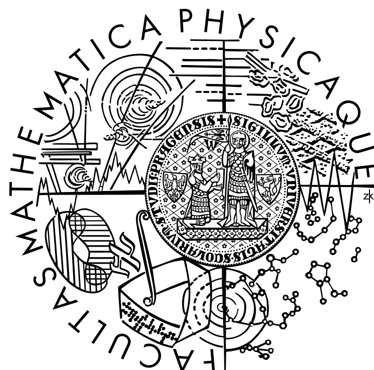
Univerzita Karlova v Praze

Matematicko-Fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Václav Klecanda

Modelování rostlin pomocí "Point-sprites"

KSI

Vedoucí bakalářské práce: RNDr. Josef Pelikán, KSVI

Studijní program: Informatika, programování

2006

Prohlašuji že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčování práce a jejím zveřejňováním.

V Praze dne: Jméno Příjmení:

Obsah :

KAPITOLA 1 MODELOVÁNÍ PŘÍRODNÍCH OBJEKTŮ....6

1.1 Modelování přírodních objektů (úvod)....6
1.2 LSystémy....6
1.3 Použití LSystémů při generování přírodních objektů....7
1.4 Generování geometrie stromu pomocí příkazů želvičce....8

KAPITOLA 2 PŘÍSTUPY POČÍTAČOVÉ GRAFIKY....8

2.1 LoD (Level of Details)....8
2.2 Point Sprites....9

KAPITOLA 3 MÁ IMPLEMENTACE GENEROVÁNÍ STROMŮ....10

3.1 Implementace LSystému....10
3.2 Struktura souboru LSystemu (.ls)....11
3.3 Implementace generování pomocí želvičky....12
3.4 Implementace LoD....13
3.5 Implementace struktur pro vykreslování....17
3.6 Obsah a Funkce jednotlivých C++ tříd....19

KAPITOLA 4 MOŽNÁ ZLEPŠENÍ....20

4.1 Možné přístupy jak zlepšit kvalitu zobrazení....20
4.2 Možné přístupy jak zefektivnit vykreslování....22

KAPITOLA 5 ZÁVĚR....23

SEZNAM ODBORNÉ LITERATURY....25

Název: Modelování rostlin pomocí "Point-sprites"

Autor: Václav Klecanda

Katedra (ústav): KSI

Vedoucí bakalařské práce: RNDr. Josef Pelikán, KSVI

E-mail vedoucího: Josef.Pelikan@mff.cuni.cz

Abstrakt: V předložené práci studujeme možnosti použití elementů harwarově akcelerované grafiky zvané Point Sprites v modelování a následném vykreslování přírodních objektů jako jsou stromy, keře, květiny. Dálším předmětem studia jsou možnosti spojení použití Point Sprites s některou z technik LoD(Level od Detail) pro dosažení největší efektivity zobrazování.

Klíčová slova: point sprites, LoD, LSystem, Gramatiky

Title: Modelování rostlin pomocí "Point-sprites"

Author: Václav Klecanda

Department: KSI

Supervisor: RNDr. Josef Pelikán, KSVI

Supervisor's e-mail address: Josef.Pelikan@mff.cuni.cz

Abstract: In the present work we study possibilities of using elemets of hardware accelerated computer graphics called Point Sprites in modeling and rendering of natural objects as trees, bushes, flowers. Next goal of the work is possibilities of connecting using Point Sprites with one of commonly used technique LoD(Level of Detail) for achieving best effectivity of rendering.

Keywords: point sprites, LoD, LSystem, Grammars

Kapitola 1

Modelování přírodních objektů

1.1 Modelování přírodních objektů (úvod)

Modelováním přírodních objektů nazýváme vytváření trojrozměrného virtuálního modelu, který se následně zobrazuje na obrazovce. Modelování tak rozmanitých struktur, jako jsou přírodní objekty, bylo po dlouhý čas velmi obtížné. Neexistoval totiž žádný formální popis procesu, kterým v přírodě objekty jako stromy, květiny, keře, vznikají. Ovšem odvodit takovýto popis vyžaduje nemalou biologickou zkušenost a znalost. Nakonec byl tento popis odvozen a vymyšleny metody jeho simulace. Je to vlastně část vědní disciplíny zvané umělý život. Artificial Life.

V následujícím textu používám slova strom jako pojmenování přírodního objektu. Přestože přírodním objektem není jen strom, ale i bezpočet jiných forem flory. Je to proto, že jsem měl omezný počet gramatik pro LSystémy (viz. níže) a z těch, které jsem k dispozici měl je výsledek nelépe vypadající podobný právě stromu. Pro konkrétní představu je možné a dobré si vzít například jabloň, která nejlépe odpovídá onomu nejlépe vypadajícímu výsledku. V některých pasážích textu je to malinko napbtíž, protože strom, tak jak ho budu v textu používat, se bude plést s pojmenováním datové struktury. Věřím ale, že z kontextu bude patrné čím myslím co.

Text je rozdělen do několika základních celků. Nejdříve je to obecný popis metod a přístupů, kterých má práce využívá. Potom následuje konkrétní popis mnou zvolených modifikací těchto postupů. Po nich následuje zamyšlení nad dalším možným vylepšováním. Závěr tvoří shrnutí výhod a nevýhod použitých postupů.

