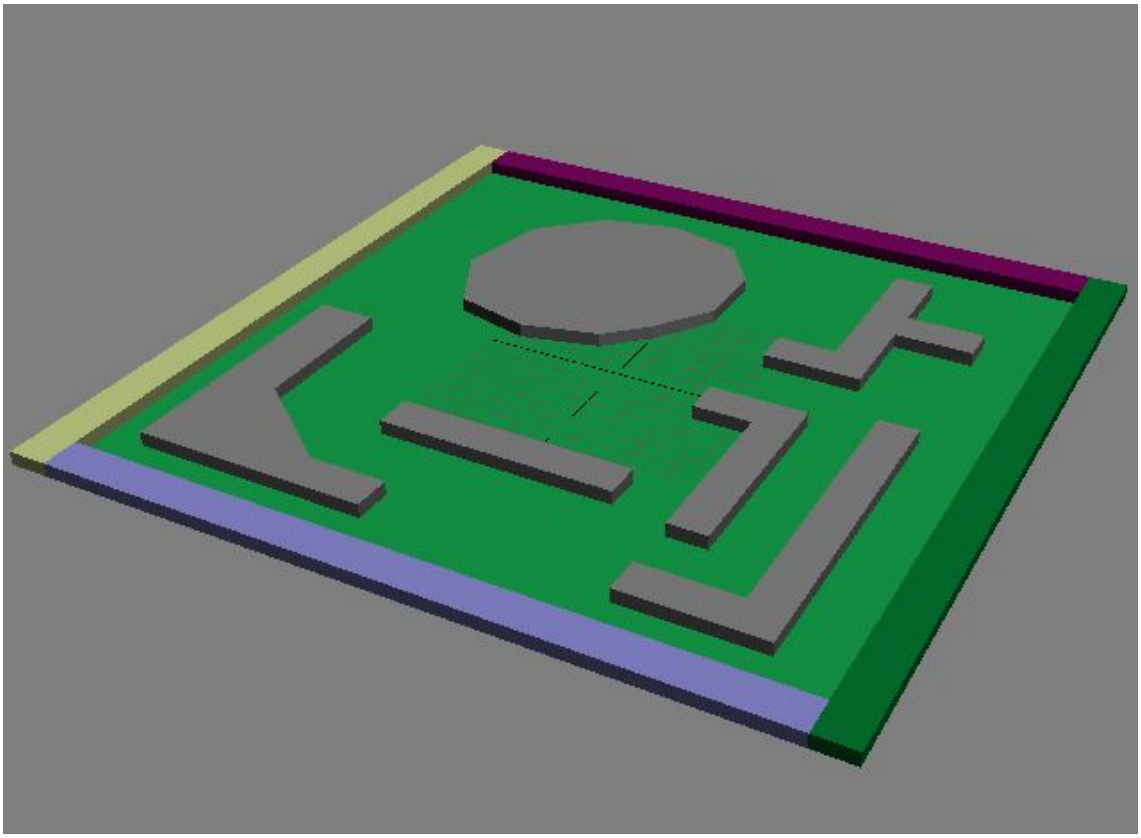
Ve funkci `intersectionCB` pak vypočítáme normálu trojúhelníku stěny, který koliduje s tankem. K tomu nám pomůže objekt plocha třídy `SbPlane`:

```
plocha=new SbPlane(pr1->vertex[0], pr1->vertex[1], pr1-
>vertex[2]);
```

```
normala=plocha->getNormal();
```

Pomocí třídy `SbRotation` zjistíme úhel, který svírá normála stěny a směr tanku:

```
SbRotation rot(normala, tank.smer);
SbVec3f axis;
float angle;
rot.getValue(axis, angle);
```



*Obrázek 3.3: Model bludiště pro počítání kolizí*

Jelikož tank může zároveň kolidovat s více trojúhelníky stěny, vybereme jen ten největší úhel mezi normálou stěny a směrem jízdy tanku. Tak docílíme správného klouzání tanku po zdi i tam, kde se zeď láme. Pomocí tohoto úhlu pak vypočítáme úhel, o který se bude měnit směr jízdy tanku. Znaménko tohoto úhlu určíme z vektoru `axix`, který jsme získali pomocí třídy `SbRotation` při výpočtu úhlu `angle`. Vektor `axix` bude vždy směřovat buď nahoru nebo dolů a ypsilonová složka bude mít hodnotu 1 nebo -1.

```
if(angle>tmpangle && angle>M_PI/2.0)
{
```

```
        tmpangle=angle;
        axis.getValue(x,y,z);
        tank.KorekceUhlu=(float)(-y*(angle-M_PI/2.0f));
    }
```

