

Simulátor vrtulníku - Grafický engine

Josef Musil

2005



Univerzita Tomáše Bati ve Zlíně
Fakulta technologická

TÉMA PRÁCE: Simulátor vrtulníku - Grafický engine

VEDOUCÍ PRÁCE: Ing. Michal Bližňák

Navrhněte a pomocí jazyka C/C++, knihovny OpenGL a dalších vhodných softwarových knihoven vytvořte plugin pro vizualizaci letu vrtulníku v prostoru. Plugin bude sloužit pro zobrazení letu vrtulníku z pohledu pilota a třetí osoby a bude součástí systému "Simulátor vrtulníku" pro MS Windows. Při práci postupujte podle následujících bodů:

1) Vytvořte rešerši na téma vizualizace 3D prostoru a zobrazení rozsáhlých scén. Popište a srovnajte nejpoužívanější algoritmy pro vykreslování 3D modelů a scén pomocí knihovny OpenGL a algoritmy pro optimalizaci vykreslování rozsáhlých scén.

2) Navrhněte a vytvořte samostatný plugin pro systém Simulátor vrtulníku s následujícími vlastnostmi:

a) Definujte aplikační rozhraní mezi pluginem a jádrem simulátoru.

b) Grafický engine zapouzdřený v enginu bude umožňovat:

i) Zobrazit exteriér - scénu (volnou krajinu) ve 3D. Pro optimalizaci vykreslování bude

použit jeden z výše popsaných algoritmů

ii) Zobrazení dalších samostatných objektů vložených do scény

iii) Zobrazení jednoduchých částicových systémů pro simulaci kouře, ohně, atd.

iv) Zobrazení 3D scény exteriéru z pohledu pilota vrtulníku i třetí osoby

c) Vlastní plugin vytvořte jako dll soubor.

3) Vytvořte dokumentaci ve formě UML diagramů popisujících vnitřní strukturu a činnost pluginu a jeho návaznost na jádro simulátoru.

Souhlasím s tím, že s výsledky mé práce může být naloženo podle uvážení vedoucího bakalářské práce a ředitele ústavu. V případě publikace budu uveden jako spoluautor.

Prohlašuji, že jsem na celé bakalářské práci pracoval samostatně a použitou literaturu jsem citoval.

Ve Zlíně, 10. 06. 2005

.....

podpis

RESUMÉ

Práce uvádí čtenáře do problematiky vizualizace 3D prostoru. V teoretické části popisuje nejpoužívanější algoritmy pro vykreslování 3D modelů a scén a algoritmy pro optimalizaci vykreslování rozsáhlých scén. Praktická část obsahuje návrh samostatného pluginu pro simulátor vrtulníku a popis rozhraní pro komunikaci s jádrem simulátoru.

SUMMARY

This work introduces reader into issues of 3D space visualization. Theoretical part describes most used algorithms for rendering of 3D models and sceneries and algorithms for optimised rendering of large scenes including Level of Detail techniques. Practical part contains suggestions for creation of stand alone plug-in for helicopter simulator and description of communication interface between the plug-in and the core application.

Na tomto místě chci poděkovat vedoucímu bakalářské práce Ing. Michalu Bližňákovi za odborné vedení, cenné rady a pomoc při řešení problémů souvisejících s touto prací.

ÚVOD.....	8
I. TEORETICKÁ ČÁST	9
1 DATOVÉ FORMÁTY MODELŮ	10
1.1 MD2	10
1.2 MD3 A MD4	10
1.3 3DS.....	10
2 ANIMACE MODELŮ VE SCÉNĚ.....	12
3 OPTIMALIZACE VYKRESLOVÁNÍ ROZSÁHLÝCH SCÉN.....	13
3.1 FRUSTUM CULLING – OŘEZ POHLEDOVÝM JEHLANEM	13
3.2 Z-BUFFERING	13
3.3 OCCLUSION CULLING	14
3.4 OCTAL TREE - OCTREE	15
4 GENEROVÁNÍ TERÉNU	17
4.1 GENEROVÁNÍ TERÉNU Z VÝŠKOVÉ MAPY	17
4.2 ALGORITMY PRO GENEROVÁNÍ TERÉNU	17
4.2.1 Parcelace – Plasma algoritmus	17
4.2.2 Skládání funkcí.....	18
4.2.3 Fault algoritmus.....	18
4.2.4 Kruhový algoritmus.....	20
5 ZOBRAZENÍ TERÉNU LOD ALGORITMY	22
5.1 TIN – TRIANGULATED IRREGULAR NETWORK – NEPRAVIDELNÁ TROJÚHELNÍKOVÁ SÍŤ	22
5.1.1 Získání TIN sítě.....	22
5.1.2 Optimalizace	23
5.1.3 Level of detail – edge collaps.....	23
5.1.4 Triangle stripy	24
5.1.5 Návaznosti parcel v různých úrovních detailů	24
5.2 ROAM - REAL-TIME OPTIMALLY ADAPTING MESHES	26
5.2.1 Binární trojúhelníkový strom	26
5.2.2 Postup vykreslování	29
II. PRAKTICKÁ ČÁST	31
6 NÁVRH SAMOSTATNÉHO PLUGINU PRO SYSTÉM SIMULÁTORU VRTULNÍKU.....	32
6.1 DYNAMICKY LINKOVANÉ KNIHOVNY – DLL	32
6.2 NÁVRH ROZHRANÍ PRO KOMUNIKACI MEZI PLUGINEM A JÁDREM SIMULÁTORU	32
6.2.1 Statické údaje	32
6.2.2 Dynamické údaje	33
6.2.3 Detekce kolizí.....	33
6.3 POPIS PLUGINU	34

6.3.1	Základ.....	34
6.3.2	Exportované funkce.....	34
6.3.3	Třída CScene	34
6.3.4	Třída Terrain	35
6.3.5	Třída CFrustum	37
6.3.6	Třída CTGA	37
6.4	ZOBRAZENÍ JEDNODUCHÉHO ČÁSTICOVÉHO SYSTÉMU	37
6.4.1	Popis částice	38
6.4.2	Inicializace objektu	38
6.5	ZOBRAZENÍ SCÉNY Z POHLEDU PILOTA I TŘETÍ OSOBY,	39
7	UML DOKUMENTACE	40
7.1	BĚH APLIKACE.....	40
7.2	TŘÍDY MODELŮ.....	41
8	ZÁVĚR.....	42
	SEZNAM POUŽITÉ LITERATURY	43
	SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK	44
	SEZNAM OBRÁZKŮ	45

ÚVOD

Vizualizace 3D prostoru v současné době nachází použití v mnoha odvětvích lidské činnosti. Namátkou jmenujme například geografické informační systémy (GIS) a filmový nebo herní průmysl.

Postupy pro zobrazování terénu se vyvíjely společně s rozvojem výpočetní techniky. Od prvních pokusů připomínajících spíše shluk pixelů, přes vektorové vykreslování, technologii voxelů až po současnost, kdy stále větší výpočetní síla umožňuje přiblížit se více realitě a vytvářet fotorealistické scény.

I přes velký výkon současných procesorů a grafických karet je stále nutné vykreslování scény v reálném čase optimalizovat. K tomu slouží na straně jedné algoritmy pro vykreslování scény v různých stupních detailů, na straně druhé postupy pro eliminaci nadbytečných polygonů.

Tato práce ve své teoretické části pokrývá problematiku algoritmů pro vykreslování proměnlivé úrovně detailů terénu, popisuje postupy pro optimalizaci vykreslování rozsáhlých scén, uvádí čtenáře do animace objektů ve scéně a do formátů pro popis objektů.

V praktické části se zabývá návrhem samostatného pluginu pro projekt simulátoru vrtulníku v jazyce C++, popisují jeho tvorbu a definici rozhraní pro komunikaci mezi tímto pluginem a jádrem a následně popisují jeho hlavní třídy a strukturu.

I. TEORETICKÁ ČÁST

1 DATOVÉ FORMÁTY MODELŮ

Mezi nejčastější metody pro načítání modelů do scény patří požití souborů ve formátu md2 nebo md3. Tyto formáty byly vytvořeny společností ID Software pro engine hry Quake 2 (MD2) a Quake3: Arena (MD3, MD4). Dalším často používaným formátem je 3ds, což je výstup z programu 3D Studio Max firmy Autodesk.

1.1 MD2

Soubor formátu md2 se skládá z hlavičky souboru a samotných dat. Hlavička má formát struktury a obsahuje informace o velikost datové části, verzi formátu, šířce a výšce textury, počtu trojúhelníku na frame, informaci o počtu skinů (vzhledů) asociovaných s modelem, počtu framů atd. Každý frame obsahuje informace o pozici všech trojúhelníků. Formát md2 má omezen počet trojúhelníků na 4096, počet koordinátů textury na 2048, počet framů na 512 a počet skinů na 32.

1.2 MD3 a MD4

MD3 formát dělí objekt, např. postavu, na jednotlivé části, tedy torzo, hlavu a končetiny. Tyto části jsou spojeny až během nahrávání souboru. Základem modelu jsou tagy, jakési pseudo-kosti, malé trojúhelníky sloužící jako objekty ke kterým připínáme další části modelu.

V Quake 3: Arena od verzi 1.29 výše ID Software uvádí nový formát pod názvem MD4, který obsahuje kosterní animaci, definovanou pomocí translací a rotací namísto popisu jednotlivých framů.

1.3 3DS

Soubor formátu 3DS se skládá z bloků dat zvaných chunks (kousky). Každý kousek začíná stejným způsobem:

```
1 short chunk_id;  
2 long chunk_len;
```

chunk_id je unikátní číslo identifikující typ dat obsažených v chunku a může také obsahovat informaci o podřízených kouscích. Chunk_len obsahuje velikost následujících dat asociovaných s aktuálním chunkem.

2 ANIMACE MODELŮ VE SCÉNĚ

V minulosti se vytvářely především pouze statické modely scén a používaly se dnes již tradiční metody modelování těles, například šablonování nebo konstruktivní geometrie těles. Nejčastějším způsobem animace byla metoda klíčování a následné interpolace tvaru a vlastností objektu mezi klíčovými snímky.