1.2 LSystémy

Pojem LSystém pochází od Aristid Lindenmayera, který úpravou formální gramatiky pro účely modelovaní vývoje jednoduchých mnohobuněčných organizmů, vytvořil teorii, která se později ukázala být vhodná i pro simulaci růstu a vývoje tak vysokých biologických forem jako jsou květiny, stromy nebo keře Tzv. *Lindenmayerovi systémy* (dále L-Systémy(LS)). Lsystém je tedy ve své podstatě gramatika. Gramatikou nazýváme čtveřici :

G(N, T, A, R), kde:

N je množina neterminálních symbolů,
T je množina terminálních symbolů
$A \in N$, je Axiom, což je počáteční řetězec symbolů z N
R je množina přepisovacích pravidel tvaru $u \to v$, kde $u,v \in (N \cup T)^*$, a u obsahuje

jeden neterminální symbol. Znaku u říkáme levá strana pravidla, řetězci v pak pravá strana.

Říkáme, že gramatika generuje nějaký jazyk. Jazyk je množina řetězců složených z terminálních symbolů. Generováním myslíme proces, kdy ze zadaného počátečního řetězce znaků(axiomu), pomocí přepisování dostáváme jiný řetězec. Přepisování není nic jiného než průchod zadaným axiomem znak po znaku a přepsáním každého neterminálního symbolu. Přepsání je nahrazení neterminálního symbolu $u \in N$ sekvencí jiných znaků definovaných pravidlem $p \in R$, které má na levé straně právě znak u. Říkáme, že u se přímo přepíše na v($u \rightarrow v$).Vznikne tak jiný řetězec, který je opět přepisován. Takto lze pokračovat až do doby, kdy se výsledný řetězec skládá pouze z terminálních symbolů. Toto ale LSystém modifikuje a místo toho se volí omezení počtu přepisování nějakou konstantou, nazývanou ***RecursionLevel***. Rozhodování , které z pravidel se použije je deterministické, proto množina pravidel R nesmí obsahovat více pravidel se stejným znakem na levé straně. LSystemu s takovýmto algoritmem výběru použitého pravidla se říká LS typu 0.

Toto je příklad přepisovacího procesu. Jednotlivé řádky odpovídají řetězci vzniklému i-tým přepisováním. Na prvním řádku je axiom.

- FFA - axiom
- A=F[&B]>(137)A - pravidlo pro přepis znaku A
- B=F[-C]C - pravidlo pro B
- C=F[+B]B - pravidlo pro C

0th:	FFA
1st:	FFF[&B]>(137)A
2nd:	FFF[&F[-C]C]>(137) F[&B]>(137)A
3rd:	FFF[&F[-F[+B]B]F[+B]B]>(137) F[&F[-C]C]>(137)F[&B]>(137)A
4th:	FFF[&F[-F[+F[-C]C] F[-C]C]F[+F[-C]C]F[-C]C]>(137)F[&F[-F[+B]B] F[+B]B]>(137)F[&F[-C]C]>(137)F[&B]>(137)A

1.3 Použití LSystémů při generování přírodních objektů

V předchozím odstavci jsem popsal, co je to LS typu 0. Tento LS používá deterministického rozhodování, které pravidlo použije. To ovšem znamená, že musí produkovat stále stejný řetězec znaků. Toto je bohužel pravda a to je jeho velká nevýhoda. Stromy vygenerované tímto druhem PS budou vypadat stále stejně. Tento neduh se snaží řešit tzv. stochastický LS. Ten se od typu 0 liší tím, že má více pravidel, které mají na pravé straně stejný symbol. Při přepisování se pak z těchto pravidel vybírá nedeterministicky, např na základě pravděpodobnosti, kterou každé pravidlo má u sebe uvedenou a s kterou se uplatňuje. Takovéto LS už produkují pokaždé jiný řetězec. Jejich jazyk je tedy rozmanitější. Jsou tedy pro generování přírodních objektů vhodnější.

To je ovšem dobrá vlastnost, ale pro účel mé práce nevhodná. Představme si, že vygenerujeme nějaký osamocený strom, ke kerému se uživatel přiblíží , zapamatuje si jeho tvar, a předpokládá, že když se na to místo vrátí, ten strom bude stejný. To mu stochastický LS tak, jak je definován, neumožní.To má ale jednoduché řešení. Stačí si zapamatovat pro každý strom ve scéně nějaký seed generátoru pseudo náhodných čísel, který v tomto případě simuluje onu náhodu.

1.4 Generování geometrie stromu pomocí příkazů želvičce

LS jako takový nám žádný strom nevygeneruje. Vygeneruje pouze dlouhý řetězec symbolů. Pro vlastní generování je třeba mít nástroj, který tyto symboly doukáže interpretovat jako pravidla pro růst nebo větvení a vygenerovat na jejich základě trojrozměrný model stromu.

Pro tento účel se vytvořila metoda zvaná „**kreslící želvička**“. Želvička je to jen pro představu. Je to jakási entita, která umí vykonávat příkazy, které jí jsou dávány. Má svůj aktuální stav, součástí kterého jsou atributy jako poloha, směr. Může jich být a je jich typicky více. Ale tyto 2 jsou základní. Příkazy pro želvičku jsou např.: otoč se o 30 stupňů okolo osy x, nebo pohni se o jednotku vpřed a nakresli tak čáru(segment). Příkazy tedy mění její stav. Takto definovaná by ještě ale žádný LS interpretovat nedokázala. Nebylo by totiž možné simulovat větvení stromu. Želvička se totiž neumí rozdvojit. Proto je potřeba mechanismus, který toto rozdojování nasimuluje. Tento se nazývá *Stavový Zásobník*. Je to klasický zásobník(stack), který uchovává stavy želvičky. Ta když narazí na rozvětvení(které musí být signalizováno speciálním symbolem), uloží svůj aktuální stav na zásobník a pokračuje ve zpracování větvící se větve. Když je s ní hotová, vezme si uložený stav zpět ze zásobníku a pokračuje dál ve zpracování původní větve.

