Vysoké učení technické v Brně

Fakulta informačních technologií

Diplomová práce

Daniel Výchopeň

2006

Prohlašuji, že jsem tuto diplomovou práci 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

Tato práce je zaměřena na vytvoření aplikace pomocí grafické knihovny Open Inventor. Zabývá se návrhem algoritmů pro jednoduchou 3D hru využívající pohyb vozidla po nerovném terénu, 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 tvořeném modelem krajiny. Cílem hry pak bude najít a zasáhnout nepřátelský tank. Nepřátelský tank bude ovládat jednoduchá umělá inteligence, nebo druhý hráč připojený přes síťové rozhraní. Ve hře jsou také řešeny kolize mezi tankem a bludištěm a vzájemně mezi tanky.

Klíčová slova

Open Inventor, Coin3D, pohyb vozidla, nerovný terén, zvuk, kolize, síťová komunikace, umělá inteligence

Abstract

The purpose of this thesis was to create an application using the Open Inventor library. It deals with design of algorithms for a simple 3D game that implements motion of a vehicle on an unlevel terrain, sound, network communication and physics of objects. The emphasis was on implementation of application based on designed algorithms. The application would contain a tank controlled by keyboard. The tank would ride on surface defined by a model of countryside. The objective of the game would be hitting the enemy tank controlled by a simple artificial intelligence or by another player connected via network interface. The game would also implement collisions between tanks and the model of the countryside.

Key Words

Open Inventor, Coin3D, motion of a vehicle, unlevel terrain, sound, collision, network communication, artificial intelligence

Obsah

1 Úvod

Tato práce se zabývá návrhem algoritmů pro pohyb vozidla, konkrétně tanku, po nerovném terénu. Algoritmy jsou demonstrovány na jednoduché aplikaci, vytvořené pomocí grafické knihovny Open Inventor. Tato aplikace vychází z mého ročníkového projektu. Konkrétně se jedná o 3D hru (obrázek 1.1), ve které hráč jezdí s tankem po bludišti a snaží se vyhledat a zasáhnout nepřátelský tank. Nepřátelský tank pak ovládá jiný hráč připojený přes síťové rozhraní. Ve hře jsou použity exploze [4] a zvuky pomocí knihovny libsndfile [6] a OpenAL[5].

Obrázek 1.1: Vzhled rozšiřované aplikace

Původní aplikace bude tedy touto prací rozšířena o pohyb tanku po nerovném terénu. Sterilní model bludiště tak může být vyměněn za mnohem zajímavější model krajiny. Pohyb tanku po nerovném terénu byl již částečně řešen v mém semestrálním projektu. Nyní bude vylepšen o počítání kolizí mezi tankem a krajinou, a mezi tanky vzájemně. Dále bude přidána

jednoduchá umělá inteligence, aby se dalo hrát jen proti počítači, bez nutnosti druhého hráče a síťového připojení.

Obrázek 1.2: Vzhled výsledné aplikace

Jak již bylo zmíněno, aplikace je vytvořena pomocí grafické knihovny Open Inventor. 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].

V kapitole *Vytvoření základní aplikace* popisuji vytvoření okna aplikace, ovládání pomocí klávesnice a způsob vkládání modelů do scény. Všechny tyto věci jsem řešil již v ročníkovém projektu, takže mohly být převzaty jen s minimálními úpravami.

V následujících kapitolách *Zpracování modelu terénu*, *Usazení tanku na terén* a *Pohyb tanku ve scéně* popisuji navržené algoritmy pro pohyb tanku po nerovném terénu. Touto problematikou jsem se již zabýval v rámci semestrálního projektu.

Dále následují kapitoly *Kolize*, *Umělá inteligence* a *Výkonnostní náročnost*. Ve kterých řeším například kolize tanku s nerovným terénem, jednoduchou umělou inteligenci nepřátelského tanku a měřím výkonnostní náročnost použitých algoritmů.

V kapitole *Další použité techniky* pak popisuji vytvoření animace výbuchu, práci se zvuky a se síťovou knihovnou TNet. Tyto kapitoly vycházejí taktéž z ročníkového projektu.

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("Tanky");
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, nebo máme starší grafickou kartu, na které by byl výpočet osvětlení pomalý, 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 sníží se tak nepříjemná prodleva při spouštění aplikace.

Do grafu scény tedy vložíme model bludiste.iv, což je model světa, ve kterém se budou

pohybovat naše tanky:

```
SoFile *model = new SoFile;
model->name.setValue("models/bludiste.iv");
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 zásahu nepřátelského tanku. Dále pak tato třída bude obsahovat metody pro inicializaci tanku, nastavení pozice a natočení tanku, vrácení rychlosti tanku a podobně.

Vlastní vložení modelu tanku do grafu scény pak provedeme v metodě `void init(SoSeparator *root, SoSeparator *rootKolize char *modelFileName)`, 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 a konečně třetí parametr udává jméno souboru s modelem tanku.. 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:

```
this->model = new SoFile;
this->model->name.setValue(modelFileName);
this->root->addChild(this->model);
```

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

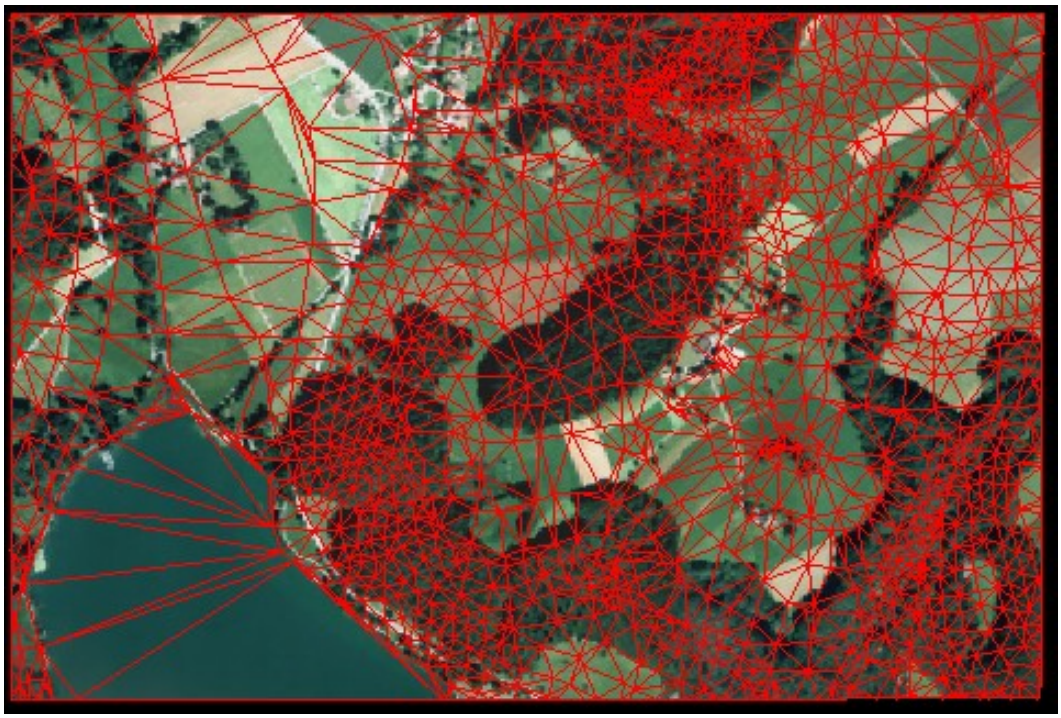
```
tank.init(root, rootkolize, "models/tank.iv");
```

Zároveň stejným způsobem inicializujeme nepřátelský tank, jen můžeme změnit jméno souboru s modelem tanku, pokud bychom si přáli rozdílný vzhled tanků:

```
tank2.init(root, rootkolize, "models/enemy.iv");
```

3 Zpracování modelu terénu

3.1 Zjištění rozměrů modelu terénu



Obrázek 3.1: Příklad modelu terénu, pohled z vrchu

Jednou z možností, jak popsat povrch terénu, po kterém se má pohybovat tank, je použití výškových map, jak popisuje například [9]. Já použil již hotový model terénu složený z jednotlivých trojúhelníků (obrázek 3.1). Povrch terénu je pak definován nejvyššími body modelu terénu. Model terénu je uložen ve formátu Inventoru (přípona souboru je .iv). Aby tank nevyjel mimo vymodelovanou oblast, je potřeba zjistit rozměry terénu. Pro jednoduchost se předpokládá, že terén bude po promítnutí do plochy, tvořené osami x a z (osa y představuje výškovou souřadnici), tvořit obdélník. Tento obdélník nesmí být natočený, to znamená, že jeho jednotlivé strany musí být rovnoběžné s osami x a z. Nyní tedy stačí najít maximální a minimální hodnoty v souřadnicích x a z, které se v modelu vyskytují. To lze provést následovně:

Nejprve se vytvoří pomocný kořen scény:

```
SoSeparator *pomroot = new SoSeparator;
pomroot->ref();
```

A vložíme do něho model terénu bludistekolize.iv, ten nemusí být tak přesný jako model určený k zobrazování, ale hlavně by měl obsahovat jen geometrii, po které se má tank pohybovat, nebo různé překážky jako jsou například domy a stromy. Nesmí obsahovat například geometrii, na které je nakreslena obloha, tank by si totiž vzhledem k použitému algoritmu „vyskočil“ na tuto geometrii a nejezdil by pak po zemi, ale po obláčcích.

```
SoFile *pommodel = new SoFile;
pommodel->name.setValue("models/bludistekolize.iv");
pomroot->addChild(pommodel);
```

Pomocí třídy `SoCallbackAction` a její metody `addTriangleCallback` nastavíme funkci `triangle_cbUrciRozmery`, To způsobí, že po zavolání metody `apply(pomroot)` se pro každý trojúhelník určený pro rendering, zavolá funkce `triangle_cbUrciRozmery`.