Dnes se častěji používá pokročilejších způsobů modelování trojrozměrných těles i celých scén. Využívají se NURBS (Non-Uniform Rational B-Splines/Surfaces) křivky a plochy, implicitní plochy, procedurální modelování a fraktální geometrie. Do popředí se také dostávají metody umožňující s již vytvořenou scénou provádět různé fyzikální simulace.

Postup práce při fyzikální simulaci v trojrozměrné scéně je takový, že uživatel v interaktivním grafickém editoru zadá geometrické a fyzikální parametry jednotlivých těles a na tato tělesa se potom aplikují globální a lokální síly, které dané těleso deformují a mění tak jeho polohu nebo další vlastnosti tělesa (barva, průhlednost, textura apod.). Některé simulační metody umožňují po působení síly provést i změnu topologie tělesa, čímž je možné modelovat například průstřely, zlomy či slepení částí tělesa.

Mezi často používané metody pro simulaci patří inverzní kinematika (Inverse Kinematics), která uživateli nabízí možnost měnit tvar tělesa resp. vzájemnou polohu jednotlivých částí tělesa zadáním polohy klíčových bodů na tělese. Klíčové body na tělese se nazývají vrcholy či prvky kostry. Následně dopočítáme výsledný tvar tělesa. Pro realistické animace je vhodné použít inverzní kinematiku v kombinaci s dynamikou.

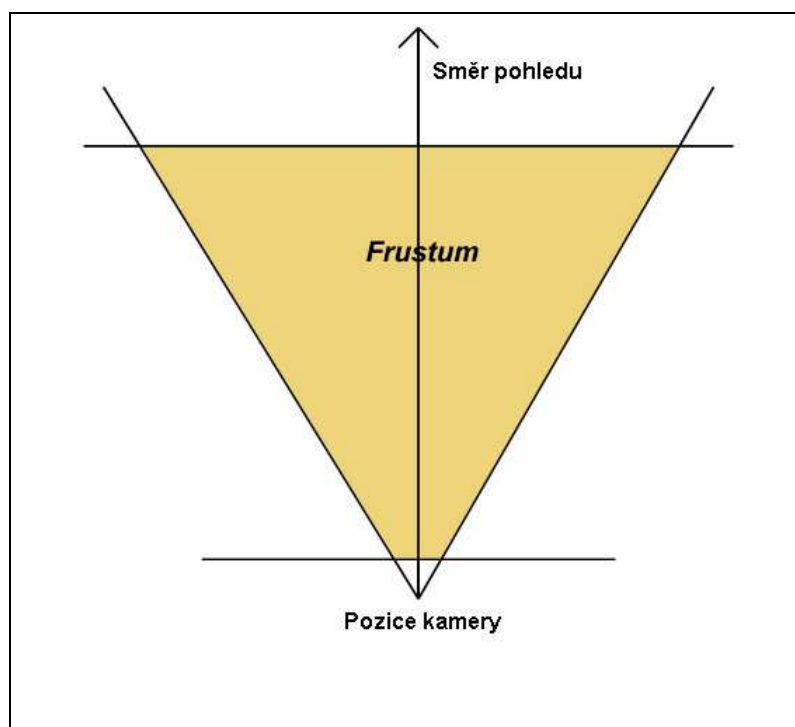
Nevýhodou klasické inverzní kinematiky je její omezení pouze na tuhá tělesa, což omezuje její použití.

3 OPTIMALIZACE VYKRESLOVÁNÍ ROZSÁHLÝCH SCÉN

3.1 Frustum culling – ořez pohledovým jehlanem

Frustum culling urychluje vykreslování scény tím, že grafický subsystém nevykresluje trojúhelníky mimo naše zorné pole.

View frustum – část scény viditelná kamerou. Má tvar komolého jehlanu. Normálové vektory ploch tvořících jehlan směřují všechny ven nebo všechny dovnitř.



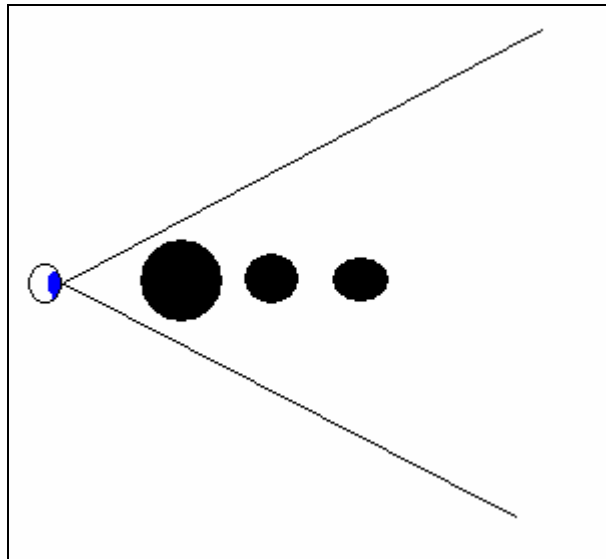
Obr. 1 – Pohledový jehlan

Pomocí analytické geometrie zjišťujeme polohu bodu vzhledem k ořezovému jehlanu - bod musí ležet ve správné polorovině vzhledem ke všem šesti plochám ořezového jehlanu nebo polohu koule - testujeme souřadnice středu koule, resp. souřadnice středu koule + poloměr koule. Body které se v ořezovém jehlanu nenacházejí se ve výsledné scéně nevykreslují.

3.2 Z-buffering

Z-buffering je algoritmus zajišťující zachování perspektivy scény. Funguje na principu testování Z souřadnice pixelu a porovnávání aktuální pozice pixelu s dalšími pixely

uloženými v z-bufferu. Pixely které jsou v zákrytu vykresleny nebudou. Nevýhoda tohoto postupu popíšeme na následujícím obrázku.



Obr. 2 – Z-buffering

Z pohledu pozorovatele je viditelná pouze první koule, ale protože se o viditelnosti rozhoduje až v z-bufferu, je nutné spočítat a vyrenderovat všechny tři objekty. Tuto vlastnost řeší occlusion culling.

3.3 Occlusion Culling

Occlusion culling algoritmy jsou založeny na odstranění objektů, které jsou v zákrytu. Tzn. objekty, které se sice nacházejí uvnitř ořezového jehlanu, ale nebudou vidět ve výsledné scéně. Výhodou oproti z-bufferingu je, že tato operace probíhá ještě před odesláním dat do z-bufferu. Postup je vidět v následujícím pseudokódu.

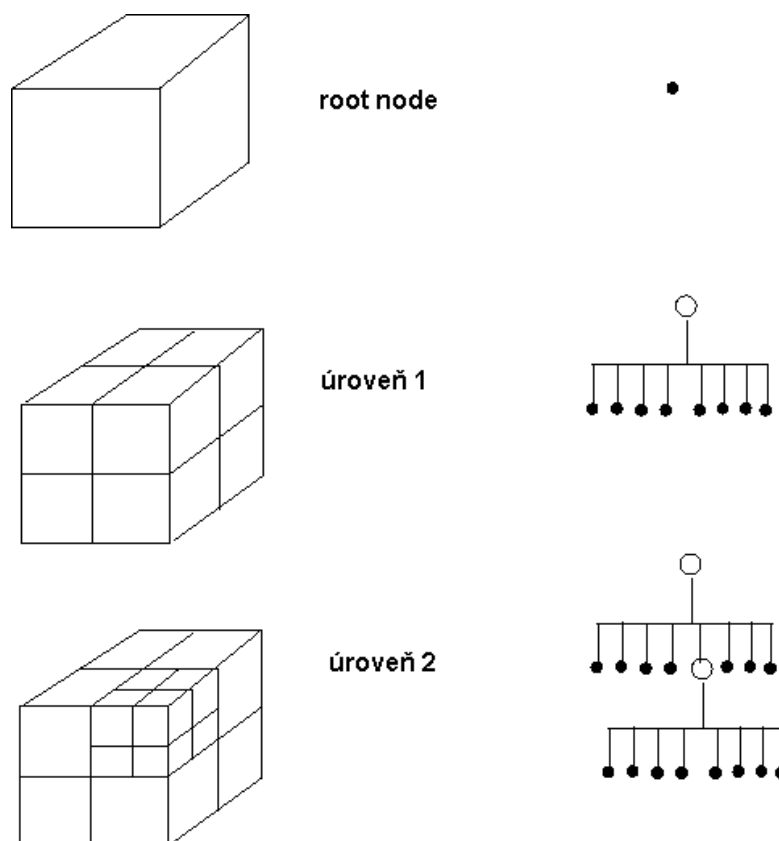
```
1 OcclusionCullingAlgorithm (G)
2 for each object g in G
3   if(isOccluded(g, OR))
4     Skip(g)
5   else
6     Render(g)
7     Update(OR, g)
8   end
9 end
```

G značí vykreslovanou scénu, g objekt ve scéně a OR označuje okluzor (zakrývající předmět). Nejprve otestujeme je-li objekt v zákrytu. Je-li zakryt, vyřadíme ho z dalšího zpracování. Pokud zakryt není, musí být vykreslen.

3.4 Octal Tree - Octree

Octree je založen na principu rozdělení 3D scény na oblasti a po aplikaci frustum culling umožňuje vykreslit pouze ty části scény, které jsou v zorném poli kamery.

Octree pro dělení používá krychle. Na nejvyšší úrovni (root node) je krychle, která obsahuje kompletní 3D scénu a tedy i všechny její vertexy. V následujícím kroku tuto krychli rozdělíme na osm menších krychlí a takto postupuje až do požadované úrovně dělení.