Kapitola 2Přístupy počítačové grafiky

2.1 LoD (Level of Details)

LoD(Level of Details) „úroveň detailu“ je technika používaná v trojrozměrné počítačové grafice. Používá se všude tam, kde se scéna skládá z velkého množství objektů. Vykreslování takovéto scény je pak nesmírně časově náročné a není možné jej zaručit v reálném čase. Cílem této techniky je modifikovat objekty a tím pádem i celou scénu tak, aby se tato náročnost snížila. Postupem který se využívá je postupné zjednodušování objektů, které leží dále od kamery(oka pozorovatele). Protože jsou ve větší vzdálenosti, není nutné aby byly kresleny v plné kvalitě, protože oko pozorovatele je nevnímá s takovou intenzitou jako objekty na popředí.

Používájí se 2 přístupy k této technice. Jedním z nich je statický. Jeho základem je vytvoření několika verzí medelu objektu dopředu. Ve vykreslovací fázy se pak jen tyto verze vyměňují podle vzdálenosti od kamery. Pro zamezení náhlé změny jedné verze ve druhou(flicking) se mezi těmito dvěmi postupně prolíná.

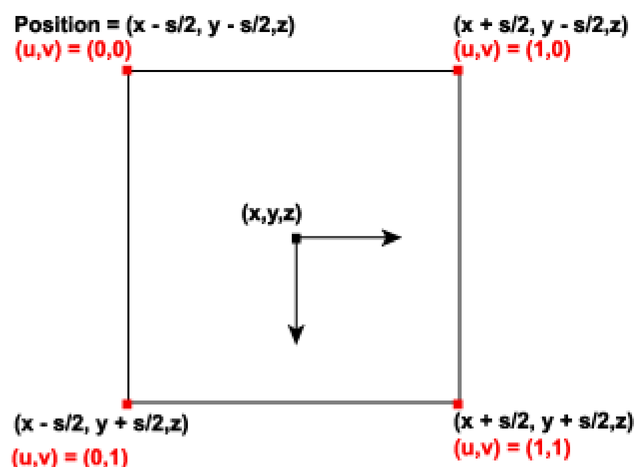
Druhým přístupem je dynamický. Tento namísto předpřipravených verzí postupně zjednodušuje určité části modelu za běhu. Dynamicky. Tento přístup je co do kvality lepší, avšak vyžaduje složité algoritmy na výběr části modelu pro zjednodušení. I na její následné zjednodušení. Významnou součástí takovéhoto LoD je rutina, která bude říkat kdy se se jednotlivé části modelu budou zjednodušovat. Jejím základem je nějaké rozdělení prostoru okolo kamery do soustředných mezikruží, které vlastně udávají prostory, různých úrovní detailu pro objekty, které se v nich nacházejí. Poloměry těchto mezikruží by se měli zvětšovat s nějakou rychleji rostoucí funkcí.

2.2 Point Sprites

Point Sprite(PS)je otexturovaný čtverec, který je vždy natočen k ose kamery. Je definován pouze svou pozicí. Což je nesmírná výhoda, která vede k nezanedbatelným úsporám hardwarových prostředků. Ať už je to úspora místa na grafiké kartě, nebo úspora množství dat na kartu posílaných. Protože je k jeho definici potřba jen poloha, stačí pouze jeden vektor, což jsou 3 floaty oproti 18ti floatům dvou trojůhelníků nebo 12ti jednoho čtverce. Protože je však ale neustále nasměrován kolmo na osu kamery, nemáme tolik možností, jak ho ovládat. Nemáme prakticky žádnou možnost. Kromě změny polohy, nebo textury s ním nic neuděláme. Proto mnoho lidí sdílí názor, že nejsou flexibilní. Avšak nachází uplatnění v tzv. Particle systems, které se používají na simulaci takových přírodních efektů jako je oheň, déšť a podobně.

Dalším atributem PS je velikost. Ta je buď fixní(čili neměnná se vzdáleností kamery od PS, takže může být chápána jako velikost v jednotkách obrazovky (screen-space)), nebo proměnlivá.V případě proměnlivé velikosti se počítá z uživatelem zadané funkce závislé na vzdálenosti od kamery(camera space) a počáteční velikosti.

Ze zadaného vektoru reprezentujícího pozici PS se na GPU dopočítají zbylé 4 body, které definují čtverec podle následujícího obrázku:



Kapitola 3

Má implementace generování stromů

3.1 Implementace LSystému

Implementován je stochastický LS, aby bylo dosaženo varialbility generovaných stromů, což bylo jedním z cílu zadání. Každé pravidlo obsahuje jednu nebo více variací. Pravidlo je tedy seznam variací se stejným neterminálem na levé straně. Každá variace je následujícího tvaru:

C(P)=R

Kde C je levá strana (neterminál), P je pravděpodobnost, nebo spíše distribuční funkce pravděpodobnosti mapována na interval 0 – 100. Tento údaj budu nazývat distribucí. A konečně R je pravá strana pravildla. Jinými slovy pro každé pravidlo C, existuje více variací, které se liší pravou stranou a distribucí.