Při pohybování tankem ve funkci `prepocitejScenu` budeme postupovat následovně: Tank natočíme a zjistíme zda došlo ke kolizi ve scéně. Pokud ano, tak vrátíme původní natočení tanku. Následně tankem posuneme a opět zkontrolujeme kolize. Pokud tank koliduje, posuneme ho zpět, snížíme rychlost vlivem tření o zeď, vypočítáme novou ujetou dráhu a pokusíme se tankem posunout ve směru, který bude tentokrát změněn o úhel `tank.KorekceUhlu`.

## 3.2 Animace střely

Pro animaci střely vytvoříme třídu `Strela`, ve které bude zapouzdřen model střely, metody pro odstartování střely, vypočítání nové pozice střely a pro odstranění střely ze scény. Při inicializaci střely vytvoříme nový uzel `rodic` typu `SoSeparator`, který připojíme ke scéně.

```
this->rodic=new SoSeparator;
root->addChild(this->rodic);
```

Dále vytvoříme uzel `this->root` taktéž typu `SoSeparator`. Ten bude reprezentovat naši střelu. Uděláme na něho referenci, a k uzlu `rodic`, tedy ke scéně, ho budeme připojovat metodou `rodic->addChild(this->root)`, jen pokud budeme chtít model střely zrovna zobrazit ve scéně.

```
this->root = new SoSeparator;
this->root->ref();
```

K uzlu `this->root` pak připojíme uzly sloužící k otočení a posunutí modelu střely a uzel reprezentující vlastní model střely `strela.wrl`. Při inicializaci si ještě uložíme ukazatel na třídu `Tank`, protože každá střela bude přiřazena k jednomu tanku.

Střela se odstartuje metodou `start`. V této metodě se nastaví příznak aktivity střely a střela se vloží do scény:

```
this->aktivni=true;
this->rodic->addChild(this->root)
```

Uloží se rychlost střely, čas spuštění a úhel, pod kterým byla střela vypuštěna. Dále se pak vypočítá počáteční pozice střely. Při výpočtu se vychází z pozice tanku, střela se ale posune tak, aby vycházela z hlavně tanku.

Po dobu života střely se bude ve funkci `prepocitejScenu` volat metoda `refresh`,

která vypočítá novou polohu střely v závislosti na aktuálním času (počítá se jako šikmý vrh):

$$y = původníY + čas.(rychlost.\sin(sklon)) - 0{,}5\ .čas.gravitace$$

$$s = rychlost.čas.\cos(sklon)$$

kde *y* je vertikální poloha střely a *s* je horizontální vzdálenost od počáteční pozice střely, kterou musíme ji promítnout do složek x a z v závislosti na směru střely.

Život střely se ukončí pomocí metody `stop`, která zruší příznak aktivity střely a odpojí střelu z grafu scény. Objekt střely se přitom neodstraní z paměti, protože při inicializaci jsme vytvořili referenci `this->root->ref()`.

```
this->aktivni=false;
this->rodic->removeChild(this->root);
```