Obr. 3 – Octree

Kriteria pro ukončení dělení:

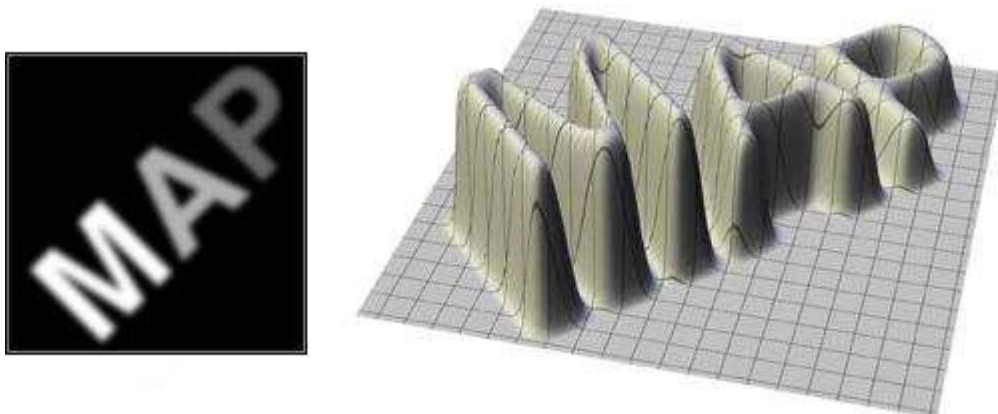
- 1) Dělení uzlu ukončíme, je-li počet jeho trojúhelníku menší než námi definovaná hodnota. Např. je zbytečné další dělení uzlu, který obsahuje pouze dva trojúhelníky. V tomto případě všechny zbylé trojúhelníky přiřadíme tomuto uzlu.
- 2) Stanovíme si maximální úroveň dělení, po jejím dosažení alokujeme zbylé trojúhelníky koncovým uzlům.
- 3) Stanovíme si maximální počet uzlů. S každým novým uzlem testujeme dosažení této hranice. Po jejím dosažení ukončíme dělení a koncovým uzlům přiřadíme zbývající trojúhelníky.

4 GENEROVÁNÍ TERÉNU

4.1 Generování terénu z výškové mapy

Pod pojmem výšková mapa (heightmapa) rozumíme 2D reprezentaci terénu formou bitmapy, kde jednotlivé odstíny šedi vyjadřují výšku terénu na dané souřadnici. Komerční systémy dokáží prakticky bez zásahu člověka vytvořit z leteckých nebo satelitních snímků krajiny prostorový digitální model terénu.

Digitalizace mapy je technika, při níž získáváme z mapy její výškopisnou složku. Jednou z technik jak získat výškovou souřadnici je digitalizovat mapu po vrstevnicích s předem definovanou výškou pro každou vrstevnici. Zde je nevýhodné, že se terén nesnímá po svých charakteristických bodech. Druhou možností je snímat terén podle charakteristických linií, kdy je nutné každému sejmutému bodu přiřadit výšku.



Obr. 4 – Generování terénu z heightmapy

4.2 Algoritmy pro generování terénu

4.2.1 Parcelace – Plasma algoritmus

Jednou z nejpopulárnějších metod tvorby terénu je rekurzivní parcelace. Tvorba probíhá takto: vezmeme čtverec a do jeho rohů umístíme vrcholy o náhodné výšce. Poté spočítáme vrcholy ve středech stran čtverce a uprostřed čtverce lineární interpolací. Následně k těmto hodnotám přičteme nebo odečteme náhodně vygenerovanou hodnotu. Tento postup rekurzivně opakujeme, ale při každé iteraci přičítáme nebo odečítáme menší hodnoty než v předchozím kroku. Výhodou tohoto postupu je jeho rychlost, nevýhodou je nemožnost

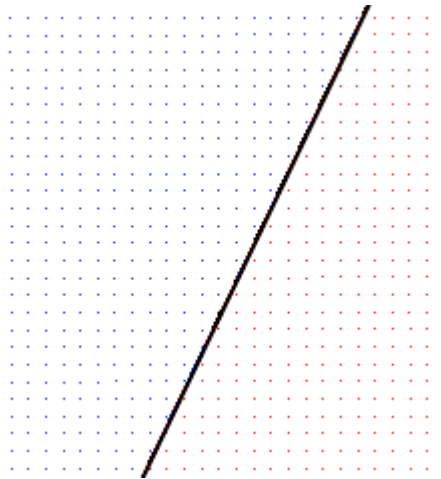
generovat koryta řek a podobné útvary. Problémové je také spojování takto vytvořených parcel.

4.2.2 Skládání funkcí

Tato metoda používá goniometrických funkcí sinus a kosinus. Základem je určení funkční hodnoty v jednotlivých bodech terénu. Výhodou je možnost určení výškové hodnoty v libovolném místě prostým dosazením, možnost generování nekonečného terénu, který je také jemnější. Nevýhodou je velká spotřeba výpočetního času.

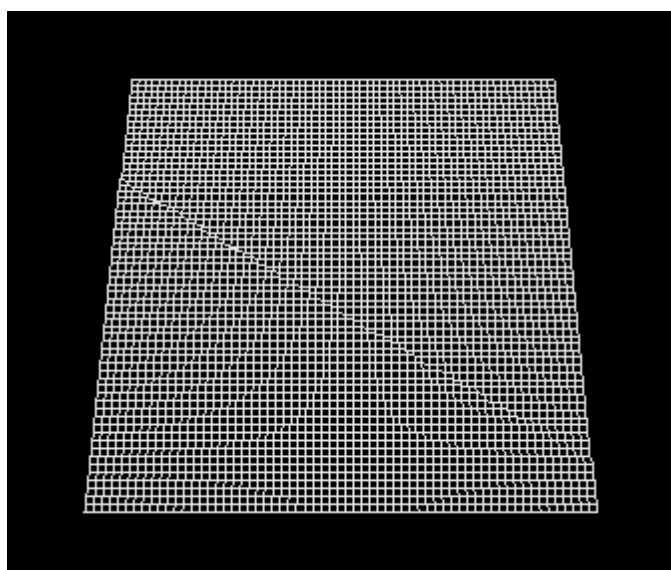
4.2.3 Fault algoritmus

Na počátku máme čtvercovou scénu s nulovou výškou. Do této scény umístíme přímkou, která plochu rozdělí na dvě nestejně velké oblasti.

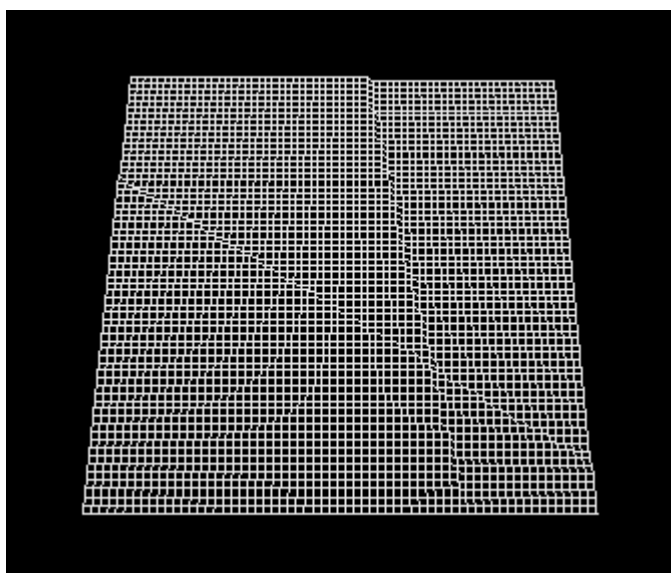


Obr. 5 – Rozdělená scéna

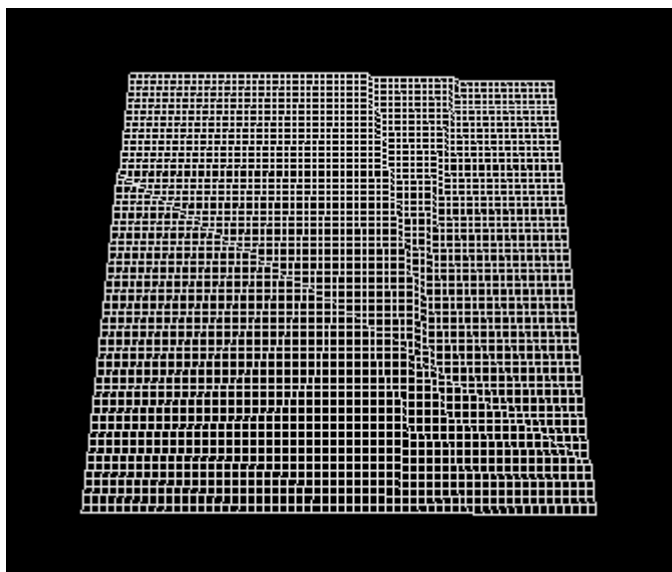
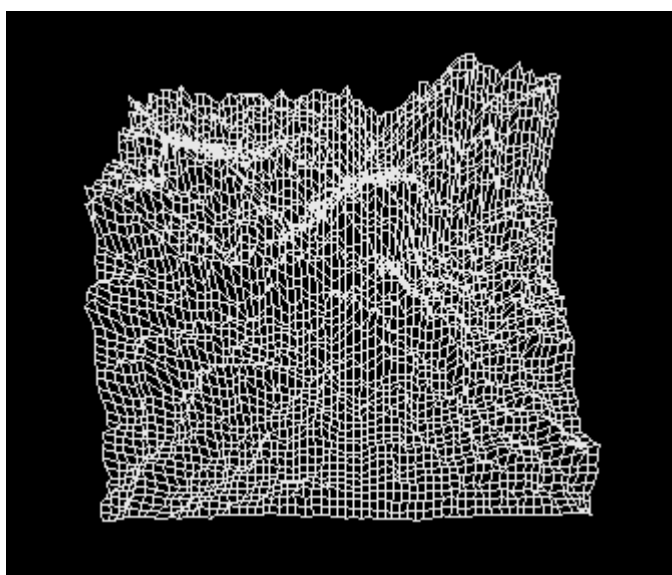
Bodům s červenou barvou výšku snížíme a modrým bodům naopak výšku přidáme. Tento postup opakujeme až do námi požadované úrovně .