Například pro pravidlo A obsahující 3 variace, které mají následující pravděpodobnosti (30%, 45% a 25%), musí být v souboru 3 záznamy, které mají následující levé strany:

A(30) =…,
A(75) =…,
A(100) =….

Jedinou výjimkou je axiom, který je zahrnut také mezi pravidla a který je tvaru: X(100) = . . . Takto je zobrazeno i v editačním boxu. Hodnota distribuční funkce(100) se nesmí měnit.

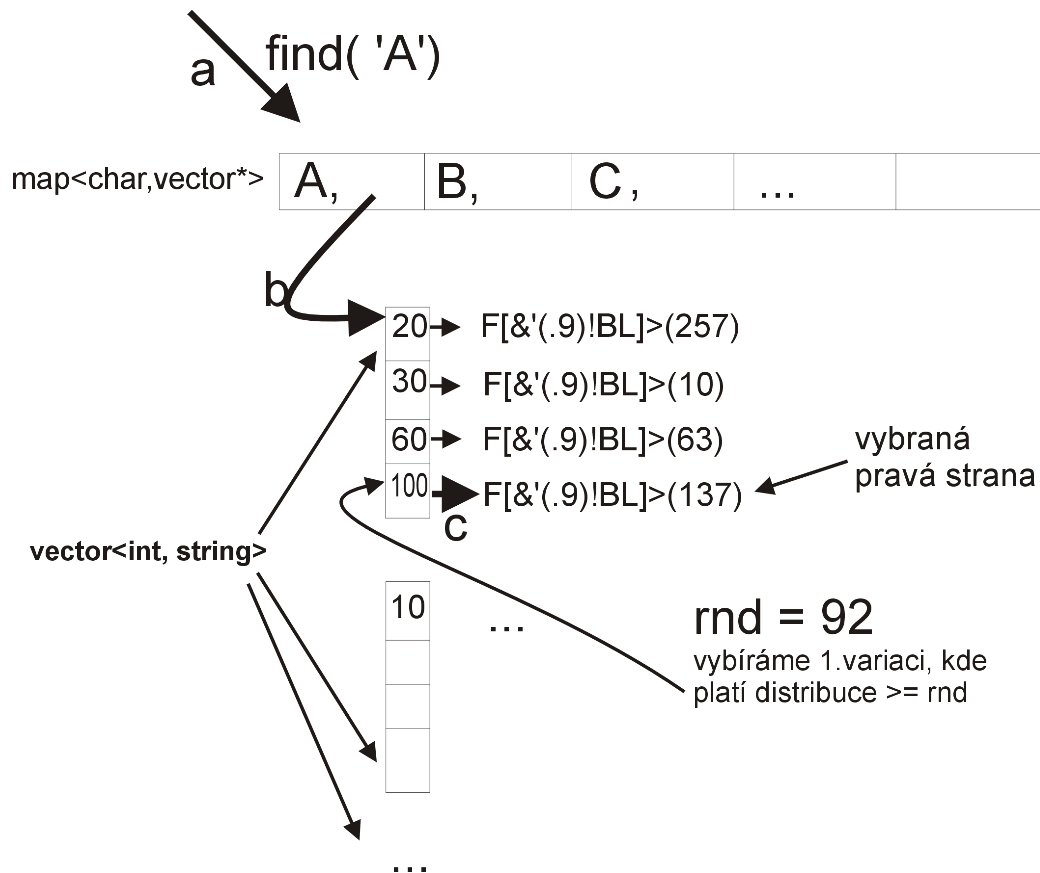
Implementace takovéhoto stochastického LS je pak založena na asociativním STL kontejneru map<char, variations>, který podle znaku char vyhledá příslušnou sadu variací variations. Tato sada je STL vector<short, string> dvojic celočíselné hodnoty distribuce a řetězce pravé strany variace.

Přepisovací proces se pak již neliší od teorie. Pro zadané n, n-krát zopakujeme přepisování dosavadně vzniklého řetězce. Přičemž na záčátku je zadaný počáteční řetězec, který chceme přepisovat. Ono jedno přepisování je iterace řetězcem znak po znaku a pro každý znak c vykonání následujících akcí:

- najít znak c v kontejneru pravidel. Když tam není, vystoupit jen c a zkončit, jinak pokračovat
- vygenerovat náhodné číslo rand z intervalu 0-100
- najít odpovídající variaci(první taková, která má distribuci >= rand
- vystoupit její pravou stranu

Na následujícím obrázku je znázorněno jak vypada struktura LS a příklad, jak probíhá výběr variace pravilda ‘A’.(šipky: a = vyhledání pravidla s A na levé straně v asociativním

kontaineru. b = přechod na odpovídající vektor variací, c = výběr pravidla na základě náhodné hodnoty rnd.)



3.2 Struktura souboru LSystemu (.ls)

Struktura souboru z LS(přípona .ls). Je následující:

- recursionLevels
- defaultAngle
- defaultThickness
- pravdla(mezi nimi i axiom)
- @

V souboru jsou 2 druhy elementů. Jednak jsou to funkční elementy, které jsou vlastními daty pro LSystem a pak jsou to komentáře, které ve funkci LS neuplatní. Struktura uvedená výše obsahuje pouze funkční elementy. Jsou to: recursionLevels(3 celočíselné hodnoty oddělené znakem '-'). Definují počet iterací přepisování pro každou úroveň *BranchTree*(viz. dále). DefaultThickness(default hodnota šířky větve v procentech), defaultAngle(default hodnota pro příkazy želvičce měnící orientaci), což jsou také celá čísla. Pravidla jsou řetězce tvaru popsaného výše. Každá variace pravidla je na samostatné řádce.