```
SoCallbackAction ca1;
ca1.addTriangleCallback(SoShape::getClassTypeId(),
triangle_cbUrciRozmery, NULL);
ca1.apply(pomroot);
```

Pomocí funkce `triangle_cbUrciRozmery` pak získáme jednotlivé vrcholy trojúhelníku v1, v2, v3 typu ukazatele na `SoPrimitiveVertex`. Dále také získáme ukazatel `action` na `SoCallbackAction`.

Pomocí ukazatele `action`, získáme matici modelu, kterou vynásobíme každý vrchol. Dostaneme tak vrcholy o souřadnicích, které jsou shodné se souřadnicemi vrcholů, které můžeme vidět například při vytváření modelu v modelovacím software:

```
const SbVec3f vtx[] = { v1->getPoint(), v2->getPoint(),
                        v3->getPoint() };
const SbMatrix mm = action->getModelMatrix();
SbVec3f vx[3];
for (j=0; j < 3; j++)
    mm.multVecMatrix(vtx[j], vx[j]);
```

Nyní stačí projít všechny vrcholy všech trojúhelníků a zapsat maximální a minimální souřadnice v osách x a z do struktury `zemInf`, ve které si uchováváme informace o terénu.

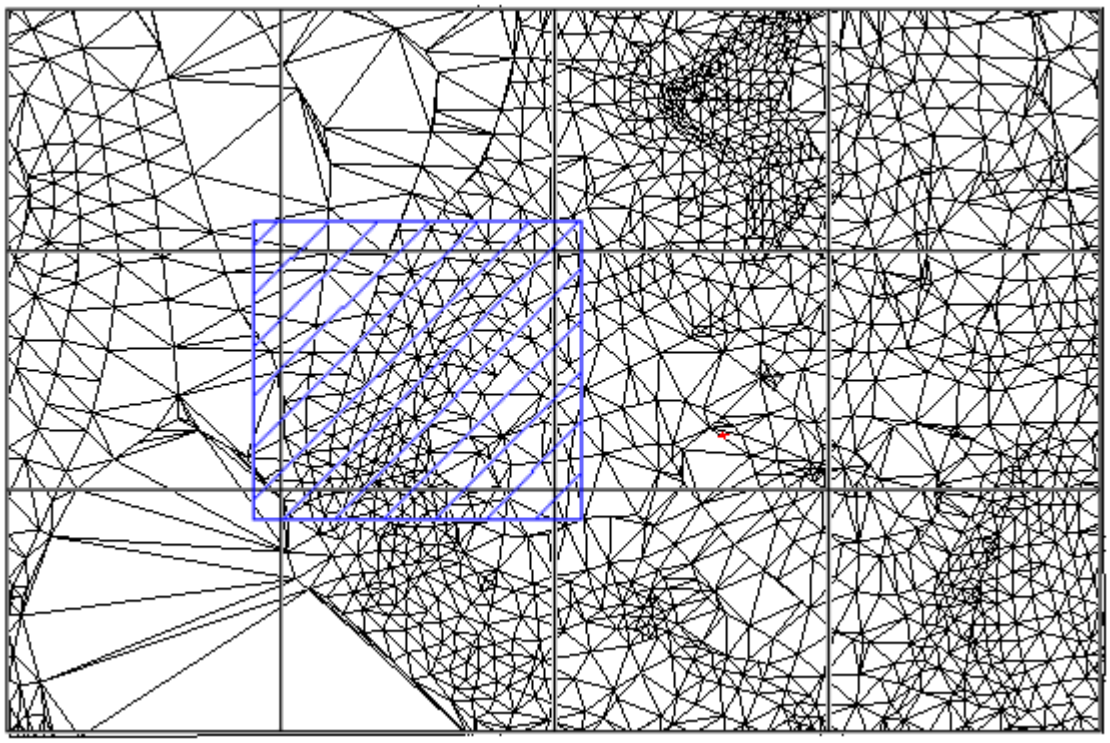
3.2 Rozdělení terénu na sektory

Aby se při umísťování tanku na terén a při počítání kolizí s terénem nemuselo počítat vždy se všemi trojúhelníky terénu, rozdělí se terén kvůli urychlení výpočtu na jednotlivé

sektory. Pak se bude počítat jen s těmi trojúhelníky, které leží v sektoru, ve kterém je právě umístěn tank, nebo střela.

V knize [7] jsou popsány čtyři nejčastěji používané způsoby dělení prostoru: mřížka, oktalový strom, strom BSP a kD-strom. Já pro jednoduchost zvolil rozdělení pravidelnou mřížkou v osách x a z. Toto řešení se mi jeví pro zvolený problém jako dostatečně výkonné.

Počet sektorů v osách x a z lze nastavit v závislosti na velikosti a složitosti terénu. Na obrázku 3.2 je vidět rozdělení na čtyři sektory v ose x a tři sektory v ose z (celkem tedy 12 sektorů). Při přiřazování trojúhelníků do jednotlivých sektorů se hranice sektoru rozšíří o hodnotu překrytí (modře vyšrafovaná oblast v obrázku 3.2). Tank totiž může být na hranici dvou nebo více sektorů a částečně zasahovat i do sousedních sektorů. Hodnota překrytí závisí na velikosti tanku.



Obrázek 3.2: Rozdělení terénu na sektory

Pro každý sektor se vytvoří vektor(`std::vector`), který bude obsahovat všechny trojúhelníky, které leží uvnitř daného sektoru, nebo daný sektor protínají. Obdobně se pro každý sektor vytvoří vektor úseček(stran trojúhelníků) a vektor bodů(vrcholů trojúhelníků). Tyto vektory se budou později hodit při usazování tanku na terén.

Plnění vektorů probíhá ve funkce `triangle_cb`, ta se volá pro každý trojúhelník, stejným způsobem jako funkce `triangle_cbUrciRozmery` při zjišťování rozměrů

terénu. Začátek této funkce je také stejný jako u funkce `triangle_cbUrciRozmery`, to znamená, že se vrcholy vynásobí modelovou maticí. Protože několik trojúhelníků může mít jeden společný vrchol, je třeba zajistit, aby ve vektorech vrcholů nebyly stejné body. To by totiž negativně ovlivnilo rychlost pozdějších výpočtů. K tomu se využije třída `SbBSPTree`, která umožňuje hledání bodů v čase O(log(n)). Při zpracování vrcholů se pak nejprve pomocí metody `findPoint` dotážeme, zda již nebyl bod zpracováván:

```
if(bsptree->findPoint(vx[k])==-1)
```

Do bsptree se body přidávají metodou addPoint:

```
bsptree->addPoint(vx[k]);
```

Strany trojúhelníků se mohou také opakovat, tak je také třeba zajistit, aby se ve vektorech sektorů nevyskytovali shodné úsečky. To se povede prostým sekvenčním prohledáním daného vektoru. Obdobně se kontrolují na duplicitu i samostatné trojúhelníky, zpracovávají se totiž jejich přední i zadní strany.

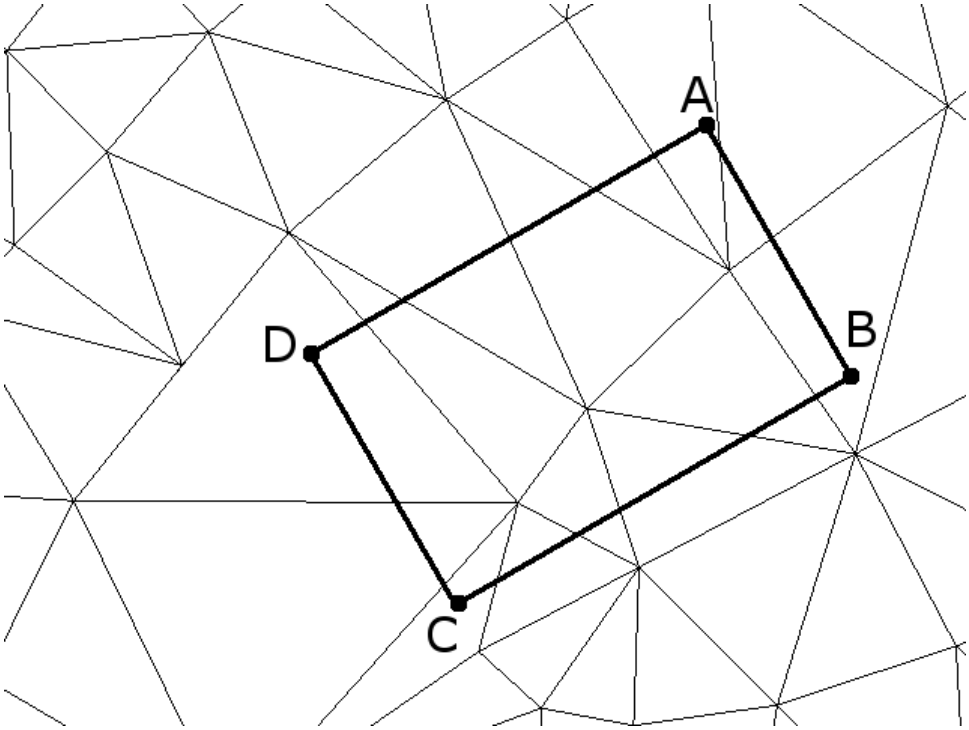
Pro každý trojúhelník a jeho strany a vrcholy se pak ve funkci `triangle_cb` zjišťuje, do kterých sektorů zasahují a do těch se přidají metodou `push_back`. Vzhledem k překrývání jednotlivých sektorů, může jeden vrchol náležet zároveň až do čtyřech sektorů.

4 Usazení tanku na terén

4.1 Výpočet opěrných bodů

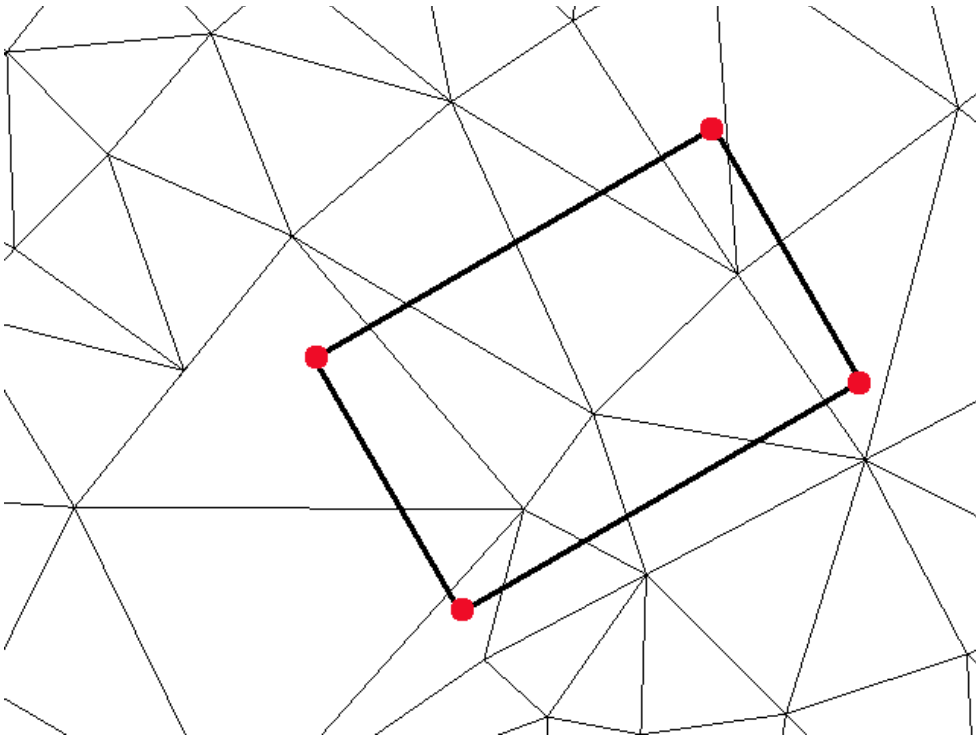
Při usazování tanku na terén se nejprve spočítají všechny body pod tankem, na kterých by tank mohl být posazen. Ve druhé fázi se na tyto body usadí plocha, reprezentující spodní část tanku.

Začne se výpočtem souřadnic A, B, C, D obdélníku, který pomyslně ohraničuje tank, jak je vidět na obrázku 4.1.



Obrázek 4.1: Obdélník ohraničující tank

K výpočtu souřadnic souřadnic postačí pozice v osách xz, úhel natočení tanku kolem osy y a rozměry tanku. Obdélník je umístěn v terénu vodorovně, to znamená, že se nijak nenaklápí. To je z toho důvodu, že ještě nevíme, jak bude tank na terénu naklopen. Toto zjednodušení může způsobovat nepřesnost při naklápění tanku, ale chyba není nijak zásadní a jde zpozorovat jen v určitých situacích při větších úhlech naklopení tanku.



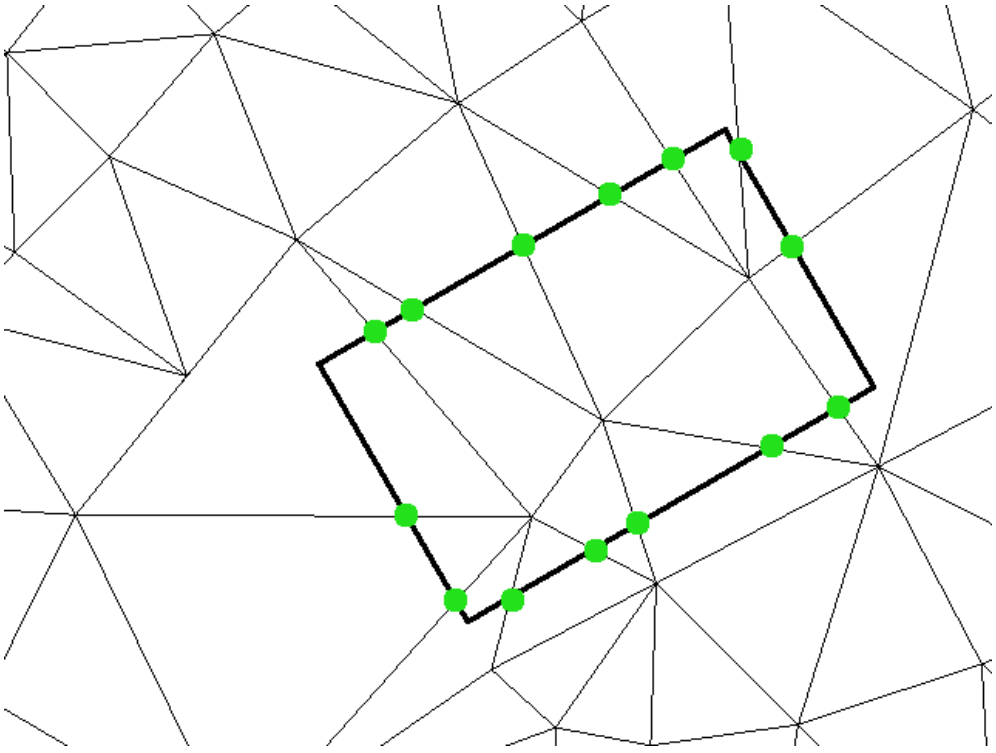
Obrázek 4.2: Opěrné body v rozích obdélníku

Nejprve se vypočítají opěrné body v rozích obdélníku (viz červené body na obrázku 4.2). K tomu se využije vektor trojúhelníků sektoru, ve kterém se právě nachází tank. Opěrné body se pak určí jako průniky svislých přímek v bodech A, B, C, D a všech trojúhelníků v sektoru. Opěrné body se ukládají do vektoru bodů „`body`" metodou `push_back`. Těchto opěrných bodů může být více než čtyři, záleží na tom, jakým způsobem je vytvářen model terénu.