Obr. 6 – Po 1. iteraci

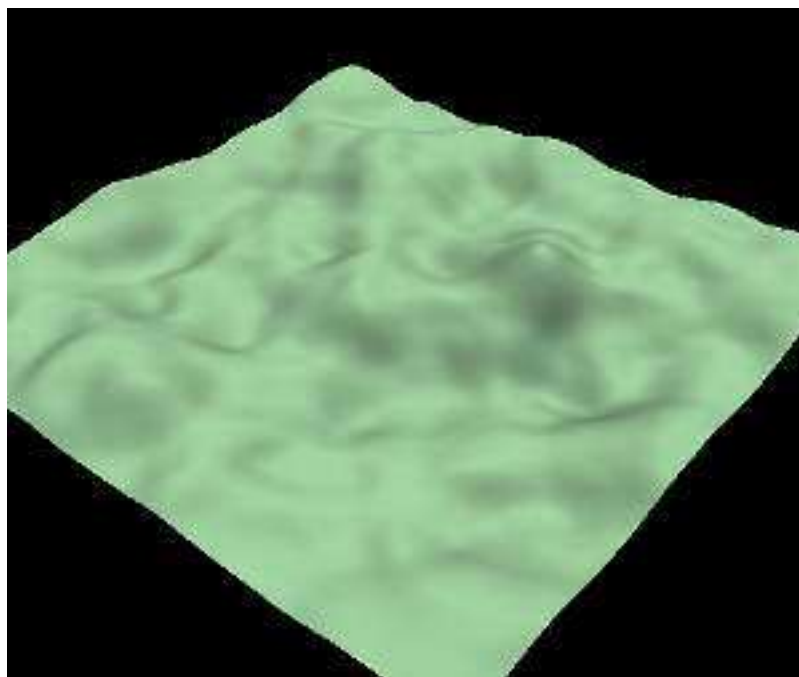


Obr. 7 – Po 2. iteraci

*Obr. 8 – Po 3. iteraci**Obr. 9 – Po 400. iteraci*

4.2.4 Kruhový algoritmus

Kruhový algoritmus je podobný předchozímu postupu. Jeho základem je namísto přímky kružnice. Body uvnitř kružnice zvýšíme. Posouváním kruhu po scéně společně s náhodným počtem zvyšování polohy vznikne plastický terén.



Obr. 10 – Po 1000. iteraci

5 ZOBRAZENÍ TERÉNU LOD ALGORITMY

5.1 TIN – Triangulated Irregular Network – Nepravidelná trojúhelníková síť

Při generování terénu z klasické výškové mapy je obtížné zajistit vysokou detailnost scény. Objekty jako koryta řek nebo skalní útvary vyžadují vysoce detailní výškové mapy. Řešením tohoto problému je použití TIN – nepravidelné trojúhelníkové sítě.

5.1.1 Získání TIN sítě

TIN síť je síť trojúhelníků na náhodně zvolených místech terénu. Body určující tvar terénu mají větší hustotu v oblastech zvrásněného terénu a nižší hustotu v rovinných částech scény. Body jsou spojeny úsečkami tak, aby tvořily trojúhelníky.

Pro uchování TIN modelu se používají tři druhy struktur:

1) Trojúhelníková struktura:

- ID trojúhelníku

- Tři ID vrcholů a jejich souřadnice

- Sousedící trojúhelníky

2) Bodová struktura

- ID bodu

- Souřadnice bodu

- Sousedící body

3) Struktura založená na stranách trojúhelníku

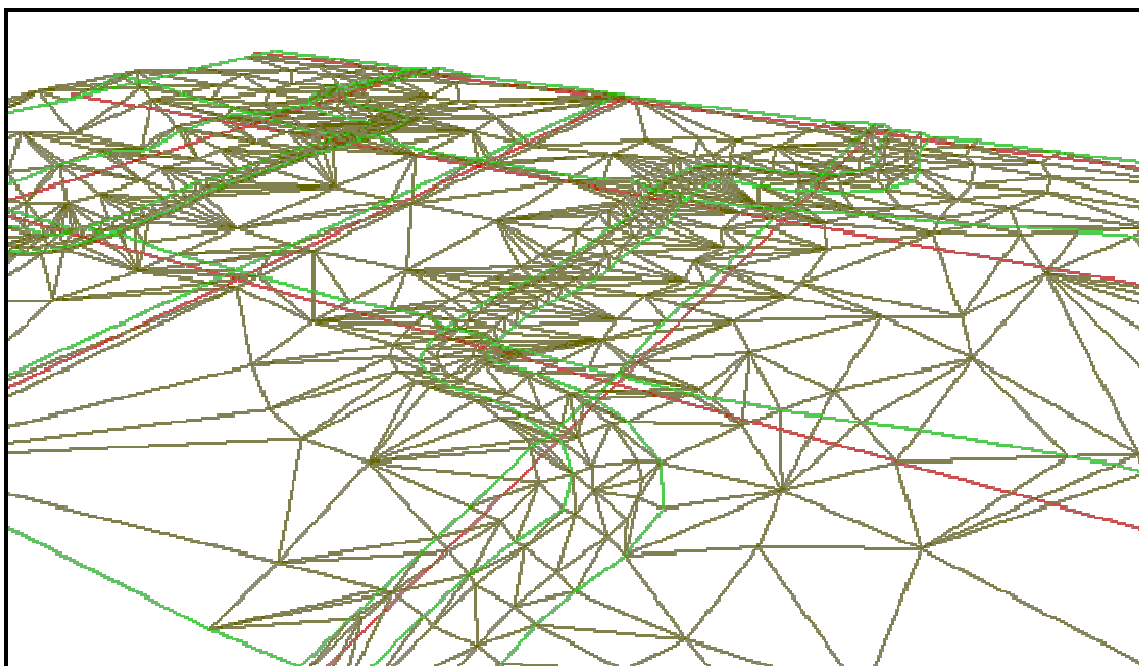
- Body s jejich souřadnicemi a ID

- Trojúhelníky popsány souřadnicemi a ID

- Strany popsány ID, dvě ID bodů a sousedících trojúhelníků

5.1.2 Optimalizace

Začneme rozdělením TIN scény na čtvercové oblasti. Trojúhelníky, které přesahují z jednoho čtverce (parcely) do druhého se rozříznu. Rozřezání trojúhelníků na hranici parcel je potřebné pro generování LOD. Parcely by neměly být příliš velké, neboť příliš velké rozměry způsobují větší nároky při načítání a vykreslování. Každá parcela by měla optimálně obsahovat řádově tisíc trojúhelníků. Rozdělení parcel je dále možno využít při testování viditelnosti ořezovým jehlanem – frustum culling.

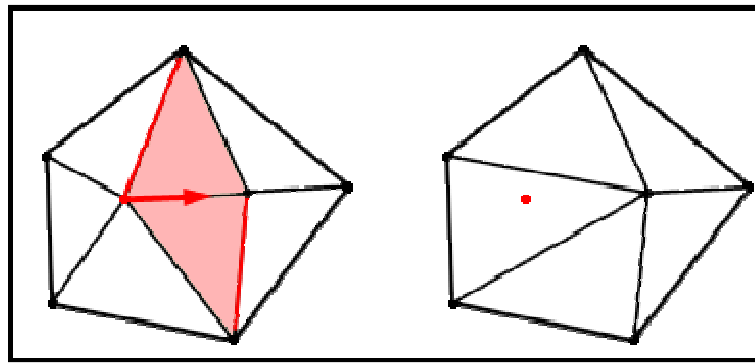


Obr. 11 – TIN scéna po rozřezání na parcely

Světle zelené hrany leží na okraji parcely, červenou barvou je znázorněna mřížka parcel.

5.1.3 Level of detail – edge collapse

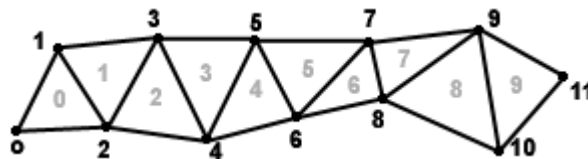
Dalším krokem je vytvoření LOD pro jednotlivé parcely. Použijeme kolaps hran: mají-li dva trojúhelníky společnou hranu, tuto hranu odstraníme. Několikanásobným kolapsem hran získáme požadovaný počet úrovní detailu.



Obr. 12 – Edge collapse

5.1.4 Triangle stripy

Dalším bodem ve vytváření naší TIN sítě je vygenerování pásů trojúhelníků (Triangle Stripů). Při kreslení trojúhelníků v Triangle Stripu je pro zadání n trojúhelníků potřeba $n+2$ vertexů (klasickým způsobem je to pro n trojúhelníků $n*3$ vertexů).

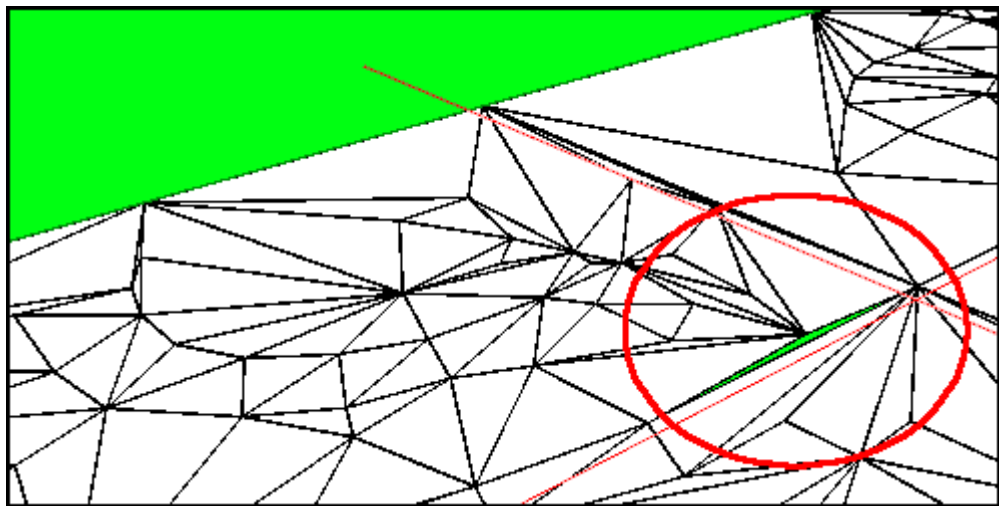


Obr. 13 – Triangle strip

Pro hledání triangle stripů v trojúhelníkové síti lze využít např. knihovnou NvTriStrip od NVIDIA nebo najdeme pásy sami. Pás začne v trojúhelníku, který má nejmeně sousedících trojúhelníků a pokračujeme do dalšího trojúhelníku, který má taktéž co nejmeně sousedů a není dosud užít v žádném pásu. Když nemůžeme nalézt pokračovatele, ukončíme strip a hledáme další.