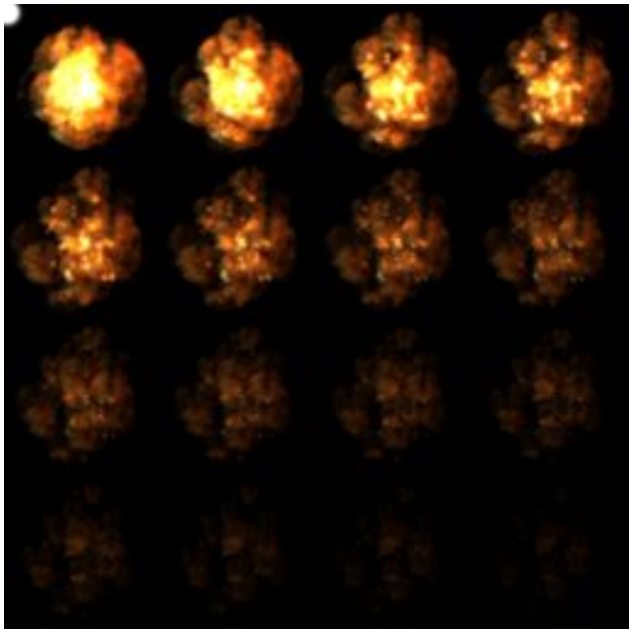
Pro zjištění, do kterého objektu střela narazila, si vytvoříme pomocnou třídu `StrelaKolize`. Ta nám poskytuje metodu `test(SbVec3f bod1, SbVec3f bod2)`, které zadáme aktuální a předcházející pozici střely. Pomocí těchto údajů vytvoří trojúhelník v grafu scény určené pro počítání kolizí. Třetí souřadnici trojúhelníku přitom vypočítá jednoduše z `bod2` tak, že jen o malý kousek sníží ypsilonovou souřadnici tohoto bodu. Vlastní trojúhelník pak vytvoří pomocí tříd Inventoru `SoCoordinate3` a `SoTriangleStripSet`. Když je trojúhelník vytvořený, můžeme zavolat kontrolu kolizí (`ida2->apply(this->rootKolize)`) a zjistit tak, zda střela reprezentovaná trojúhelníkem, nekolidovala s nějakým objektem. Ke kontrole kolizí je opět použita třída Inventoru `SoIntersectionDetectionAction`. Jen je použita jiná funkce pro filtrování objektů (`filterCB2`) a jiná funkce, která je volána při kolizi (`intersectionCB2`). Ve funkci filtru je třeba odfiltrovat počítání kolizí vzájemně mezi zdmi a mezi střelou a tankem který ji vyslal. Dále si zde zapamatujeme, jestli se bude testovat nepřátelský tank. V těle funkce `intersectionCB2` se pak dotážeme zda nastala kolize s nepřátelským tankem, pokud ano, tak zvýšíme počítadlo zásahů tanku. Případně můžeme zvýšit poškození nepřátelského tanku. Také v této funkci nastavíme příznak indikující, že nastala kolize. Podle něho pak víme, že máme ukončit život střely a zobrazit animaci výbuchu.

Ve funkci `prepocitejScenu` pak zkontrolujeme, zda je střela aktivní (`ifstrela.jeAktivni()`)) a pokud ano, tak provedeme výpočet nové pozice střely (`strela.refresh(casNovy)`) a kontrolu kolizí (`strelaKolize.test(pozice1, pozice2)`) . Pokud nastane kolize, nebo ypsilonová složka bude menší než nula (střela se dostane pod úroveň podlahy), ukončíme život střely a na místě předposlední pozice střely zobrazíme billboardovou animaci výbuchu. Používáme zde předposlední pozici proto, aby byla animace viditelná. Při použití poslední pozice střely, by totiž mohla být animace zobrazena uvnitř objektu, se kterým kolidovala.

## 3.3 Animace výbuchu

Pro animaci výbuchu vytvoříme třídu `Vybuch`. Ta zobrazí čtverec v místě výbuchu, vytvořený ze dvou trojúhelníků a potažený texturou (obrázek 3.4). Čtverec bude vždy natočený směrem ke kameře. Toho docílíme jednoduše tak, že uzlu výbuchu `rotace` třídy `SoRotation` přiřadíme v metodě `refresh` objekt `otoceni` třídy `SoSFRotation`, ten získáme z objektu `kamera`: `kamera->orientation`, při volání metody `refresh`. Texturovací koordinátory se pak budou měnit v závislosti na čase a vytvoří se tak animace. Aby nebylo vidět černé pozadí, použijeme třídu Inventoru `SoTransparencyType`, která slouží k nastavení průhlednosti:

```
this->transparencyType=new SoTransparencyType;
transparencyType->value=SoTransparencyType::DELAYED_ADD;
this->rootBillboard->addChild(this->transparencyType);
```



*Obrázek 3.4: Textura výbuchu*

Volba `DELAYED_ADD` zde znamená, že se zobrazení objektu provede až v druhém renderovacím průchodu (až po všech objektech které nejsou `DELAYED`) a bez zápisu do hloubkového bufferu. Nová barva pixelu se přitom určí jako nynější barva pixelu plus barva zdroje, násobená alfa hodnotou zdrojového pixelu.