Variace musí být setříděny vzestupně podle distribuce. Seznam pravidel musí být zakončen znakem @ ležícím na začátku samostatné řádky z důvodu usnadnění rozpoznání konce seznamu pravidel. Každý element musí být na začátku samostatném řádku. Platí i pro komentáře. Avšak ty se mohou vyskýtnout i za funkčními elementy. Nikde v souboru však nesmí být mezery, kromě komentářů.

3.3 Implementace generování pomocí želvičky

Interpretaci řetězce, který vznikne přepisováním z LS zajišťuje objekt CursorStack. Ten má za úkol generování objektu. Jeho geometrie, topologie, *Nodes*, včetně tvorby a odesílání vertex a index bufferů. Pro generování obsahuje zásobník na stavy „želvičky“ (CursorStack). Tyto stavy jsou reprezentovány objektem Cursor. Objekt Cursor obsahuje metody, které vykonávají příkazy obsažené v řetězci, které mění atributy, které jsou také v Cursor a které odpovídají atributům stavu „želvičky“. CursorStack je tedy jakýsi řadič příkazů, které posílá Cursoru na vrcholu zásobníku stavů(aktuálnímu stavu „želvičky“). V případě, že narazí na příkaz „rozdvoj se“, přidá nový Cursor na vrchol, který dál akceptuje příkazy. Po přijetí příkazu „obnov stav“ jednoduše smaže tento nový Cursor a tím obnový původní stav i s jeho atributy.

Atributy jsou: poloha a orientace(reprezentovaná maticí). Následují atributy, jejichž počáteční hodnoty jsou definovány v LS: délka kreslených segmentů(float), defaultní hodnota pro příkazy změny orientace, které nemají parametr – defaultAngle(float), a šířka kreslených segmentů(float).

Příkazy se dají rozdělit do několika skupin podle toho, co znamenají. Jsou to Orientační(mění polohu a orientaci želvičky), Atributové(mění atributy) a Řídící(oznamují například větvení).

Následuje seznam implementovaných příkazů v přehledné tabulce rozdělěné podle skupin příkazů. V prvním sloupci je znak, který reprezentuje příkaz, druhém pak význam tohoto příkazu. U příkazů měnících směr je uvedena analogie s pohybem hlavy:

Tabulka 1, Seznam orientačních příkazů

f	Posune se o délku úseku v aktuálním směru, ale nekreslí
F	Posune se o délku úseku v aktuálním směru a nakreslí úsek
L	Zaznamenání nového listu
&	Otočení kolem osy X CW („sklopení hlavy“), *
^	Otočení kolem osy X CCW („zdvihnutí hlavy“), *
<	Otočení kolem osy Y CCW(„otočení hlavy“ vlevo), *
>	Otočení kolem osy Y CW(„otočení hlavy“ vpravo), *
+	Otočení kolem osy Z CW(„naklonění hlavy“ vpravo), *
-	Otočení kolem osy Z CCW(„naklonění hlavy“ vlevo), *
|	Otočení o 180° kolem osy Y(„otočení hlavy“ o 180°)
%	Otočení kolem osy Z o 180°
$	Otočení do roviny(„naklopení hlavy do roviny“ = úhel 0°)

Tabulka 2, Seznam řídících příkazů

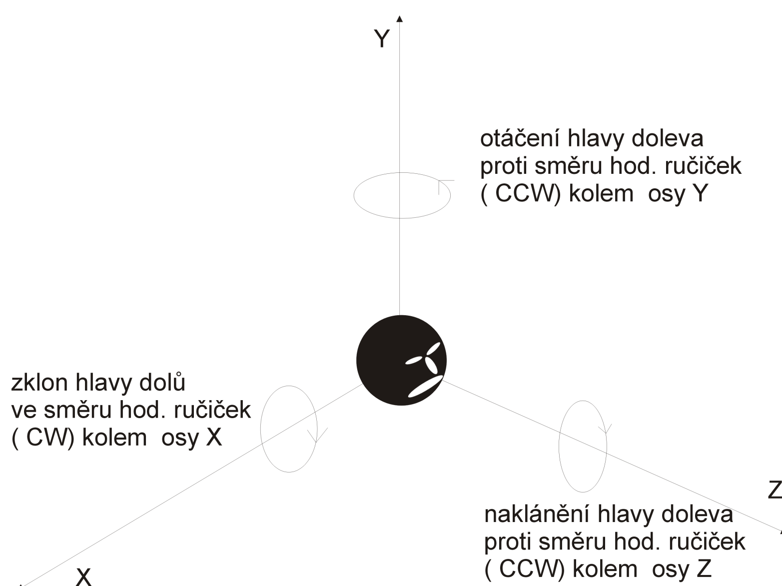
[	Uloží aktuální stav na zásobník přidáním nového Cursoru „rozdvoj se“
]	Smaže vrchol zásobníku a obnoví tak předchozí stav „obnov stav“

Tabulka 3, Seznam Atributových příkazů

"	Zvýšení délky úseku o 10%, *
'	Snížení délky úseku o 10%, *
;	Zvýšení defaultAngle o 10%, *
:	Snížení defaultAngle o 10%, *
?	Zvýšení defaultThickness o 30%, *
!	Snížení defaultThickness o 30%, *

Příkazy označené * je možné rozšířit o parametr uzavřením jeho hodnoty do závorek těsně za příkaz. Hodnota prametru je jakékoliv reálné číslo. Opět vše musí být bez mezer jako v souboru s LSystemem. Napr. +(67) = otočení kolem osy Z CV o úhel 67°.