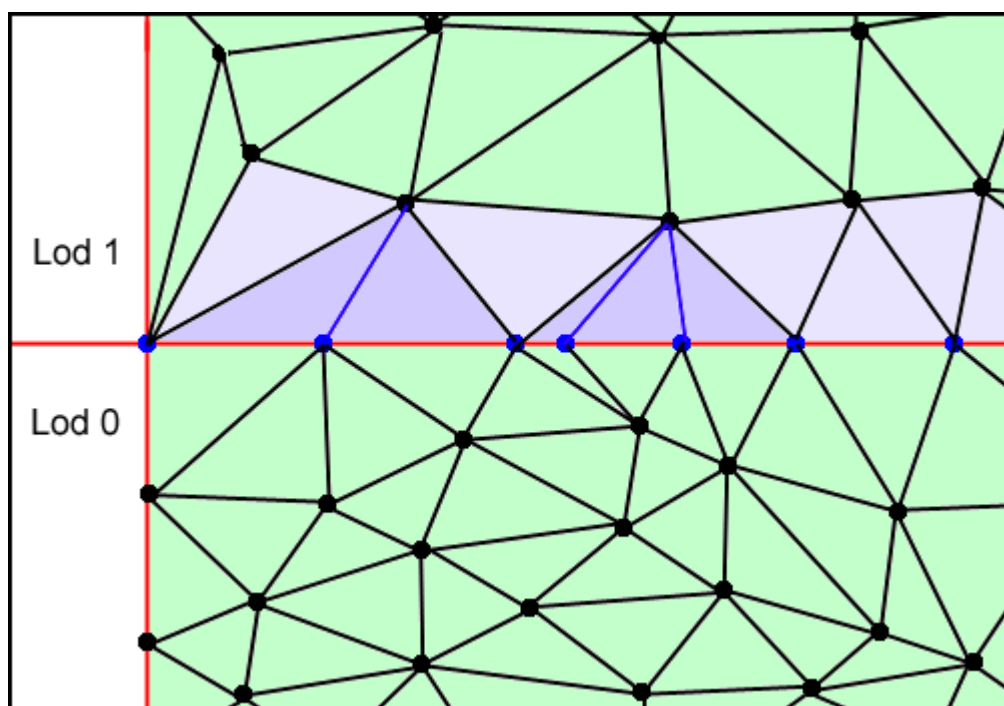
5.1.5 Návaznosti parcel v různých úrovních detailů

Na hranici dvou parcel s odlišnými úrovněmi může vzniknout nepříjemný problém – trhliny a „T - vertexy“.



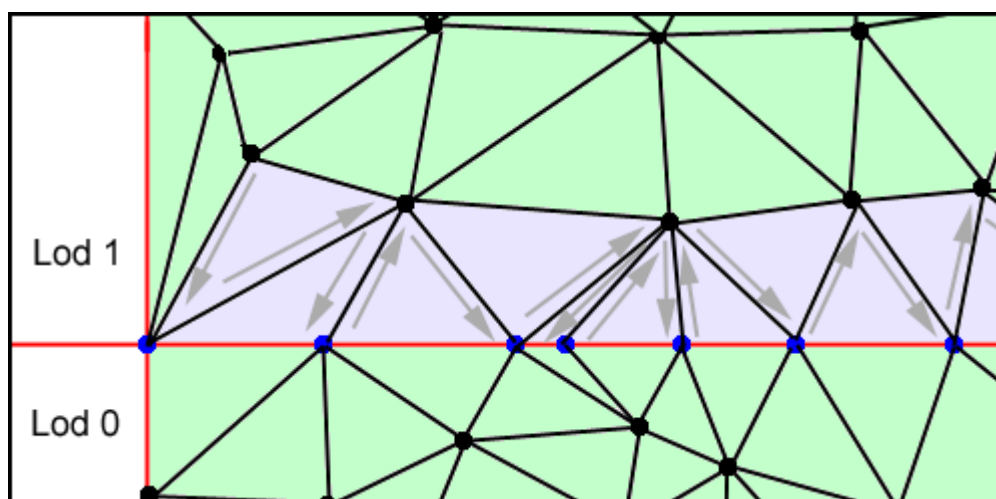
Obr. 14 – Trhlina vzniklá na hranici parcel

Tento problém jde napravit pomocí dělicích pásek. Parcela v nižším Lodu použije dělicí pásek k adaptaci na parcelu ve vyšším Lodu. Pokud mají dvě sousedící parcely stejnou úroveň detailů, nemusí se na sebe adaptovat (návaznost zajistíme už během inicializace). Jak takový dělicí pásek vypadá a jak ho hledat? Najdeme jej tak, že testujeme všechny trojúhelníky, které mají nějaké body na hranici parcel. Pokud má trojúhelník 2 body na hranici, zjišťujeme, zda nejsou mezi nimi ještě nějaké jiné. Pokud ano, musíme trojúhelník rozdělit na několik menších.



Obr. 15 – Adaptace parcel

Při vykreslování parcely zjišťujeme v jaké úrovni detailu je každá sousední parcela. Pokud je ve vyšším detailu, vykreslíme pouze ty trojúhelníky, které nejsou na hranici (na obrázku 16 jsou to ty zelené) a místo trojúhelníků, které jsou na hranici vykreslíme dělicí pásek. Dělicí pásek můžeme vykreslit i jako triangle strip (je to rychlejší) užijeme-li opět degradovaných trojúhelníků.



Obr. 16 – Užití triangle stripů

Pásky generujeme pro každou úroveň detailu (s výjimkou prvního) 4x, přičemž parcely na kraji mřížky jich budou mít méně.

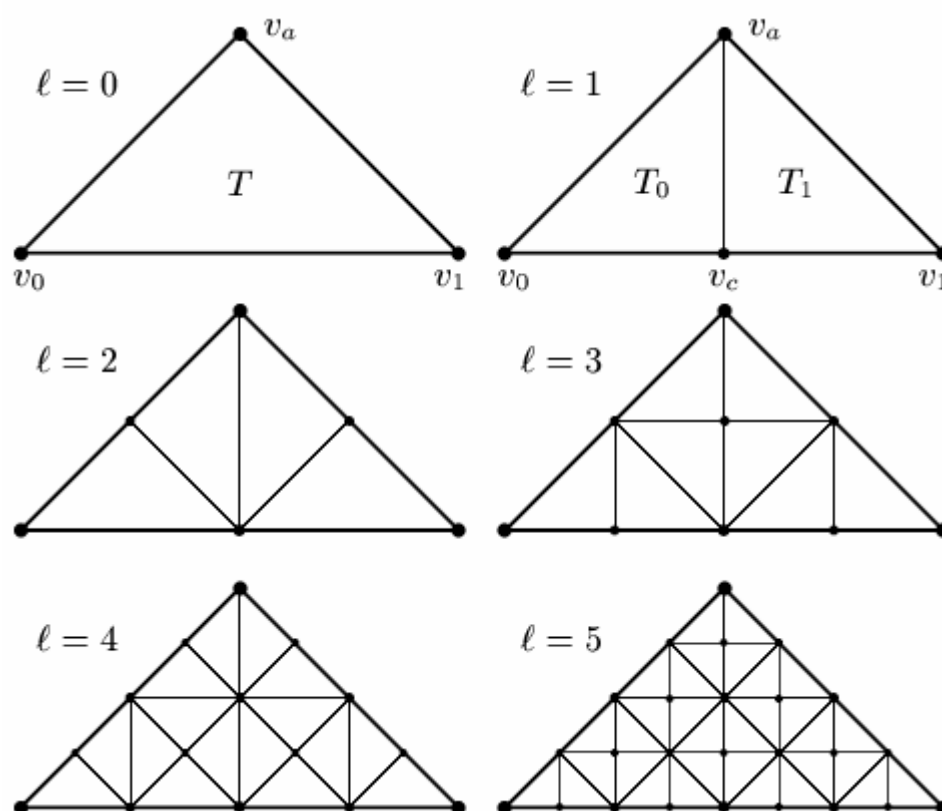
Mezi výhody TIN algoritmu patří možnost práce s velmi rozlehlým terénem. Díky rozdělení na jednotlivé parcely je můžeme za běhu programu podle potřeby nahrávat resp. uvolňovat.

Pozn. Obrázky v této kapitole jsem čerpal z [11].

5.2 ROAM - Real-time Optimally Adapting Meshes

5.2.1 Binární trojúhelníkový strom

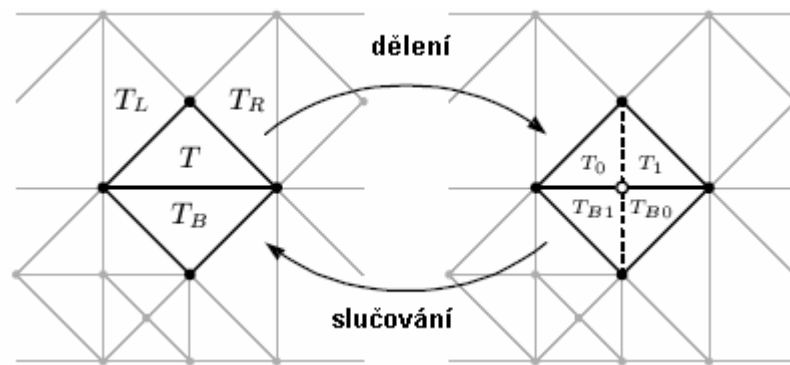
Roam pro zobrazení povrchu používá Triangle Binary Tree (trojúhelníkový binární strom). Na následujícím obrázku vidíme několik počátečních úrovní trojúhelníkového binárního stromu. L značí stupeň dělení.