```
for(i=0; i<trojVec[a][b].size(); ++i)
{
    if(trojVec[a][b][i].prunik(tankObd.A, pomb))
        body.push_back(pomb);
    if(trojVec[a][b][i].prunik(tankObd.B, pomb))
        body.push_back(pomb);
    if(trojVec[a][b][i].prunik(tankObd.C, pomb))
        body.push_back(pomb);
    if(trojVec[a][b][i].prunik(tankObd.D, pomb))
        body.push_back(pomb);
}
```

Další opěrné body mohou vzniknout na hranách trojúhelníků (viz zelené body na obrázku 4.3). Spočítají se jako průniky čtyř svislých ploch, které jsou dané úsečkami AB, BC, CD, DA a všech úseček z vektoru úseček (stran trojúhelníků) sektoru, ve kterém se právě nachází tank.

```
for(i=0; i<useckaVec[a][b].size(); ++i)
{
   if(useckaVec[a][b][i].prunik(tankObd.A,tankObd.B, pomb))
        body.push_back(pomb);
   if(useckaVec[a][b][i].prunik(tankObd.B,tankObd.C, pomb))
        body.push_back(pomb);
   if(useckaVec[a][b][i].prunik(tankObd.C,tankObd.D, pomb))
        body.push_back(pomb);
   if(useckaVec[a][b][i].prunik(tankObd.D,tankObd.A, pomb))
        body.push_back(pomb);
}
```

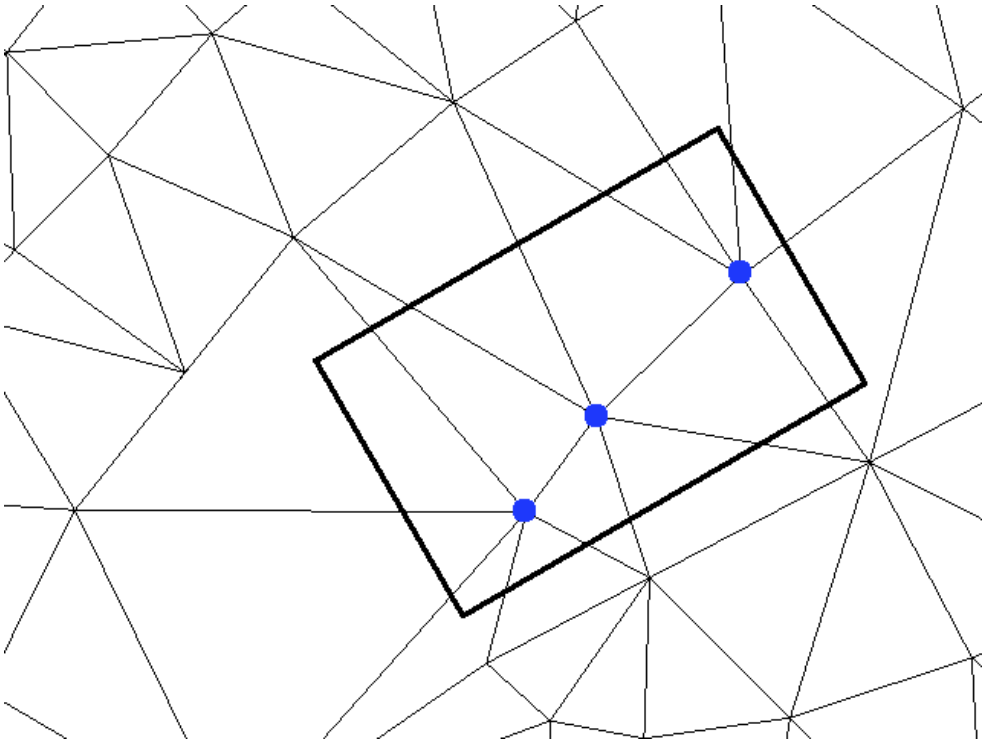


Obrázek 4.3: Opěrné body na hranách trojúhelníků

Poslední body, o které by se tank mohl potencionálně opřít, tvoří vrcholy trojúhelníků

uvnitř obdélníku (viz modré body na obrázku 4.4). K tomu se využije vektor bodů (vrcholů trojúhelníků) sektoru, ve kterém se právě nachází tank. Z tohoto vektoru se vyberou ty body, které leží uvnitř obdélníku. Aby byl bod uvnitř obdélníku, musí platit, že je nalevo od čtyř úseček, které jsou tvořeny vrcholy obdélníku, tedy od úseček AB, BC, CD a DA:

```
for(i=0; i<bodVec[a][b].size(); ++i)
{
    pomb=bodVec[a][b][i];
    if(pomb.jeVlevo(tankObd.A, tankObd.B)
            && pomb.jeVlevo(tankObd.B, tankObd.C)
            && pomb.jeVlevo(tankObd.C, tankObd.D)
            && pomb.jeVlevo(tankObd.D, tankObd.A))
        body.push_back(pomb);
}
```



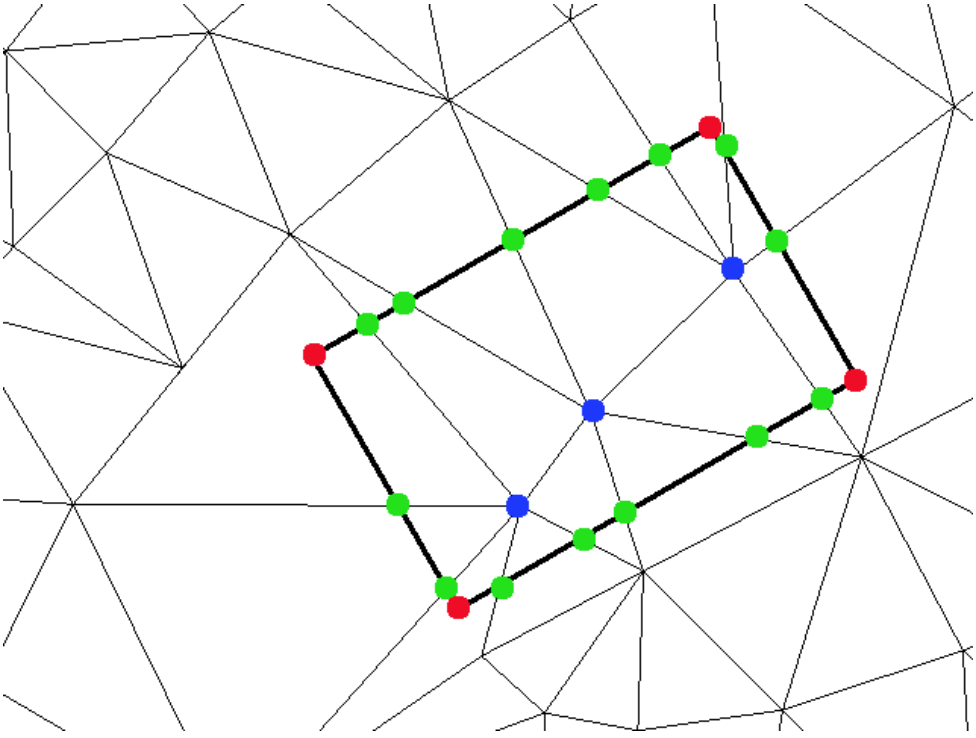
Obrázek 4.4: Opěrné body na vrcholech trojúhelníků

4.2 Snížení počtu opěrných bodů

Nyní jsou ve vektoru „`body`“, všechny opěrné body, které jsou potřeba k usazení tanku na terén (viz body na obrázku 4.5). Těchto bodů, však může být, v závislosti na modelu terénu, relativně hodně. Mnohdy lze počet bodů výrazně snížit provedením následujících optimalizací a urychlit tak výpočet sloužící k usazení tanku na terén.

Body nebudeme z vektoru fyzicky odstraňovat, vytvoříme jen pole hodnot typu `boolean`, které bude mít stejný počet prvků jako vektor opěrných bodů a bude říkat, který bod je platný (hodnota `true` na stejném indexu) a který bod je neplatný (hodnota `false` na stejném indexu). Pokud bude bod označen jako neplatný, už se s ním dál nebude počítat.