Na obrázku je znázorněno pár příkazů měnících orientaci. (&, <, -) :



3.4 Implementace LoD

Základní myšlenka LoDu v mém podání je snaha redukovat komplikovanost modelu s rostoucí vzdáleností od kamery použitím jednodušších textur na *PointSprites* místo složité geometrie větví v plných detailech. Cílem je napodobit následující scénář: když pozorujete strom z blýzka, sledujete jeho listy, strukturu větví, jednotlivé detaily… ale jakmile se od něj

začnete vzdalovat, jednotlivé listy již přestanou být rozeznatelné a struktura větví se začíná čím dál tím více ztrácet a stávat nevýznamnou. Už i pozice jednotlivých listů ztrácí na významu a my je pozorujeme, že tam jsou, mají nějakou barvu ale jejich obrysy už nerozeznáme. Pro věrohodný a uvěřitelný pohled nám stačí pouze když to co pozorujeme má pořád stejnou barvu a prosvítá to(tak jako korunou stromu prosvítá světlo).Výnamným se nakonec stane tvar stromu.

Tento tvar determinuje několik prvních větví, které se oddělují od kmene. Je tedy determinován už na začátku svého vývoje. Tohoto se drží i můj systém.

Pro architekturu objektů, které reprezentují strom jsem zvolil hierarchickou reprezentaci kopírující vlastně způsob, jakým jsou stromy tvořeny v přírodě. Základem je kmen, ze kterého vyrůstají větve, z kterých další, menší větve, a tak dál až posledním prvkem jsou listy. Takový strom budu nazývat ***BranchTree*** („strom větví“). Tato reprezentace je pro potřeby takového systému, který dovoluje symulovat scénář popsaný výše, jako stvořená.

Stromová reprezentace stromu je výhodná hned z několika hledisek:

- Za prvé je identická s datovou strukturou strom, která vlastně vznikla abstrakcí stromu jak ho známe z přírody. V mém případě je to struktura nehomogenní. Jedním elementem je větev, druhým je list. Protože kmen není nic jiného než také větev(co do struktury) je strom tvořen stromem(datová struktura) větví, který má v listech(datové struktury) listy(jako části stromu, budou nazývány „listy“).
- Za druhé je to možnost definovat listy jako Point Sprites.
- Za třetí je to modifikovatelnost. Protože máme strom složen z jednotlivých větví, lze jednoduchými úpravami docílit efektů jako je například pohybování větví ve větru.

Strom je tvořen úrovněmi větví, což je pro LoD nejdůležitější vlastnost. Protože tyto úrovně nějak souvisí s pořadím postupného zjednodušování s rostoucí vzdáleností. Nejdříve se zjednodušuje nejspodnější úroveň. To jsou ty nejtenčí větve s listy, které při malé vdálenosti rozeznáváme a které jsou pro pozorovatele měřítkem kvality modelu. Ovšem s roustucí vzdáleností je pozorovatelovo oko přestane vnímat a soustředí se(dá se říci, že je zprůměruje) na jakési plochy. Následně se zjednoduší ta nad ní a takto se pokračuje až ke kořeni struktury. To ale přesně odpovídá našemu scénaři.

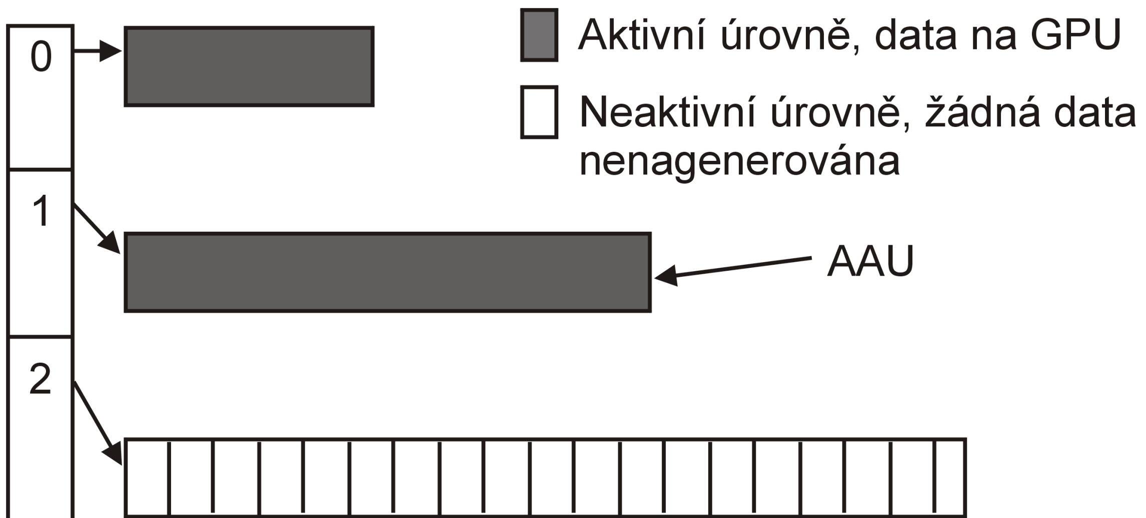
BranchTree by neměl být moc vysoký, protože jeho průchod by zabral moc času a zároveň jeho reprezentace v paměti by nebyla nejvýhodnější. Protože každá větev musí v sobě obsahovat informaci potřebnou pro její nakreslení a tato informace je v mém případě index buffer, není větší množsví žádoucí i z důvodu velkého množství těchto index bufferů, protože by to bylo spojeno s velkou režií na grafické kartě. Proto jsem zhora omezil jeho výšku na 3 (konstanta NUM_LVLS). Součástí modelu stromu jsou i listy, které by měli vlastně tvořit nejspodnější patro stromu větví. To by ovšem znamenalo dalsí režii pro průchodové algoritmy a i pro zprávu paměti, kterou by toto patro zabíralo. Proto jsem dalsí patro nepřidával a listy jsou součástí větví(netvoří speciální patro v *BranchTree*). Čili každá větev v sobě nese i své listy. Listy jsou reprezentovány jako body a později vykreslovány jako *PointSprites*.