Obr. 17 – Dělení stromu

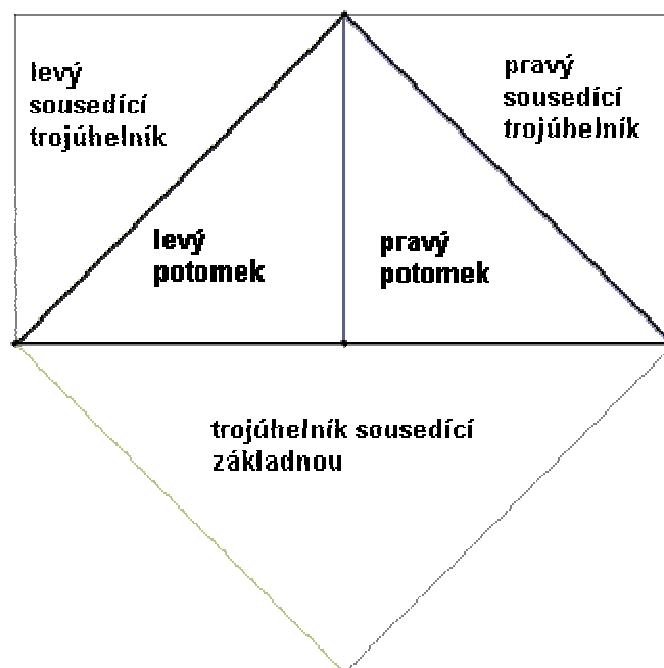
Kořenový uzel $T=(v_0, v_1, v_a)$ je definován jako rovnoramenný pravoúhlý trojúhelník. Rozdělením kořenového uzlu T pomocí úsečky spojující střed základny v_c a vrchol v_a vzniknou trojúhelníky T_0 a T_1 . Ty nazveme levým a pravým potomkem. V dalším dělení pokračujeme rekurzí.

Tvorba scény probíhá tak, že každému vrcholu trojúhelníkového binárního stromu přiřadíme jeho souřadnice ve vykreslovaném prostředí. Několik takových trojúhelníků vytvoří spojitou část povrchu, tento proces nazveme triangulací. Obr. 18 ukazuje okolí trojúhelníku T během triangulace.



Obr. 18 – Dělení a slučování

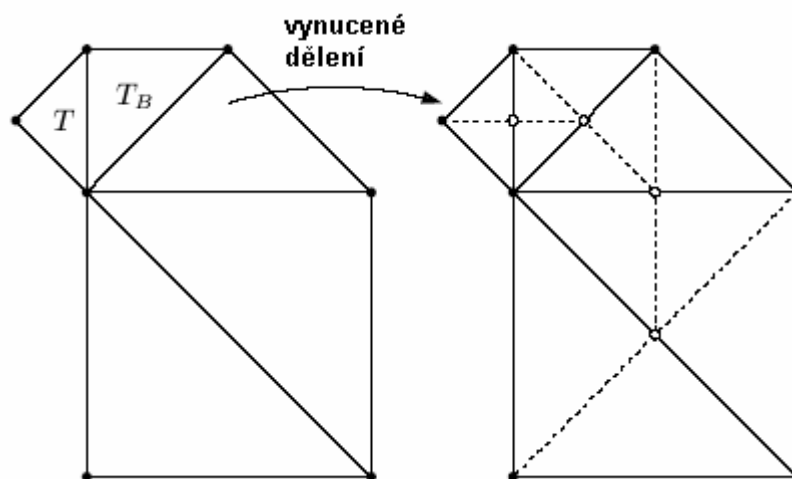
Důležitou skutečností je, že sousedící trojúhelníky jsou buďto stejného stupně dělení, nebo z o stupeň vyššího ($L+1$) – pravý a levý sousedící trojúhelník, nebo o stupeň nižšího ($L-1$) pro trojúhelníky sousedící základnou.



Obr. 19 – Popis vztahů pro binární trojúhelníkový strom

Jsou-li T a T_b stejné úrovně dělení, nazveme tento pár diamantem. Jak je z dosavadního popisu patrné, lze z libovolné úrovně získat libovolnou jinou úroveň používáním pouhých dvou operací – dělení a spojování.

Trojúhelník T nemůžeme rozdělit okamžitě, má-li jeho základnový soused T_b nižší stupeň dělení. V takovém případě musíme nejdřív trojúhelník T rozdělit tak, aby s trojúhelníkem T_b vytvořil diamant, viz následující obrázek.



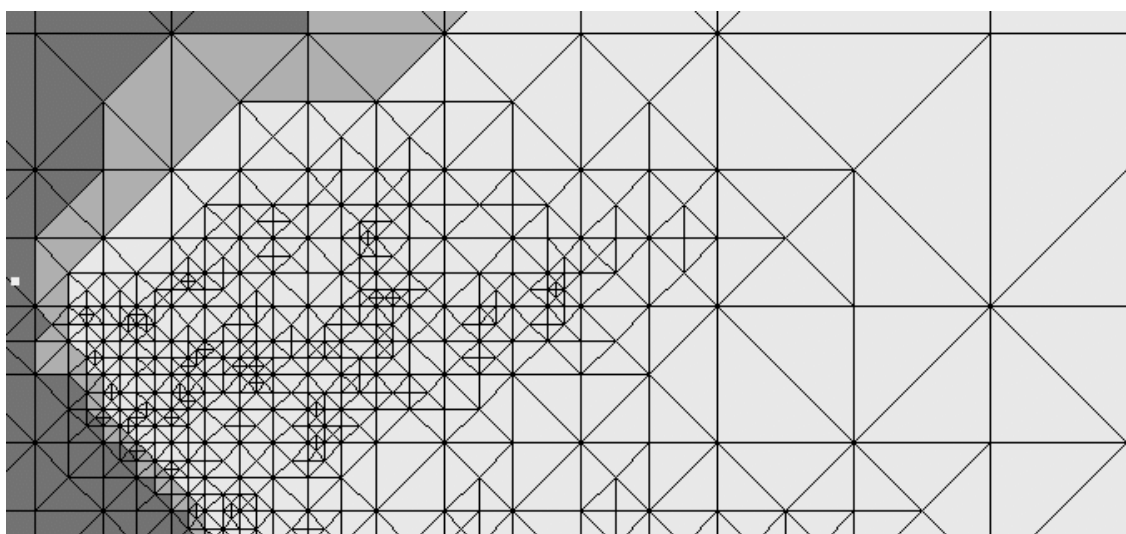
Obr. 20 – Vynucené dělení

Spojovací a dělicí operace poskytují dostatečný prostředek pro dosažení požadované aproximace terénu.

5.2.2 Postup vykreslování

Při tvorbě aproximace terénu vycházíme z výškové mapy a rekurzivně přidáváme potomky do trojúhelníkového binárního stromu, dokud není dosaženo požadované úrovně detailu. Po skončení této operace procházíme stromem v opačném směru a vykreslujeme jeho uzly.

Na následujícím obrázku vidíme ukázkový frame terénů generovaného algoritmem ROAM. Nejtmavší oblast je mimo view frustum, nejsvětlejší je viditelná a světle šedá se nachází na hranici viditelnosti.



Obr. 21 – Ukázkový frame algoritmu ROAM

II. PRAKTICKÁ ČÁST

6 NÁVRH SAMOSTATNÉHO PLUGINU PRO SYSTÉM SIMULÁTORU VRTULNÍKU

6.1 Dynamicky linkované knihovny – dll

Knihovna dll obsahuje zkompilované funkce a některé proměnné potřebné pro jejich běh. Hlavní program si poté za běhu podle potřeby připojí dll knihovnu a naimportuje funkce.

Dobrým příkladem je např. operační systém Windows.

Výhodou dll knihoven je, že jednak šetří operační paměť počítače, neboť se nahrávají pouze v případě potřeby a také to, že jednu dll knihovnu můžeme použít v různých programech.

Funkci exportujeme pomocí "předpony" extern "C" _declspec(dllexport). Příklad:

```
1 extern "C" _declspec(dllexport) int Add(int x, int y)
2 {
3     return x+y;
4 }
```

Při přeložení projekty je výsledkem dll knihovna a lib soubor, který obsahuje kolekci obj souborů.

6.2 Návrh rozhraní pro komunikaci mezi pluginem a jádrem simulátoru

Při návrhu rozhraní bylo nejprve nutné rozmyslet si, které údaje bude plugin pro svou činnost potřebovat. Tyto údaje můžeme rozdělit na statické a dynamické.

6.2.1 Statické údaje

Statické údaje jsou ty, které se za běhu simulátoru nebudou měnit. Jedná se např. o název výškové mapy, z které následně generujeme požadovaný terén, jedná se o textury terénu, nebo skyboxu, umístění objektů ve scéně (lesy, řeky, budovy).

6.2.2 Dynamické údaje

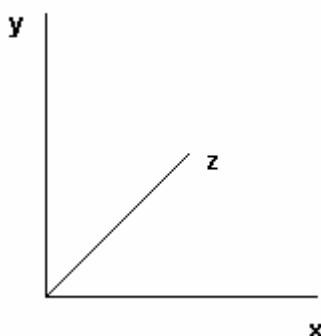
Dynamické údaje jsou ty údaje, u nichž za běhu simulátoru dochází ke změnám. Jde tedy hlavně o pozici vrtulníku a směr pohledu.

Tyto údaje jsou potřebné nejen pro umístění kamery, tzn. její pozici a vektor pohledu, ale i pro optimalizaci vykreslování dané scény metodami LOD nebo ořezového jehlanu.

Dále potřebujeme znát pozici pohyblivých objektů ve scéně, jakými jsou např. ostatní letadla a jiné pohyblivé cíle, nebo naším vrtulníkem vypálená raketa.

Pozice

Pozici udáváme na osách x, y a z, kde změna na ypsilonové ose označuje změnu výšky a osy x a z slouží pro pohyb po scéně.



Obr. 22 – Osy

V pluginu používám pro pohyb ve scéně pozici relativní.