```
int pocBodu = body.size();
bool * platny = new bool[pocBodu];
for(i=0; i<pocBodu; ++i)
    platny[i]=true;
```



Obrázek 4.5: Všechny opěrné body

Pokud budou mít dva body stejné nebo velmi podobné souřadnice x a z, ponecháme jen ten bod, který je výš, to znamená ten, který má větší hodnotu v ose y.

```
for(i=0; i<pocBodu-1; ++i)
    for(j=i+1; j<pocBodu; ++j)
        if(platny[i] && platny[j])
            if((body[i].A[0]-body[j].A[0])
                *(body[i].A[0]-body[j].A[0])
                +(body[i].A[2]-body[j].A[2])
                *(body[i].A[2]-body[j].A[2])<0.00001)
                if(body[i].A[1]>body[j].A[1])
                    platny[j]=false;
                else
                    platny[i]=false;
```

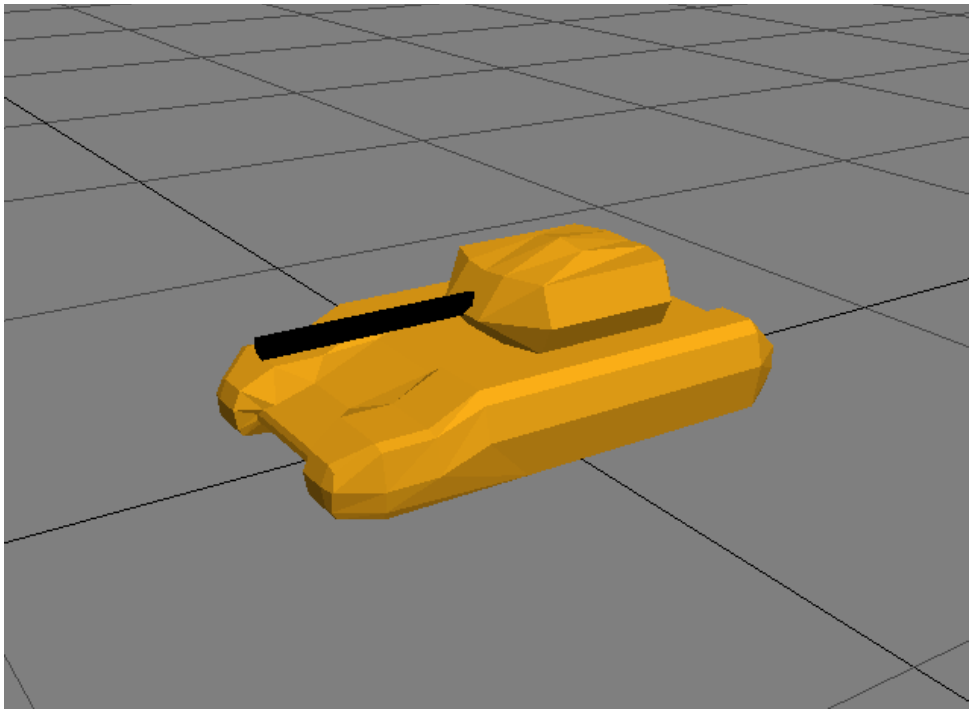
Dále se můžou odstranit ty body, které leží na úsečce mezi dvěma jinými body nebo leží pod touto úsečkou. K tomu slouží funkce `naStejnePrimceAPod`, která ze tří bodů vrátí číslo bodu, který leží na úsečce nebo po úsečkou mezi zbylými dvěma body. Pokud to nesplňuje žádný bod, funkce vrátí nulu.

```
for(i=0; i<pocBodu-2; ++i)
    for(j=i+1; j<pocBodu-1; ++j)
        for(k=j+1; k<pocBodu; ++k)
            if(platny[i] && platny[j] && platny[k])
                if((m=naStejnePrimceAPod(body[i].A,
                    body[j].A,body[k].A))!=0)
                {
                    if(m==1)
                        platny[i]=false;
                    if(m==2)
                        platny[j]=false;
                    if(m==3)
                        platny[k]=false;
                }
```

4.3 Usazení tanku na opěrné body

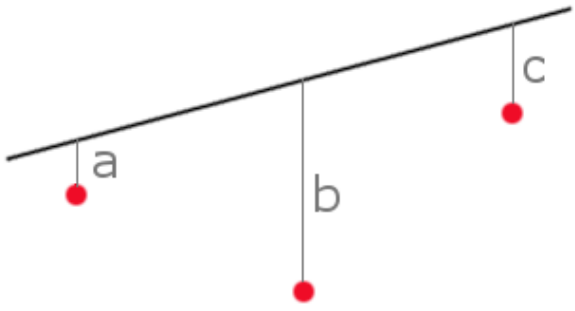
Spodní část modelu tanku (viz obrázek 4.6) je relativně plochá, proto si můžeme dovolit

aproximovat ji jednou plochou.



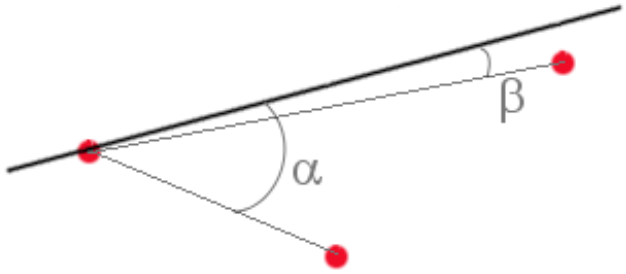
Obrázek 4.6: Model tanku bez textur

Jako počáteční naklopení plochy se použije minulé naklopení tanku. Nyní se určí první podpěra, což bude bod, který má nejmenší vertikální vzdálenost k ploše. Např. na obrázku 4.7 by se ze tří bodů vybral bod s vertikální vzdáleností „a“.

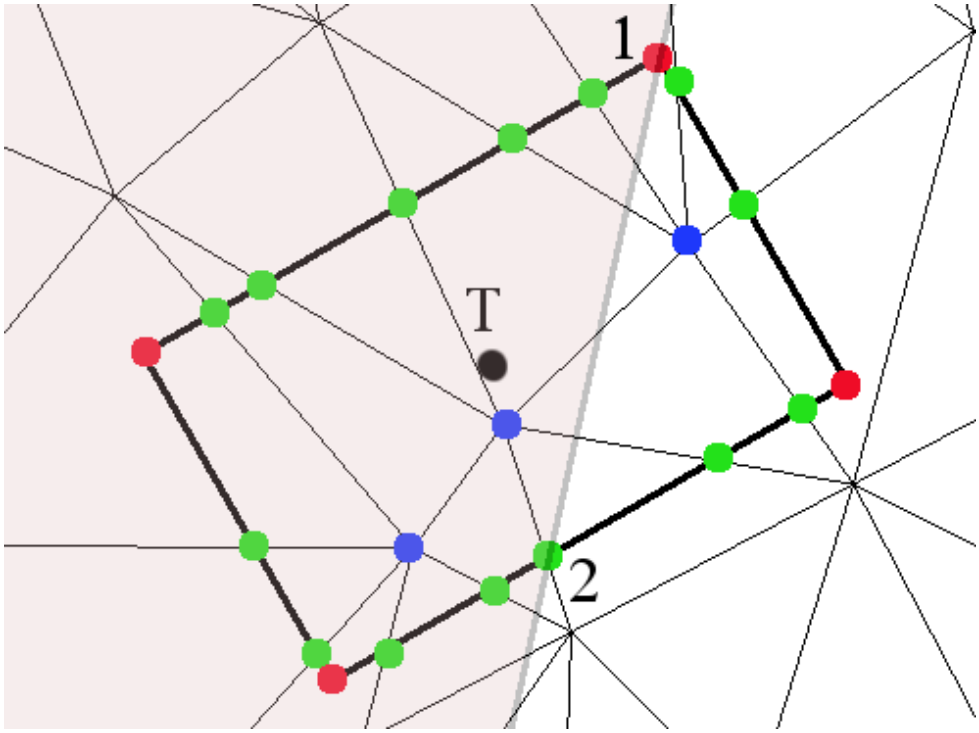


Obrázek 4.7: Volba prvního podpěrného bodu

Jako druhý podpěrný bod se vybere ten bod, který plochu natočí kolem prvního podpěrného bodu o nejmenší úhel. Na obrázku 4.8 by to byl bod, který s plochou svírá úhel β.



Obrázek 4.8: Určení druhého podpěrného bodu



Obrázek 4.9: Výběr třetího podpěrného bodu

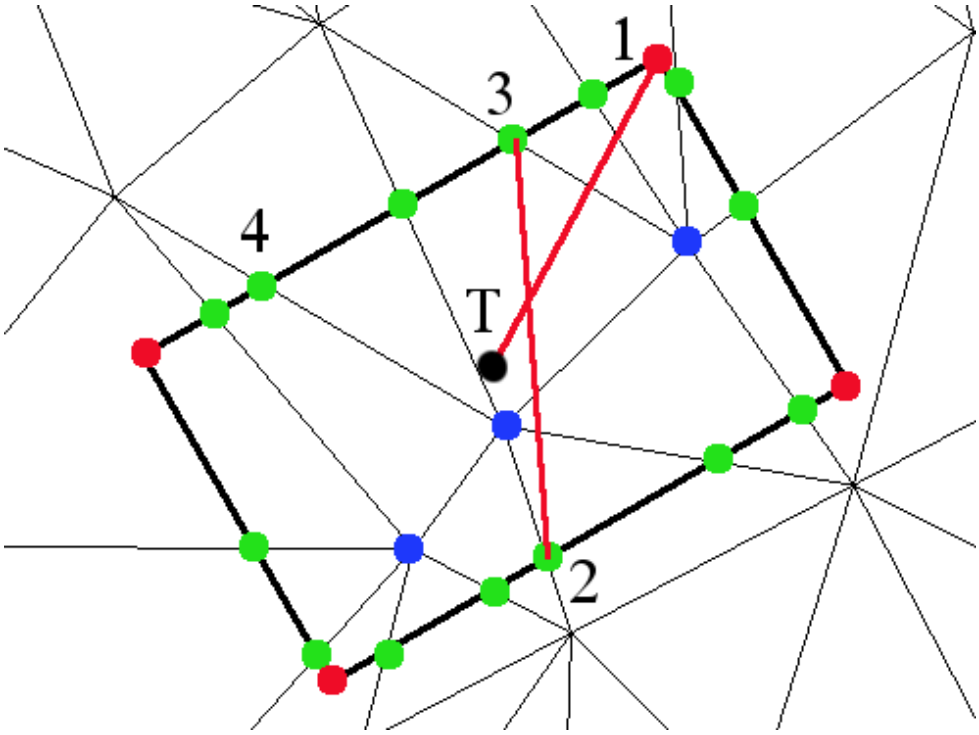
Nyní se plocha opírá o dva podpěrné body (na obrázku 4.9 označené čísly 1 a 2). Nyní se bude otáčet kolem přímky určené těmito body. Směr otáčení je dán těžištěm tanku (bod T na obrázku 4.9). Třetí bod se tedy bude vybírat jen z té poloroviny (počítáno v osách xz), ve které

je umístěno těžiště tanku (tmavší plocha na obrázku 4.9). Vybere se ten bod, který natočí plochu o nejmenší úhel.

Nyní jsou vybrány tři podpěrné body a musí se zjistit, jestli těžiště tanku leží uvnitř trojúhelníku, který je tvořen těmito body. Pokud tomu tak je, našli jsme tři body, které určují plochu, podle které se naklopí tank. Pokud tomu tak není, musíme odstranit jeden podpěrný bod a pokračovat v hledání třetího podpěrného bodu.

Vytvoříme úsečku z prvního podpěrného bodu do třetího podpěrného bodu a úsečku z druhého podpěrného do těžiště tanku. Zjistíme jestli se úsečky protínají (bez uvažování osy y). Pokud se protínají, znamená to, že těžiště neleží mezi třemi podpěrami. V tom případě prohlásíme druhý podpěrný bod za neplatný, třetí podpěrný bod prohlásíme za druhý podpěrný bod a znovu hledáme třetí podpěrný bod (stejným způsobem jako jsme hledali třetí podpěrný bod).

Pokud se úsečky neprotínají, vytvoříme úsečku z druhého podpěrného bodu do třetího podpěrného bodu a úsečku z prvního podpěrného do těžiště tanku. Zjistíme jestli se úsečky protínají (bez uvažování osy y). Pokud se protínají, znamená to, že těžiště neleží mezi třemi podpěrami. V tom případě prohlásíme první podpěrný bod za neplatný, třetí podpěrný bod prohlásíme za první podpěrný bod a znovu hledáme třetí podpěrný bod.



Obrázek 4.10: Výběr třetího podpěrného bodu

Pokud se úsečky v obou případech neprotínají, znamená to, že těžiště tanku leží uvnitř trojúhelníku, který je tvořen třemi nalezenými podpěrnými body a hledání končí.

Na obrázku 4.10 je ukázán příklad, kdy známe dvě podpěry (body číslo 1 a 2) a je určena třetí podpěra (bod číslo 3). Při vytvoření úsečky z druhého podpěrného bodu do třetího podpěrného bodu a úsečky z prvního podpěrného bodu do těžiště tanku se zjistí, že se úsečky protínají (těžiště leží mimo podpěrné body). Bod 1 se tedy prohlásí za neplatný a bod 3 se prohlásí za první podpěrný bod. Znovu se hledá třetí podpěrný bod (plocha se nyní bude natáčet kolem přímky dané body 2 a 3). Pokud se vybere bod označený číslem 4 (to znamená že natočí plochu o nejmenší úhel), žádný průnik úseček nevznikne (těžiště leží uvnitř trojúhelníku, daném body 2, 3 a 4).

Tři podpěrné body definují plochu, na které leží tank. Vypočítáme výšku tanku y jako průnik podpěrné plochy se svislou přímkou, danou pozicí tanku v osách x a z. Pak posuneme model tanku na souřadnice x, y, z. Dále natočíme model tanku kolem osy y a sklopíme model podle normály podpěrné plochy:

```
SbRotation rot(SbVec3f(0,1,0),this->spodek.getNormal());
this->setRotation(this->getRotation());
this->rotace->rotation =
                this->rotace->rotation.getValue()*rot;
```

5 Pohyb tanku ve scéně

5.1 Rychlost a dráha ujetá tankem

Jako první se vypočítá vektor `smer`, který udává směr pohybu tanku:

```
SbVec3f smer;
smer[0]=-sinf(natoceni);
smer[1]=0;
smer[2]=-cosf(natoceni);
```

A naklopí se podle normály spodní (podpěrné) plochy tanku:

```
SbRotation rotuj(SbVec3f(0,1,0),tank.spodekNormala);
rotuj.multVec(smer, smer);
smer.normalize();
```

Dále se vypočítá maximální rychlost tanku v závislosti na stoupání, výkonu a hmotnosti tanku, gravitačním zrychlení a třecí síle Ft (představuje odpor prostředí, valivý odpor apod.). Stoupání je zde reprezentováno ypsilonovou souřadnicí normalizovaného směrového vektoru (`smer[1]`), což vlastně odpovídá funkci sin(úhelStoupání).

```
float vmaxsklon;
if(Ft+g*hmotnost*smer[1]>0)
    vmaxsklon = vykon/(Ft+g*hmotnost*smer[1]);
else
    vmaxsklon=maxv;
```

Rychlost tanku je dále omezena maximální konstrukční rychlostí tanku `maxv`:

```
if(vmaxsklon>maxv)
    vmaxsklon=maxv;
```

Obdobně se pak vypočítá minimální rychlost tanku `vminsklon` (rychlost jízdy zpět, záporná hodnota).

Podle stisknutých kláves vypočítáme novou rychlost. Pokud je například stisknuta klávesa pro pohyb tanku dopředu, vypočítá se nová rychlost následovně:

```
if(rychlost<vmaxsklon)
{
    rychlost += (float)uplynulyCas*(0.5*zrychleni
                                    -0.5*zrychleni*smer[1]);
```

```
        if(rychlost>vmaxsklon)
            rychlost = vmaxsklon;
    }else
    {
        rychlost-=(float)uplynulyCas*(0.5*zpomaleni
                                      +0.5*zpomaleni*smer[1]);
        if(rychlost<vmaxsklon)
            rychlost = vmaxsklon;
    }
```

Je zde vidět, že pokud je `rychlost` tanku vyšší než rychlost `vmaxsklon` (k čemuž může dojít například v situaci, pokud je tank rozjetý jízdou po rovině a následně jede do kopce), tak se rychlost postupně zpomaluje, dokud nedosáhne rychlost `vmaxsklon`.

Zrychlení a zpomalení je zde rozděleno na dvě části, z nichž jedna část je vždy konstantní a jedna část je ovlivněna stoupáním. Poměrem velikostí jednotlivých částí můžeme ovlivnit jízdní vlastnosti tanku. Zde je napevno nastaven poměr 1:1.

Analogicky je pak počítána rychlost při stisknutí klávesy pro jízdu dozadu. Podobně se také počítá i rychlost natáčení tanku, tam je však výpočet jednodušší, protože se neuvažuje sklon terénu.

Pokud není stisknuta klávesa pro pohyb dopředu ani pro pohyb dozadu, dochází pouze k postupnému zpomalování tanku na rychlost rovnou nule.

Nyní se vypočítá dráha, kterou tank ujel za dobu uplynulyCas (doba od posledního přepočítání scény). Vztah by se měl ještě podělit číslem 2, protože se uvažuje průměrná rychlost, ale není to nutné, jelikož toto dělení lze zahrnout přímo do konstant, ze kterých se počítá rychlost.

```
float s =(rychlost+tank.getRychlost())*(float)uplynulyCas;
```

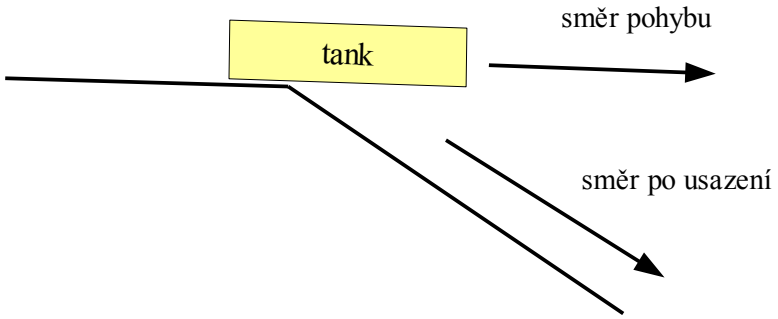
Teď posuneme tank ve směru pohybu o dráhu s:

```
SbVec3f posunTanku=smer*s;
tank.setPosition(tank.getPosition()+posunTanku);
```

Po posunutí se zavolá metoda posadNaTeren, která určí přesnější usazení tanku (viz kapitola Usazení tanku na terén).

5.2 Plynulejší sklápění tanku

Aby byl pohyb tanku po terénu plynulejší, je třeba zařídit, aby se tank při jízdě přes hrboly nesklápěl skokově (obrázek 5.1).



Obrázek 5.1: Usazení tanku

Rychlost sklápění tanku je tedy třeba zpomalit. Nejprve detekujeme, jestli se vůbec jedná o sklápění směrem dolů a že je tedy třeba tlumit pohyb. To se provádí porovnáním ypsilonových složek směru pohybu před a po novém usazení. V úvahu je třeba přitom brát i směr pohybu tanku (záporná nebo kladná rychlost):

```
bool ztlum=false;
if(rychlost>0 && novySmer[1]<smer[1])
    ztlum=true;
else
    if(rychlost<0 && novySmer[1]>smer[1])
        ztlum=true;
```

Zjistíme úhel (proměnná `angle`) mezi normálami minulé (this->spodekNormala) a nově vypočítané spodní plochy (Sn):

```
rot.setValue(this->spodekNormala, Sn);
rot.getValue(axis, angle);
```

Určí se rychlost naklápění a maximální úhel (proměnná `naklon`) o který je možno naklonit tank za dobu uplynulyCas (dobu uplynulou od posledního přepočítání scény):

```
rychlostNaklapeni += (float)uplynulyCas*zrychleniNaklapeni;
naklon += (float)uplynulyCas*rychlostNaklapeni;
```

Pokud je `angle>naklon`, probíhá sklápění moc rychle a je třeba zmírnit úhel sklopení spodní plochy na hodnotu `naklon`. Naklápění bude probíhat podle osy `axis`, která je již vypočítaná (od zjišťování úhlu mezi normálami):

```
Sn=this->spodekNormala;
rot.setValue(axis, naklon);
rot.multVec(Sn,Sn);
```

Na závěr se zjistí umístění nové plochy (definované normálou `Sn`) v ose y, určením jednoho opěrného bodu (viz zjištění prvního opěrného bodu v kapitole Usazení tanku na terén).

5.3 Skoky a pády tanku

Další vylepšení pohybu tanku se týká zprovoznění skoků a pádů tanku. Vzpomeňme si, jak se nastavovala pozice tanku:

```
SbVec3f posunTanku=smer*s;
tank.setPosition(tank.getPosition()+posunTanku);
```

Jelikož vektor `smer` je trojrozměrný nastavovala se tehdy správně i ypsilonová složka pozice tanku. Toho nyní využijeme při počítání minimální výšky pozice tanku:

```
this->getPosition(x, y, z);
float ymin = y-0.5f*gr*uplynulyCas*uplynulyCas;
```

Proměnná `gr` zde značí gravitační zrychlení. Jedná se vlastně o počítání šikmého vrhu a ypsilonová složka pozice tanku nesmí klesnout pod hodnotu `ymin`. Po usazení tanku na terén to tedy zkontrolujeme a případně posuneme tank do výšky `ymin`:

```
if(y<ymin)
{
    y=ymin;
    this->veVzduchu=true;
}else
    this->veVzduchu=false;
this->setPosition(x,y,z);
this->spodek=SbPlane(Sn, this->getPosition());
```

Přitom se nastaví i proměnná veVzduchu, která značí, zda se tank dotýká podložky. Ta se pak využívá k rozhodnutí, zda měnit rychlost a natočení tanku.

6 Kolize

6.1 Kolize tanku s terénem

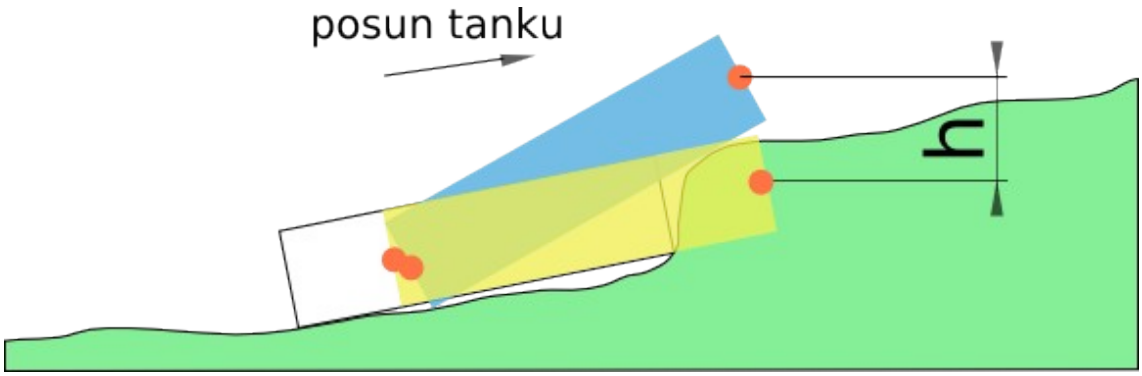
Kolize tanku s terénem provádím následovně. Nejprve se, po každém natočení nebo posunutí tanku, v každém rohu tanku nadefinuje jeden bod, na který se uplatní stejné transformace jako na samotný tank:

```
for(i=0; i<4; ++i)
     kolizniBod1[i] =
          tank.transformujBodJakoTank(tank.kolizniBod[i]);
```

Pak se teprve tank usadí na terén:

```
tank.posadNaTeren(uplynulyCas);
tank.naklop();
```

Následně se vypočítají nové pozice rohových bodů tanku, stejným způsobem jako před usazením tanku.

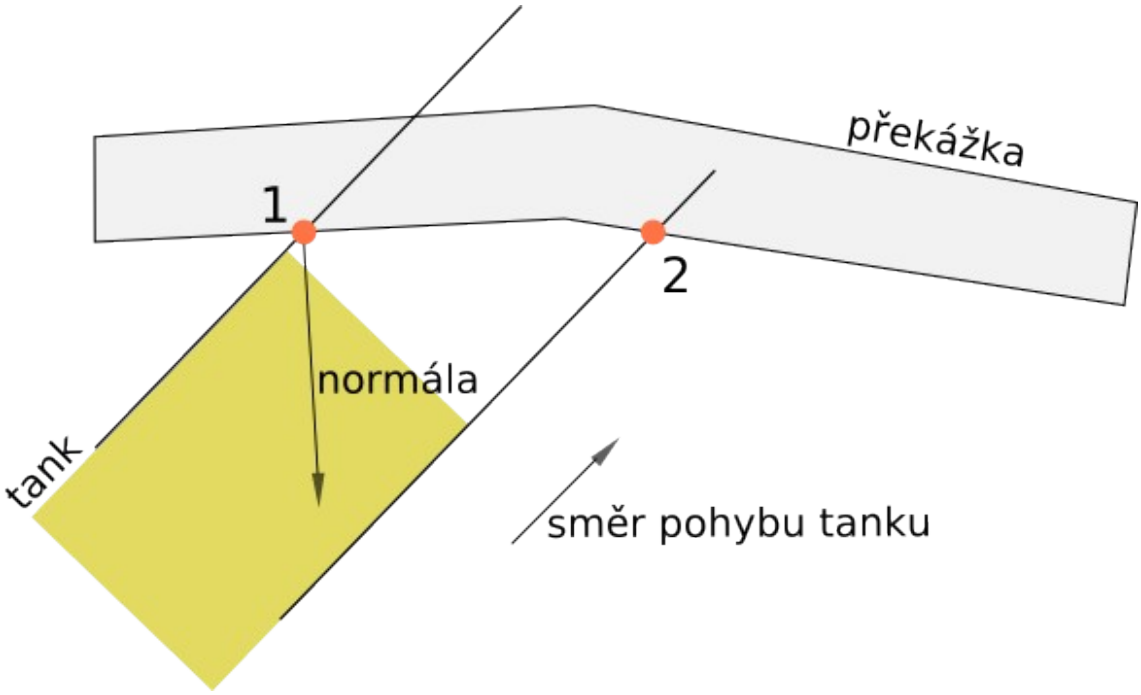


Obrázek 6.1: Kolize tanku s terénem

Nyní se určí, zda došlo ke kolizi. To se provádí tak, že se vypočítá vertikální vzdálenost mezi původní a následnou pozicí rohových bodů. Pokud je tato vzdálenost příliš veliká, je vyhodnoceno, že došlo ke kolizi tanku s terénem a tank se musí posunout zpět na původní pozici. Vše je pro názornost ukázáno na obrázku 6.1, kde je vidět původní poloha tanku (bílý obdélník), posunutý tank (žlutý obdélník) a tank po usazení na terén (modrý obdélník). Na obrázku jsou také znázorněny rohové body (červené tečky) a vertikální vzdálenost mezi nimi (kóta h).

6.2 Klouzání tanku podél překážky

Pokud dojde ke kolizi s terénem, je tank vrácen zpět na svou předešlou pozici. Aby však jízda s tankem vypadala věrohodněji, zkusí se ještě tankem pohnout podél překážky.



Obrázek 6.2: Určení normály překážky

K posunutí tanku podél překážky, budeme potřebovat normálu trojúhelníku terénu, do kterého tank narazil. K tomu si nadefinujeme dvě úsečky (viz obrázek 6.2), dané bočními stranami tanku a směrem pohybu tanku. Pomocí funkce `kolizeSTerenem` pak určíme průsečíky úseček s terénem a normály od proťatých trojúhelníků. Vybereme tu normálu, která je blíže tanku. Podle obrázku 6.2 by to tedy byla normála trojúhelníku u průsečíku označeného číslem 1.

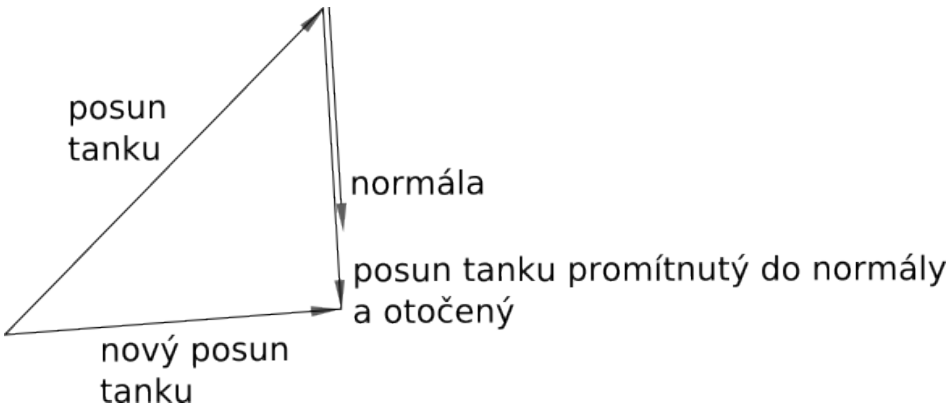
Funkce `kolizeSTerenem` vrací při kolizi s terénem bližší průsečík k prvnímu bodu úsečky. Průsečíky se počítají se všemi trojúhelníky v sektorech, kterými prochází úsečka. Postup pro výpočet průsečíku s trojúhelníkem je popsán například v [7], nejprve se vypočítá průsečík přímky s rovinou, ve které trojúhelník leží, pak se testuje, zda průsečík leží mezi body úsečky a zároveň leží uvnitř trojúhelníku.

Když máme normálu překážky, můžeme již určit nový posun tanku (viz obrázek 6.3). Promítneme vektor `posunTanku` do vektoru `normala`. K tomu použijeme skalární součin těchto vektorů:

```
float dotProd=posunTanku.dot(normala);
```

Skalární součin by měl být záporný, protože vektory by měli jít proti sobě. Pokud není záporný, znamená to, že je normála otočená. Provedeme tedy kontrolu a případně normálu otočíme:

```
if(dotProd>0.0f)
{
    dotProd = -dotProd;
    normala = -normala;
}
```



Obrázek 6.3: Určení nového posunu tanku

Nyní můžeme vypočítat nový posun tanku, jen dodám, že `normala` je již normalizovaný vektor:

```
SbVec3f novyPosun = posunTanku -(dotProd*normala);
```

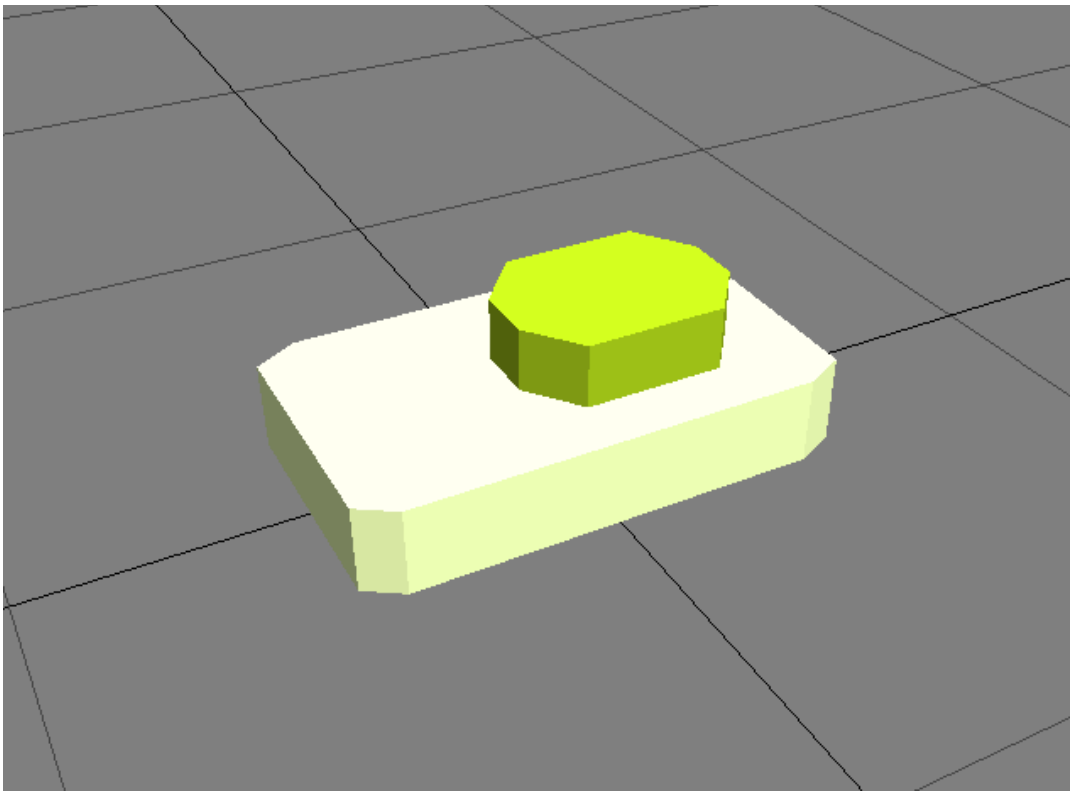
Teď už jen posuneme tank podle vektoru novyPosun a opět zkontrolujeme, zda nedošlo ke kolizi. Pokud dojde ke kolizi, vrátíme tank zpět a nastavíme rychlost tanku na nulu. Pokud ke kolizi nedojde, snížíme rychlost tanku o „d“, což je rychlost, o kterou se tank zpomalí vlivem tření o zeď a vypočítá se následovně:

```
d = (float)uplynulyCas *
(1.0f-novyPosun.length()/posunTanku.length())*zpomaleniZed;
```

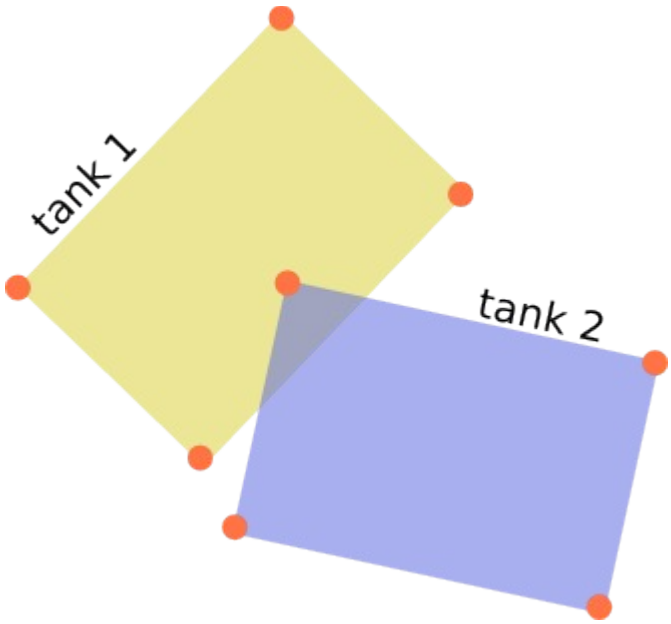
6.3 Kolize mezi tanky

Původně jsem pro zjišťování kolizí mezi tanky používal třídu Inventoru `SoIntersectionDetectionActio`. Vytvořil jsem tak pomocnou scénu pro počítání

kolizí se zjednodušenými modely tanku (viz obrázek 6.4) a nechal počítání kolizí na Inventoru. Když jsem však aplikoval tuto metodu, byl patrný pokles vykreslovaných snímků za sekundu (v mém případě z 73FPS na 66FPS). Z toho důvodu tuto metodu používám už jen u počítání kolizí mezi střelou a tankem.



Obrázek 6.4: Zjednodušený model tanku pro výpočet kolizí



Obrázek 6.5: Kolize mezi tanky

Počítání kolizí mezi tanky řeším jednoduše tak, že zjistím, zda nějaký rohový bod tanku nezasahuje do druhého tanku, tedy do obdélníku tvořeném rohovými body druhého tanku (viz obrázek 6.5).

6.4 Kolize mezi kamerou a krajinou

Kamera, kterou sledujeme scénu, je umístěna kousek za tankem. Pokud se mezi kameru a tank dostane nějaký objekt krajiny, je třeba zajistit, aby se kamera posunula blíže k tanku, jinak by hráč ztrácel přehled o dění na scéně. Daný problém vyřešíme opět pomocí funkce `kolizeSTerenem`, která, pokud dojde ke kolizi, nám vrátí nejbližší průsečík orientované úsečky s terénem. Úsečka je zadána pomocí pozice tanku a pozice kamery.

7 Umělá inteligence

7.1 Popis jednotlivých stavů

Aby se dalo hrát jen proti počítači, bez nutnosti druhého hráče a síťového připojení, přidal jsem do hry jednoduchou umělou inteligenci nepřátelského tanku. Umělá inteligence je zde řešena jako stavový automat s následujícími stavy:

`PohybPoCeste` – v tomto stavu se tank pohybuje po cestě, definované souřadnicemi v souboru „enemy.shac“. Pokud je vidět nepřátelský tank, přejde se do stavu `StrelNaTank`. Pokud je tank ostřelován, přejde se do stavu `PodivejSePoTanku`.

`StrelNaTank` – v tomto stavu se tank snaží natočit směrem na nepřátelský tank a následně tento tank zasáhnout. Pokud nepřátelský tank zmizí ze zorného pole tanku, přejde se do stavu `NatocZpet`.

`NatocZpet` – v tomto stavu se tank natáčí zpět tak, aby mohl pokračovat v jízdě po cestě. Pokud je vidět nepřátelský tank, přejde se do stavu `StrelNaTank`. Pokud je tank ostřelován, přejde se do stavu `PodivejSePoTanku`. Pokud je dosaženo požadovaného natočení tanku, přejde se do stavu `PohybPoCeste`.

`PodivejSePoTanku` – v tomto stavu se tank snaží natočit směrem na nepřátelský tank. Nepřátelský tank ale není v jeho zorném poli. Do toho stavu se automat dostane, pokud v blízkosti tanku dochází k výbuchům nepřátelských střel, tank je tedy ostřelován nepřítelem. Pokud tank zahlédne nepřítele, přejde se do stavu `StrelNaTank`. Pokud se tank natočí směrem na místo, odkud byl ostřelován a přesto nezahlédne nepřátelský tank, přejde se do stavu `NatocZpet`.

Zde popsaná inteligence je velmi jednoduchá, tank se pohybuje stále dokola po jedné, předem dané cestě. Složitější varianty pohybu objektů ve scéně jsou popsány například v [8].

7.2 Zamíření na tank

Pokud chce tank vystřelit na nepřítele, tak se nejprve natočí jeho směrem pomocí funkce `Enemy::natocNa(const float & uhel)`. Poté se vypočítá úhel sklonu hlavně a může se střílet na cíl. Aby to tank řízený počítačem neměl tak jednoduché, je k pozici cíle připočítávána vzdálenost, která je závislá částečně na počtu výstřelů na cíl umístěný ve stejné oblasti a částečně na generátoru pseudonáhodných čísel:

```
switch(vystrelCislo++)
{
```

```
case 0:
    nepresnost = 28+int(10*rand()/(RAND_MAX + 1.0f));
    break;
case 1:
    nepresnost = 14+int(8*rand()/(RAND_MAX + 1.0f));
    break;
      …
}
```

Úhel sklonu hlavně se pak vypočítá metodou půlení intervalu. Nejprve se nastaví maximální a minimální úhel sklonu hlavně `alfaMin` a `alfaMin`. Pak se v jednotlivých iteracích vypočítá úhel sklonu hlavně `alfa` jako střed mezi maximální a minimální hodnotou. Pomocí úhlu sklonu hlavně se vypočítá výška zásahu `yVypocitane` při požadované horizontální vzdálenosti `x`. Následně se porovná vypočítaná výška zásahu `yVypocitane` se zadanou výškou cíle `y`. Pokud platí, že `yVypocitane < y`, nastaví se `alfaMin` na hodnotu `alfa`, jinak se na hodnotu alfa nastaví `alfaMax`:

```
while(fabs(yVypocitane-y)>presnost && i<15)
{
    i++;
    alfa=0.5f*(alfaMax+alfaMin);
    float cosAlfa=cosf(alfa);
    float sinAlfa=sinf(alfa);
    yVypocitane=x*sinAlfa/cosAlfa-0.5f*gravitace*x*x/
        (rychlostStrely*rychlostStrely*cosAlfa*cosAlfa);
    if(yVypocitane<y)
        alfaMin=alfa;
    else
        alfaMax=alfa;
}
```

Výpočet končí, dosáhne-li se požadované přesnosti, nebo pokud se překročí povolený počet iterací.

8 Výkonnostní náročnost

8.1 Výkonnostní náročnost v závislosti na počtu sektorů

Pro přibližnou představu o výpočetní náročnosti použitých algoritmů, zde uvedu počet vykreslených snímků za jednu sekundu. Pro zajímavost budu toto měření provádět několikrát, vždy pro určitou velikost jednoho sektoru, tedy pro určitý celkový počet sektorů, na který se rozdělí krajina (viz kapitola 3.2 Rozdělení terénu na sektory).

Měření jsem provedl na následujících počítačových sestavách:

Počítač A: AthlonXP 2400+, 512MB RAM, ATI Radeon 9600XT

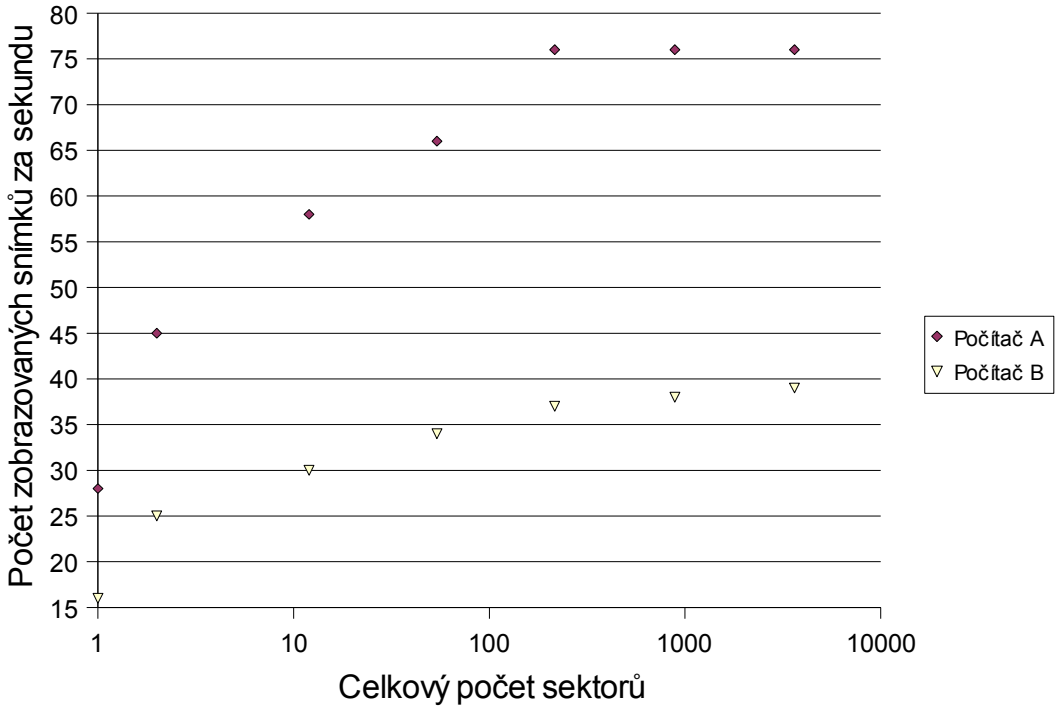
Počítač B: Duron 1200, 512MB RAM, ATI Radeon 9500

Při měření jsem postupoval tak, že jsem spustil aplikaci, chvíli počkal, až se ustálí hodnoty zobrazovaných snímků za sekundu a poté zapsal výsledek. Naměřené výsledky lze vidět v tabulce 8.1. Hodnoty jsem pro názornost vynesl do grafu (viz obrázek 8.1).

Počet sektorů ve směru osy X	Počet sektorů ve směru osy Z	Celkový počet sektorů	Počet zobrazených snímků za sekundu, měřeno na počítači A	Počet zobrazených snímků za sekundu, měřeno na počítači B
1	1	1	28	16
2	1	2	45	25
4	3	12	58	30
9	6	54	66	34
18	12	216	76	37
37	24	888	76	38
74	49	3626	76	39

Tabulka 8.1: Výkonnostní náročnost v závislosti na počtu sektorů

Z naměřených hodnot je patrné, že zvyšování počtu sektorů je přínosné pro zvyšování výkonnosti jen do určité meze. Složitost krajiny ale není ve všech sektorech stejná, proto se může vyšší počet sektorů pozitivně projevit ve složitějších částech mapy.



Obrázek 8.1: Graf výkonnostní náročnosti v závislosti na počtu sektorů

8.2 Výkonnostní náročnost v závislosti na počtu sektorů, bez zobrazení modelů

V předchozí kapitole, jsem ukázal výkonnostní náročnost v reálné aplikaci. Velký díl výkonu počítače byl ale použit na samotné vykreslování modelů scény (madel krajiny + modely tanků). Proto nyní, pro lepší představu o výkonnostní náročnosti použitých algoritmů, provedu měření ještě jednou, ale se scénou, která již nebude obsahovat modely určené pro vykreslování. K tomu stačí z programu odstranit řádky, pomocí kterých se tyto modely načítají. Zde jde například o model zobrazované krajiny:

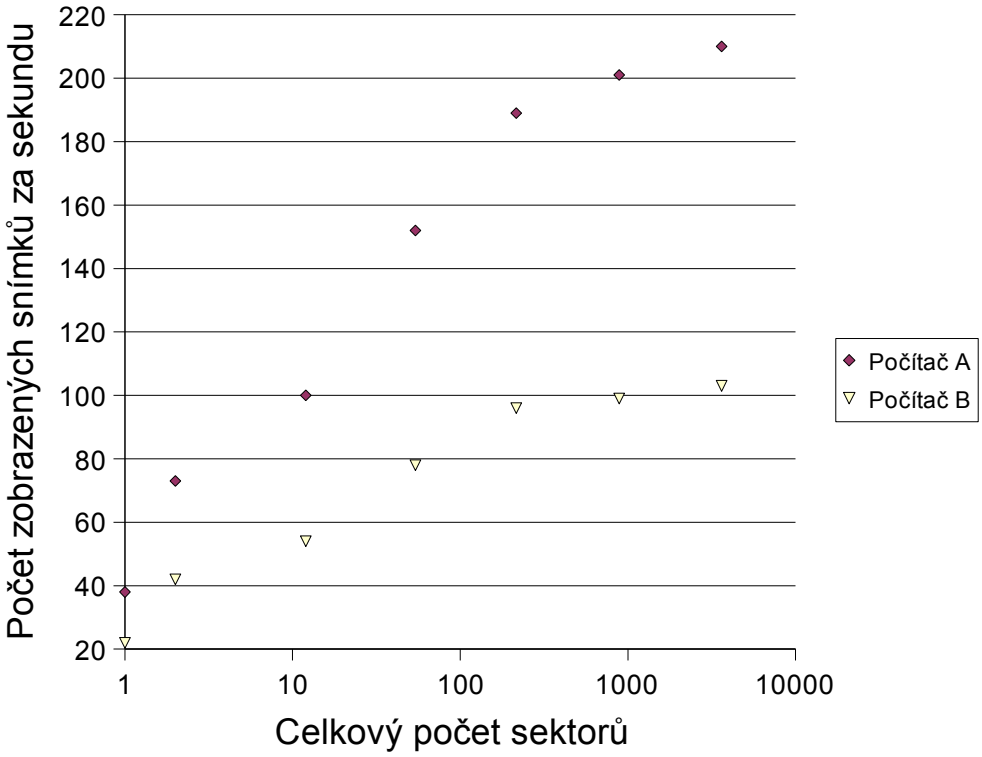
```
//model->name.setValue("models/bludiste.iv");
```

Naměřené hodnoty jsou v tabulce 8.2. Graf je pak znázorněn na obrázku 8.2. Z výsledků lze vypozorovat, že vykreslování scény si ukrojí značnou část výkonu počítače. Jen ještě dodám, že pokud jsem maximalizoval okno aplikace, která zobrazovala modely, na výkonnost to nemělo prakticky žádný vliv. Daná grafická karta tedy neměla se zvětšeným rozlišením aplikace žádné problémy. Velikost okna se přitom zvětšila z hodnoty 400x400 bodů na velikost 1280x1024 bodů. Pokud by se tedy měla snížit výkonnostní náročnost aplikace, bylo by vhodné se zaměřit nejprve na použití výkonnějšího zobrazovacího algoritmu, než jaký je použit v

grafické knihovně Inventoru.

Počet sektorů ve směru osy X	Počet sektorů ve směru osy Z	Celkový počet sektorů	Počet zobrazených snímků za sekundu, měřeno na počítači A	Počet zobrazených snímků za sekundu, měřeno na počítači B
1	1	1	38	22
2	1	2	73	42
4	3	12	100	54
9	6	54	152	78
18	12	216	189	96
37	24	888	201	99
74	49	3626	210	103

Tabulka 8.2:Výkonnostní náročnost v závislosti na počtu sektorů, bez zobrazení modelů



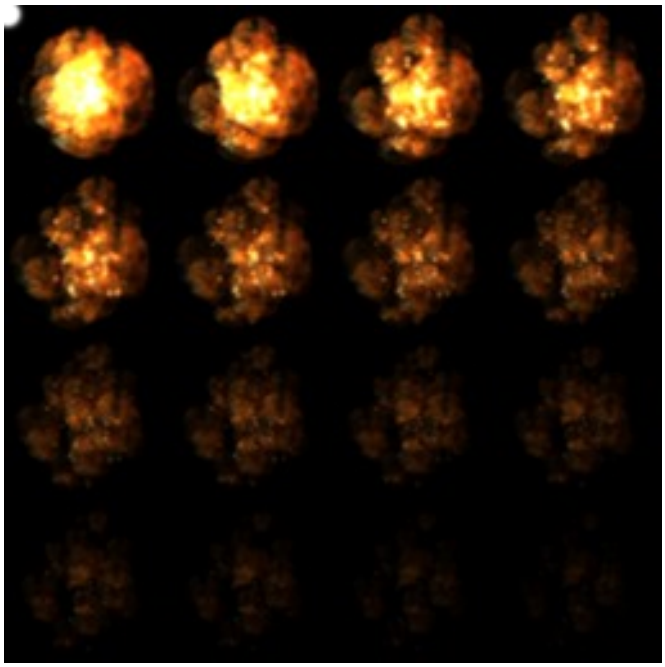
Obrázek 8.2: Graf výkonnostní náročnosti v závislosti na počtu sektorů, bez zobrazování modelů

9 Další použité techniky

9.1 Animace výbuchu

Pro animaci výbuchu použijeme techniku zvanou billboarding, ta je popsaná například v [10]. Vytvoříme třídu `Vybuch`. Ta zobrazí čtverec v místě výbuchu, vytvořený ze dvou trojúhelníků a potažený texturou (obrázek 9.1). Čtverec bude vždy natočený směrem ke kameře. Toho docílíme jednoduše tak, že uzlu výbuchu `rotace` třity `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 9.1: Textura výbuchu

Volba `DELAYED_ADD` zde znamená, že se zobrazení objektu provede 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.

Pro zvýšení plynulosti animace, zařídíme, aby se čtverec s texturou výbuchu plynule zvětšoval v čase. K tomu slouží objekt `scale` třídy `SoScale`:

```
float s = 0.9f + 2.0f*(cas - this->casStartu);
this->scale->scaleFactor.setValue(s,s,s);
```

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 9.1) 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.

9.2 Zvuky

Pro přehrávání zvuku ve scéně slouží v Inventoru třídy `SoVRMLAudioClip` a `SoVRMLSound`. `SoVRMLAudioClip` slouží k načtení a uchování zvukových 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`, který reprezentuje umístění posluchače:

```
posluchac = new SoListener;
root->addChild(posluchac);
```

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

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

Inventor přehrává zvuk přes rozhraní OpenAL [5]. Na platformě Windows se mi stávalo, že se vždy po přehrání zvuku hra 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 (například 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, alespoň ve verzi Coin3D 2.3.0.0). 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);
```

9.3 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

10 Závěr

V této práci jsem navrhl a popsal algoritmy pro pohyb vozidla po nerovném terénu. Algoritmy jsem implementoval pomocí grafické knihovny Open Inventor a vyhodnotil výpočetní náročnost použitého řešení, čímž jsem splnil zadání. Spojením těchto algoritmů s aplikací vytvořenou v mém ročníkovém projektu tak vznikla jednoduchá 3D hra, využívající pohyb vozidla po nerovném terénu, zvuk, komunikaci po síti, jednoduchou umělou inteligenci a fyziku těles. Slabinou použitého řešení je, že povrch, po kterém se pohybuje tank, je reprezentován vždy nejvyššími body modelu krajiny. Tank tedy například nemůže projet pod mostem, protože ten pro něj představuje překážku.

Jak jsem již podotkl v kapitole Výkonnostní náročnost, vykreslování složitého modelu krajiny zabere relativně hodně výpočetního výkonu počítače. Další vývoj projektu by se tedy mohl věnovat urychlení vykreslování složitějších modelů krajin. Zde by se snad dala použít práce Petra Vrby: „Zobrazení krajiny pomocí Chunked-LOD algoritmu“.

Z dalších souběžně řešených projektů by se mohl využít projekt Tomáše Buriana: „Algoritmy pro zobrazování stínů v reálném čase“. Tím by se projekt obohatil o zobrazování stínů a vypadal by zase o něco líp.

Další vývoj by se pak mohl týkat vylepšování fyzikálního modelu, vytvoření přívětivého a efektního menu, přidání grafických efektů jako je například mlha, déšť, nebo osvětlení tmavé scény z reflektoru tanku. Dále by se mohla vylepšit hratelnost samotné hry, například rozšířením zbrojního arzenálu tanků, přidáním různých bonusů, nebo vylepšením umělé inteligence. 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] Wernecke, J.: 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/

[7] Žába, J., Beneš, B., Sochor, J., Felkel, P.: Moderní počítačová grafika, druhé vydání, Brno, Computer Press, 2004, ISBN: 80-251-0454-0

[8] Walsch, P.: Advanced 3D Game Programming with DirectX 9.0, Wordware Publishing, Inc., 2003, ISBN: 1556229682

[9] Snook, G.: Real-Time 3D Terrain Engines Using C++ and DirectX 9, Charles River Media, Inc., 2003, ISBN: 1584502045

[10] Lengyel, E.: Mathematics for 3D Game Programming and Computer Graphics, Second Edition, Charles River Media, Inc., 2004, ISBN: 1584502770