Pro další potřeby je vhodné nazvat příslušné operace LoD zjednodušení ***Simplify***, kdy se z komplikované části modelu(větev s listy) vytvoří jednoduchá plocha. Tato plocha ale musí mít stejnou barvu(což je důležité), podobnou strukturu a prosvítá skrz ní světlo. Protichůdná operace je opětovná komplikace ***Enhance*** modelu. Kdy se jednoduchá plocha vymění za vygenerované komplikované listy a větve.

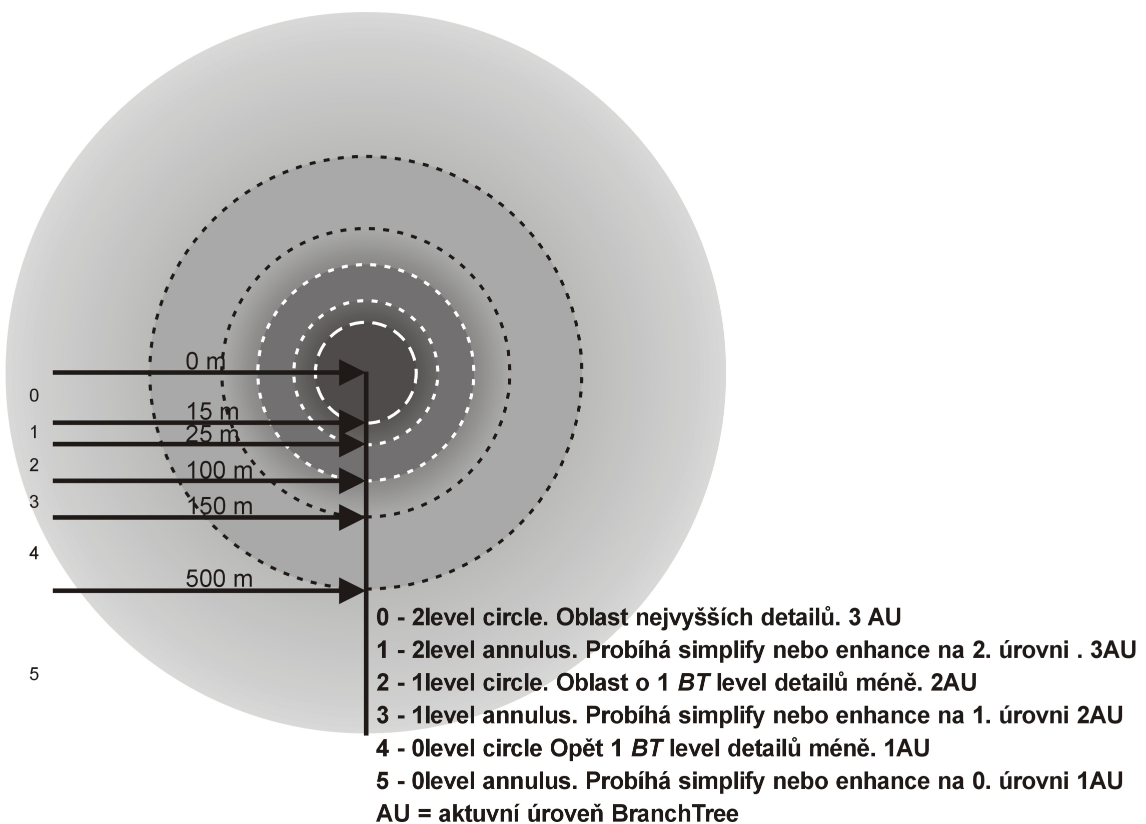
Protože můj LoD systém je úzce provázan s *PointSprites*(PS), operace *Simplify* převádí geometrii větviček a listů(opět PS) na plochu, která je ve formě 2 rozměrné textury a která se pak mapuje na PS. Slovo převádí je malinko zavádějící. Ve skutečnosti se vymění složitější geometrie za texturu, která se vybere(náhodně) z nějakého poolu textur. To ovšem porušuje požadavek na podobnost plochy(textury) s větví a listy(geometrie 3D) a může se stát (a taky stane), že si nebudou tvarově a strukturálně odpovídat. Otázkou je zda to v takové vzdálenosti, v jaké s *Simplify* dochází, vadí. Kdybychom toto chtěli eliminovat, museli bychom tuto texturu generovat. Například vykreslením do *OffScreen textury* . To ale přináší řadu svízelů, díky kterým se nevyplatí takovouto metodu implementovat. Je to například to, že, když pak provádíme operaci *Enhance* a pozice kamery se změnila od té, z jaké se provádělo *Simplify*, výsledný obraz se změní skokově, protože složitější model je jiný než zjednodušený. A to díky offscreen textuře, která se nemění s pozicí kamery.