Získání informací o aktuální poloze je realizováno tímto způsobem: hlavní třída CBScene, která zapouzdřuje plugin obsahuje pointer na objekt vrtulníku. Mezi funkce tohoto objektu patří funkce `GetModel()`, která vrátí ukazatel na něj samotný. Z jeho proměnných `m_TempGraphicX`, `m_TempGraphicY` a `m_TempGraphicZ` získáme informaci o poloze těžiště vrtulníku, které následně předáme jako parametry funkci pro vykreslení scény `appDraw(X, Y, Z);`

6.2.3 Detekce kolizí.

Je otázkou, jak řešit kolize s terénem. První možností je, že informaci o terénu a o výšce vzhledem k povrchu bude obsahovat pouze grafický plugin. Při testování této vzdálenosti

pak vrátíme informaci, zdali došlo ke kontaktu s povrchem a s ním spojené havárii, pokud je vzdálenost od povrchu rovna nule, nebo můžeme pokračovat v letu. Jádru simulátoru přijme informaci o kolizi a patřičně zareaguje. Eventuální poškození bude záviset na rychlosti a pozici vrtulníku vzhledem k místu kolize.

Druhou možností je ponechat testování na jádře simulátoru, který tedy bude obsahovat i informace o tvaru terénu a koordinátech pozice. Důvodem pro tuto možnost je potřeba testování např. zásahu zbraní nebo let autopilota.

6.3 Popis pluginu

6.3.1 Základ

Po dohodě s vedoucím práce jsem si jako základní stavební kámen pluginu zvolil [5], neboť již obsahoval implementaci algoritmu ROAM pro LOD.

6.3.2 Exportované funkce

Následující funkce jsou volány jádrem simulátoru.

Mezi exportované funkce jsem zařadil hlavně funkci `RenderScene()` která zajišťuje vykreslení scény a dále funkce pro vytvoření okna `CreateWindowSceneDK()`, zavření okna `CloseScene()` a změnu velikosti okna `ResizeScene()`.

6.3.3 Třída `CBScene`

Třída `CBScene` obsahuje proměnné renderovacího kontextu, kontextu zařízení, handle okna a instanci aplikace, příznak maximalizování a aktivního okna, dále ukazatel na objekt vrtulníku a základní funkce okna. Mezi ně patří funkce pro inicializaci renderovacího okna, funkce pro samotné vykreslení scény a funkce pro korektní uvolnění renderovacího kontextů a kontextu zařízení.

```
1 class CBScene
2 {
3     public:
4         HDC          m_hDC;
5         HGLRC        m_hRC;
6         HWND         m_hWnd;
```

```
7 HINSTANCE    m_hInstance;    int m_nWidth;
8 int m_nHeight;
9 float m_nClearColor[4];
10 BOOL m_bActive;
11 BOOL m_fFullScreen;
12 BYTE m_nProperties;
13 float m_nTime;
14 CModel *m_poModel;
15 void InitScene();
16 BOOL InitWindowScene();
17 void RenderScene();
18 void ClearScene();
19 };
```

6.3.4 Třída Terrain

Třída Terrain zapouzdřuje všechny proměnné týkající se samotného terénu, funkce pro načtení výškové mapy terénu, pro načtení textur a jeho generaci algoritmem ROAM. Dále zapouzdřuje strukturu trojúhelníkového binárního stromu, která vychází z [6] a [9]

```
1 struct TriNode
2 {
3     TriNode *left_child;
4     TriNode *right_child;
5     TriNode *base_neigh;
6     TriNode *left_neigh;
7     TriNode *right_neigh;
8     unsigned char flags;
9     float z;
10 };
```

Samotná třída je definována takto:

```
1 class Terrain
2 {
3 public:
4     int init(void);
5     bool LoadMap(const char *FileName);
6     ~Terrain();
7     void updateVariance();
8     int tessellate( float viewX, float viewY, float viewZ);
9     int render();
```

```
10 void setThreshold(float threshold);
11 float clipPoint(float x, float y);
12 friend void splitNode(TriNode *const target);
13 int m_tris;
14 int m_culls;
15 int m_verts;
16 public:
17 int m_size;
18 int m_log2size;
19 float m_hscale, m_vscale;
20 float m_viewX, m_viewY, m_viewZ;
21 float m_threshold;
22 float t_scale;
23 unsigned char *m_heightfield;
24 unsigned char *m_variances;
25 TriNode *m_nodePool;
26 int m_nextFreeNode;
27 TriNode *m_leftRoot;
28 TriNode *m_rightRoot;
29 void tessellateRecurse( TriNode *target, int level,
30                        int apexX, int apexY,
31                        int leftX, int leftY,
32                        int rightX, int rightY);
33 void renderRecurse(TriNode *target, int level,
34                  int apexX, int apexY,
35                  int leftX, int leftY,
36                  int rightX, int rightY);
37 TriNode *allocNode();
38 void clearNodes();
39 unsigned char updateVarianceRecurse(
40                        int apexX, int apexY,
41                        int leftX, int leftY,
42                        int rightX, int rightY);
43 void saveVariances();
44 };
```

6.3.5 Třída CFrustum

Třída Frustum obsahuje proměnné a funkce pro korektní optimalizaci scény metodou Frustum Culling, viz. kap. 3.1. Umožňuje testování polohy koule, bodu, přímky a polygonu vzhledem k ořezovému jehlanu.

```
1 class CFrustum
2 {
3 public:
4     bool TriangleInFrustum(float A[3], float B[3], float C[3]);
5     bool LineInFrustum(float A[3], float B[3]);
6     bool CubeInFrustum(float fPoint[3], float fSize);
7     bool SphereInFrustum(float fPoint[3], float fRadius);
8     bool PointInFrustum( float x, float y, float z );
9     bool PolygonInFrustum(BBOX *bbox);
10    bool Polygon2(vec3 point1, vec3 point2, vec3 point3);
11    void UpdateFrustum();
12    int CubeLocation(float fPoint[], float fSize);
13    int SphereLocation(float fPoint[3], float fRadius);
14    bool RightParallelepipedInFrustum(float fMin[3], float fMax[3]);
15    int cull_frustum( BBOX *bbox );
16    bool CubeFrustumD3D( float x, float y, float z, float size );
17    void CalculateFrustum(vec3 CameraPos, vec3 Cdir,
18 float Aspect, float Fov, float zNear, float zFar);
19    CFrustum();
20    virtual ~CFrustum();
21    float frustum[6][4];
22 };
```

6.3.6 Třída CTGA

Třída CTga obsahuje funkce a proměnné popisující formát po nahrání textury TGA. Jedná se o konstruktor, destruktor a funkci pro samotné nahrání textury.

6.4 Zobrazení jednoduchého částicového systému

Částicové systémy využíváme pro zobrazení takových efektů, jako je např. voda, oheň a kouř, úlomky po explozi, déšť, sníh apod.

Při návrhu částicového systému jsem vycházel z NeHe tutoriálu č. 19 [4].

6.4.1 Popis částice

Pro popis částice a jejího chování budou sloužit následující veličiny. Potřebujeme vědět, je-li částice stále aktivní, její životnost a rychlost stárnutí. Dále potřebujeme znát její barvu, rozloženou na složky RGB spektra. Musíme znát její souřadnice v 3D prostoru a rychlost pohybu na osách x, y a z. Na částici působí také gravitační síla Země, reprezentovaná ypsilonovou složkou. Vzhledem k tomu, že gravitační působení jiných objektů zanedbáváme, budou další složky gravitace nulové.

Výše zmíněné veličiny budou uloženy v následující struktuře.

```
1 typedef struct{
2   bool active;
3   float life;
4   float fade;
5   float r;
6   float g;
7   float b;
8   float x;
9   float y;
10  float z;
11  float xi;
12  float yi;
13  float zi;
14  float xgrav;
15  float ygrav;
16  float zgrav;
17 } particles;
```

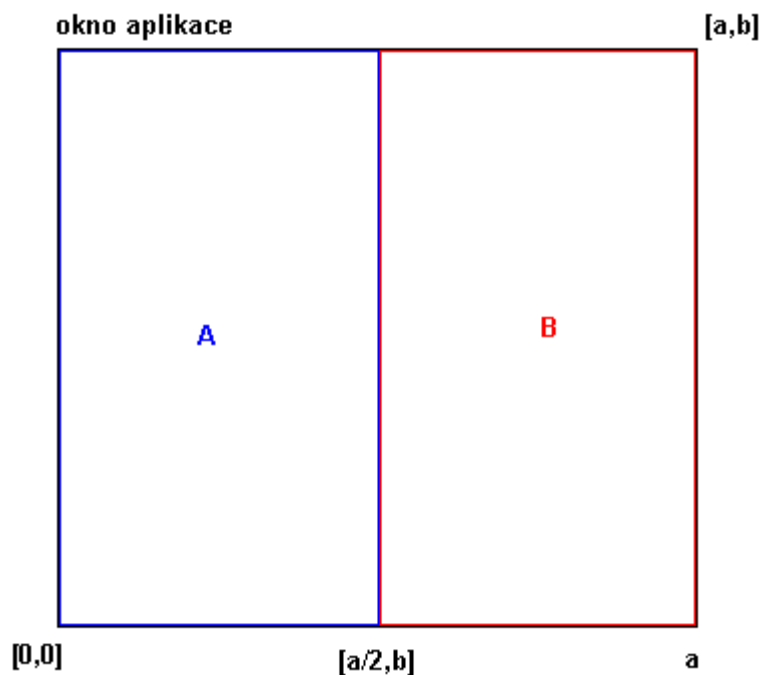
Tato struktura popisuje jednu částičku našeho systému. Pro simulaci efektu např. ohně jich budeme potřebovat několik set až tisíc. Nastavení proměnných provedeme při inicializaci částicového objektu.

6.4.2 Inicializace objektu

Při inicializaci částicového objektu nastavíme počáteční hodnoty a označíme částice za aktivní.

6.5 Zobrazení scény z pohledu pilota i třetí osoby.

Pro zobrazení více pohledů v okně aplikaci jsem použil funkci `glViewport`. Funkce `glViewport` má 4 parametry, kterými jsou pozice levého dolního rohu na ose x a na ose y a pozice pravého horního rohu na ose x a y . Představme si okno aplikace jako obdelník o stranách a, b . Na následujícím boráku je vidět rozdělení tohoto okna na dva viewporty.