Na vygenerování textury výbuchu jsem použil Explosion Generator [4], který ale bohužel negeneruje textury s alfa kanálem. Při použití takové textury, pak bylo vidět černé pozadí. Proto

jsem texturu editoval grafickým editorem tak, že jsem zprůhlednil levý horní roh textury (obrázek 3.4) a uložil jsem ji ve formátu png. Takto upravená textura se už zobrazuje správně.

Místo použití volby `DELAYED_ADD` by bylo vhodnější použít volbu `DELAYED_BLEND`, ale potřebovali bychom vygenerovat texturu i s alfa kanálem. S volbou `DELAYED_ADD` se totiž barvy sčítají a černý dým výbuchu pak není vůbec vidět.

## 3.4 Zvuky

Pro přehrávání zvuku ve scéně slouží v Inventoru třídy `SoVRMLAudioClip` a `SoVRMLSound`. `SoVRMLAudioClip` slouží k načtení a uchování audio dat. `SoVRMLSound` pak slouží k reprezentaci zdroje zvuku. Instance těchto tříd vložíme na místo ve scéně, ze kterého má vycházet zvuk :

```
this->clip = new SoVRMLAudioClip;
this->sound = new SoVRMLSound;
this->root->addChild(clip);
this->root->addChild(sound);
```

Zdroji zvuku přiřadíme zdroj audio dat:

```
this->sound->source = clip;
```

A nastavíme parametry tak, aby se nám zvuk nepřehrál hned po startu aplikace:

```
clip->startTime=0.0;
clip->stopTime =0.1;
```

Upravíme rychlost přehrávání (proměnnou rychlost přehrávání využívám při přehrávání zvuku motoru, kdy se zvyšuje rychlost přehrávání v závislosti na rychlosti tanku) a hlasitost:

```
clip->pitch=0.3f;
sound->intensity=1.0;
```

A přiřadíme soubor, ze kterého se budou načítat audio data:

```
this->clip->url="sound/vybuch.wav";
```

Pro načítání dat z wav souborů ale musíme mít zkompilovánu knihovnu simage (součást Coin3D) s podporou knihovny libsndfile [6].

Přehrání zvuku pak zajistíme nastavením `startTime` na aktuální čas:

```
clip->startTime = cas;
```

Zdroje zvuku se ve scéně pohybují, pohybuje se ale i příjemce zvuku. Vložíme proto do scény objekt třídy `SoListener`:

```
posluchac = new SoListener;
```

```
root->addChild(posluchac);
```

Pozici a natočení posluchače nastavíme na místě, kde se nastavuje kamera:

```
posluchac->orientation=otockameru;
posluchac->position=tank.getPosition();
```

Inventor přehrává zvuk přes rozhraní OpenAL [5]. Podpora zvuku ale bohužel není v Inventoru ještě úplně dořešená. Pod operačním systémem Linux se mi zatím nepodařilo zvuk přehrát, ale snad se mi to ještě podaří. Na platformě Windows zvuky přehrávat jdou, jen na jednom počítači se mi hra vždy po přehrání zvuku trochu zadrhla. Toto zadrhávání jde odstranit použitím jiného zařízení pro přehrávání zvuku. Místo standardně nastaveného "DirectSound3D", jsem proto použil jen obyčejný "DirectSound". Tímto nastavením se ale už zvuk nebude přehrávat prostorově (na čtyřreproduktorových sestavách), ale jen jako dvoukanálový zvuk. Nastavení zařízení se provede pomocí `SoAudioDevice`:

```
SoAudioDevice * AudioDevice = SoAudioDevice::instance();
AudioDevice->init("OpenAL", "DirectSound");
```

Pomocí `AudioDevice` pak můžeme nastavit i globální hlasitost zvuků:

```
AudioDevice->setGain(1.0);
```

Vzdálenost od zdroje, do jaké lze slyšet zvuk, je pevně daná (lze sice nastavit, ale Inventor toto nastavení zatím nepodporuje). Proto, aby bylo slyšet i vzdálenější zvuky od posluchače, je třeba na začátek grafu scény vložit uzel třídy `SoScale`, kterým zmenšíme celou scénu:

```
SoScale*scale=new SoScale;
root->addChild(scale);
float s=.02f;
scale->scaleFactor.setValue(s,s,s);
```