Obr. 23 – Viewporty A a B

V případě na obr. 23. budou funkce vypadat takto:

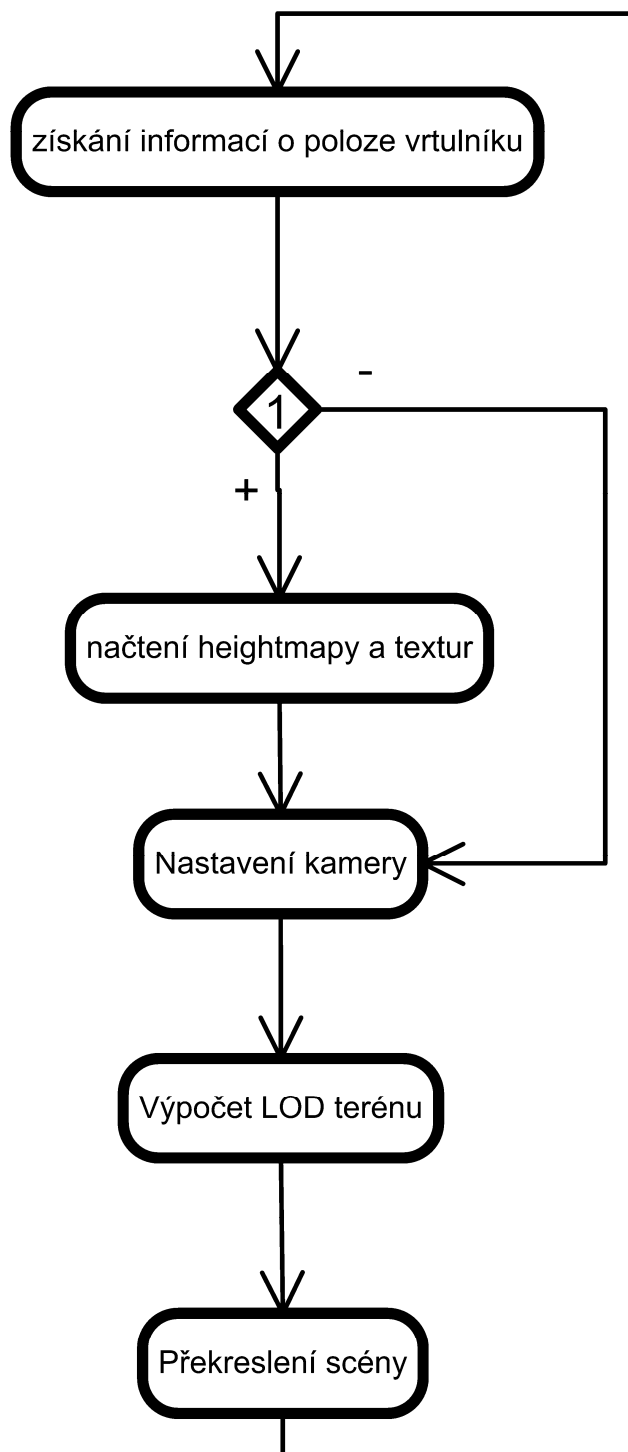
- 1) Pro viewport A: `glViewport(0,0,a/2,b);`
- 2) Pro viewport B: `glViewport(a/2,0,a,b);`

Protože nechceme, abychom měli dva identické pohledy, je nutné provést také odpovídající posun kamery funkcí `glTranslate` pro viewport B.

7 UML DOKUMENTACE

7.1 Běh aplikace

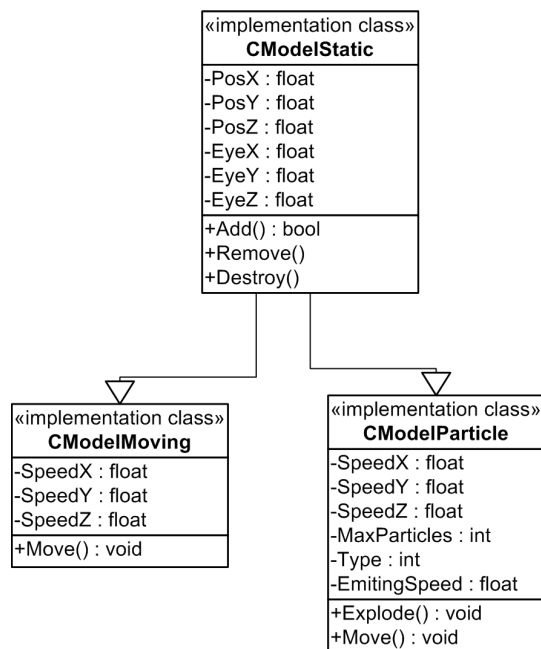
Následující UML diagram popisuje činnost pluginu.



Obr. 24 – Běh engine

7.2 Třídy modelů

Návrh tříd pro vkládání objektů do scény.



Obr. 25 – Návrh tříd pro objekty

8 ZÁVĚR

Tato práce měla za cíl vytvoření samostatného grafického pluginu pro projekt simulátoru vrtulníku využitím grafické knihovny OpenGL a jazyka C/C++.

Na začátku bylo nutné seznámit se s tvorbou dynamicky linkovaných knihoven systému Windows a s problematikou optimalizace vykreslování rozsáhlých scén a algoritmů pro generování terénů s proměnlivou úrovní detailů - LOD. Získané poznatky jsem následně uplatnil při tvorbě teoretické a hlavně praktické části.

Z algoritmů pro dynamický výpočet úrovně detailu jsem si vybral algoritmus ROAM – Realtime Optimally Adapting Meshes. Důvodů, které mě k tomu vedly, bylo několik. Tento algoritmus je středně náročný na pochopení, má přijatelné nároky na výkon procesoru i využití operační paměti a existuje k němu bohatá dokumentace včetně příkladů implementace v různých programovacích jazycích.

Dalším krokem tvorby pluginu byl návrh rozhraní mezi knihovnou a jádrem simulátoru vrtulníku. Bylo potřeba definovat způsob přístupu k datům obsaženým v hlavní aplikaci a volání funkcí definovaných v pluginu. Tato část mé práce vyžadovala úzkou spolupráci s ostatními řešiteli projektu.

Posledním krokem byla samotná realizace knihovny a vykreslovacích funkcí, která již mohla probíhat samostatně. I když jsem z časových důvodů nestihnul implementovat naprosto všechny požadované vlastnosti, poskytuje konečný stav grafického pluginu dostatečnou ilustraci zadaného tématu a může posloužit případným zájemcům jako základ pro pozdější rozšíření o nové funkce a vlastnosti.

SEZNAM POUŽITÉ LITERATURY

- [1] TIŠNOVSKÝ, Pavel. Modelování a vizualizace elastických těles pomocí systému vázaných částic [online]. Dostupný z WWW:
<http://www.elektrorevue.cz/clanky/03006/kap_1.htm.iso-8859-1>
- [2] SHOENBLUN, Daniel E. .md2 File Format Specification [online]. Dostupný z WWW: <<http://www.icculus.org/~phaethon/q3/formats/md2-schoenblum.html>>
- [3] HUMPHREY, Ben. Octree [online]. Dostupný z WWW:
<http://nehe.opengl.cz/cl_gl_octree.php>
- [4] MOLOFEE, Jeff. Particle Engine Using Triangle Strips [online]. Dostupný z WWW: <<http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=19>>
- [5] Proton engine [online]. Dostupný z WWW:
<<http://opengl.org.ru/proton/Land3.zip>>
- [6] TURNER, Bryan. Terrain Rendering with ROAM [online]. Dostupný z WWW:
<http://www.gamasutra.com/features/20000403/turner_01.htm>
- [7] MOLOFEE, Jeff, Particle Engine Using Triangle Strips [online]. Dostupný z WWW: < <http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=19> >
- [8] www.opengl.org [online]. Dostupný z WWW:
<<http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=19>>
- [9] DUCHAINEAU Mark, WOLINSKY Murray, SIGETI DavidE., MILLER MarkC., ALDRICH Charles, MINEEW-WEINSTEIN MarkB.: ROAMing Terrain: Real-time Optimally Adapting Meshes [online]. Dostupný z WWW:
<<http://www.llnl.gov/graphics/ROAM/roam.pdf>>
- [10] Terrain Tutorial [online]. Dostupný z WWW:
<<http://lighthouse3d.com/opengl/terrain/>>
- [11] SEMOTÁN, Vladimír. Vykreslování rozsáhlé krajiny v reálném čase [online]. Dostupný z WWW:
<<http://www.ceskehry.cz/magazin/view.php?cislocclanku=2004051901>>

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

LOD	Level of Detail
TIN	Triangle Irregular Network
ROAM	Real-Time Optimally Adapting Meshes
DLL	Dynamic Loadable Library
GIS	Geographic information system
Octree	Octal Tree

SEZNAM OBRÁZKŮ

Obr. 1 – Pohledový jehlan	13
Obr. 2 – Z-buffering.....	14
Obr. 3 – Octree.....	15
Obr. 4 – Generování terénu z heightmapy	17
Obr. 5 – Rozdělená scéna.....	18
Obr. 6 – Po 1. iteraci	19
Obr. 7 – Po 2. iteraci	19
Obr. 8 – Po 3. iteraci	20
Obr. 9 – Po 400. iteraci	20
Obr. 10 – Po 1000. iteraci	21
Obr. 11 – TIN scéna po rozřezání na parcely.....	23
Obr. 12 – Edge collaps.....	24
Obr. 13 – Triangle strip.....	24
Obr. 14 – Trhlina vzniklá na hranici parcel	25
Obr. 15 – Adaptace parcel	25
Obr. 16 – Užití triangle stripů	26
Obr. 17 – Dělení stromu	27
Obr. 18 – Dělení a slučování.....	28
Obr. 19 – Popis vztahů pro binární trojúhelníkový strom	28
Obr. 20 – Vynucené dělení	29
Obr. 21 – Ukázkový frame algoritmu ROAM	30
Obr. 22 – Osy	33
Obr. 23 – Viewporty A a B	39
Obr. 24 – Běh enginu	40
Obr. 25 – Návrh tříd pro objekty.....	41