## 3.5 Síťová komunikace

Pro síťový kód jsem využil knihovnu TNet. Ta umožňuje vytvořit spojení mezi počítači pomocí IP adresy a čísla portu. IP adresa a číslo portu jsou uloženy v souborech ip.cfg a port.cfg, ze kterých se načítají při spouštění aplikace. Jeden z počítačů musí být server. Ten poslouchá na portu a čeká až se někdo připojí:

```
connectionListen=tnListen(port);
connection=tnAccept(connectionListen);
while(connection<=0)
    connection=tnAccept(connectionListen);
```

Klient se naproti tomu pokouší na danou adresu a port připojit (pomocí funkce `tnConnect`). Když je spojení ustaveno, je hra spuštěna a počítače si mezi sebou přenáší data, jako je poloha tanku, zásah tanku apod. Všechna data jsou uložena do pole `sitData` typu `float` a odeslána funkcí `tnSend`.

Na závěr ještě uvedu výčet některých funkcí z knihovny TNet:

- `void tnSend(TNConnection c, const void *buf, unsigned int bufsize)` – slouží k odesílání dat
- `int tnRecv(TNConnection c, void *buf, int bufsize)` – slouží k přijímání dat
- `TNConnection tnConnect(ip_t remoteIP, port_t remotePort, port_t localPort)` – slouží k navázání spojení se vzdáleným počítačem
- `TNConnection tnListen(port_t port)` – slouží k poslouchání na portu
- `TNConnection tnAccept(TNConnection lc)` – akceptuje síťové spojení
- `void tnDisconnect(TNConnection c)` – rozpojí síťového spojení
- `void tnClose(TNConnection c)` – uzavře síťového spojení
- `enum TNState tnGetState(TNConnection c)` – zjistí v jakém stavu se nachází síťové spojení
- `bool_t tnStr2IP(const char *s, ip_t *ip)` – převede IP adresu z řetězce na číslo

# 4 Závěr

V této práci jsem předvedl, jak lze pomocí grafické knihovny Open Inventor poměrně rychle a jednoduše vytvořit jednoduchou 3D hru. Bylo ukázáno jak lze pomocí Open Inventoru ve scéně řešit kolize objektů, a to bez použití speciálních editačních nástrojů na tvorbu modelů. Kontrola kolizí objektů zde pracuje na modelech vytvořených v běžných grafických editorech, které umožní export do formátu VRML. Ukázal jsem, jak lze ozvučit scénu. Bohužel je však podpora zvuku v knihovně Coin ještě v experimentální fázi a nemusí být tedy vždy plně bezproblémová na všech platformách. Nastínil jsem také, jak může aplikace komunikovat přes síťové rozhraní. Moje práce tedy může sloužit jako návod a inspirace pro rozsáhlejší projekty.

Další vývoj projektu by se mohl týkat např. vytvoření uživatelsky přívětivého a efektního menu, vytvoření modelu poškození tanků nebo přidání různých grafických efektů jako je např. mlha, déšť, nebo osvětlení scény z reflektoru tanku. Mohl by se také rozšířit zbrojní arzenál tanků (tepelně naváděné rakety, miny) a přidat různé bonusy (oprava tanku, vylepšení odolnosti, násobič poškození, urychlení tanku). Zlepšit by se také mohl síťový kód, aby mohlo hrát zároveň více hráčů. Také by se mohli přidat statistiky úspěšnosti jednotlivých hráčů. Rezervy jsou i v ozvučení scény, např. při nárazu tanku do jiného objektu nebo ve vytvoření dynamického hudebního doprovodu.

# Reference

[1] Open Inventor tutoriál na http://www.root.cz/

[2] Josie Wernecke, The Inventor Mentor, Addison-Wesley Professional, 1994, ISBN: 0201624958

[3] Coin3D, http://www.coin3d.org/

[4] Explosion Generator, http://www.geocities.com/starlinesinc/

[5] OpenAL, http://www.openal.org/

[6] Libsndfile, http://www.mega-nerd.com/libsndfile/