Výběr větví, na které se provede Simplify se provádí sekvenčně a z aktuálně nejvyšší aktivní větve(aktuální aktivní úroveň, AAU). Takže je třeba pro ní mít seznam větví, které jsou už zjednodušené(SIMPLIFIED) a které ne(ENHANCED).Aktivní úrovní je uroveň, která má svá data aktualně na grafické kartě. Při potřebě provést jednu z operací se prostě vybere další prvek z odpovídajícího seznamu.

Aktivní úrovně přibývají postupně od kořene BranchTree a ubývají ke kořeni. Takže po vyprázdnění seznamu ENHANCED v AAU se provede operace *DecreaseActiveLevels*. Tato operace sníží počet aktivních úrovní stromu. Přesněji řečeno zruší data na kartě, pošle nové o naší, zatím ještě AAU, menší(operace *ReleaseLevel*), smazání úrovně o 1 výše nad AAU(operace DeleteLevel) a ustanovení nižší aktivní úrovně za AAU. Naopak při prázdném seznamu SIMPLIFIED (tj. AAU nemá žádnou větev zjednodušenou) a žádosti o další *Enhance* se přidává další aktivní úroveň, která je prohlášena za aktuální AAU. (operace *IncreaseActiveLevel*). Součástí této operace je generování dat(geometrie) úrovně o 1 výše než AAU při současném vygenerování další úrovně(zatím ještě neaktivní). Následně se zruší data na kartě a pošlou se nová, o přidanou úroveň větší. Nakonec se nová úroveň prohlásí za AAU. Jak je vidět, data se na kartu posílají po blocích o velikosti všech aktivních úrovní. Je to proto, že jsou sdílena jednotlivémi úrovněmi(viz. dále). Dále vidíme, že rozdíl mezi aktivní a neaktivní úrovní je to, že aktivní má vygenerovaná data(prošla operací *Generate*, která generuje geometrii, viz. Interpretace Želvičkou) a neaktivní nikov. Protože součástí *Generate* je i vytvoření nové, neaktivní, úrovně, můžou se neaktivní úrovně postupně mazat a zůstat může jen jedna(ta, co se bude potřebovat při další operace *IncreaseLevel*). Viz. obrázek. Výsledkem je tak méně dat pro vzdálenější stromy jak na kartě, tak i v systémové paměti. Což je záměr a úkol LoD.



Další částí, kterou popíši je rutina, která říká kdy se bude zjednodušovat. Její tělo tvoří několik podmínek. Protože se jedná o intervalové podmínky, nemůže být použit switch. Tyto intervaly jsou vlastně poloměry mezikruží(viz. obecná část o LoD). Počet těchto mezikruží odpovídá počtu úrovní BranchTree. Přidány jsou navíc mezikruží, ve kterých probíhá proces přechodu mezi jednotlivými úrovněmi detailu, čili mezi počtem aktivních úrovní *BranchTree*. Vše je vidět na obrázku:



3.5 Implementace struktur pro vykreslování

Základem struktur jsou objekty(C++) CNO(reprezentuje stom) a CBranch(reprezentuje větev). Protože jedním z hlavních cílů mé práce je rychlost vykreslování je tomu přiszpůsoben i *BranchTree* a důsledkem jsou určité optimalizace které vycházejí z následujích úvah, které v průběhu implementace zrály a měnili se. Uvedu některé z nich chronologicky uspořádané. Dostanu se tak ke strukturám tak jak jsou implementovány ve finální verzi:

- Z důvodu rychlosti není v každě větvy informace i listech jako taková, ale je tam pouze index buffer. Body reprezentující listy jsou ve vertex bufferu, který je společný pro celý strom a je také v objektu stromu(CNO). Index buffery ve větvých jsou tedy jen ukazatele do tohoto společného vertex bufferu. Stejně tak je tomu i u geometrie vlastních větví jako součístí kmene(***BODY***). Slovem *Body* budu nazývat vlastní kmen a větve jakožto útvary trojrozměrné pro jejichž kreslení jsou, kromě pozice, potřeba ještě normály a texturovací souřadnice. Zatímco *PointSprites* , jak budu nazývat listy, vyžadují jenom souřadnice. Geometrie *Body* i *PointSprites* je tvořena globálním vertex bufferem společným pro celý strom i přes to, že obě entity mají vertexy s rozdílnými atributy. U *PointSprites* se prostě nepoužily normaly a texturovací souřadnice. V každé větvy je pak jen index buffer s topologií. Toto je velice redundantní řešení. Které však bylo motivováno nezanedbatelnou myšlenkou, že grafické akcelerátory mají radi velké a hlavně celistvé objemy dat. Dále jsem ale použil pro Body a PointSprites rozdílné vertex buffery, aby k žádným redundancím nedocházelo a obětoval tak mensí množství index a vertex bufferů na kartě spojených s režií jejich udržování a zprávy za cenu menšího množství dat v nich.
- Další optimalizací je změna průchodu *BranchTree*. Předtavíme-li si postup, jakým by musel být vykreslován náš dosavadní model, dojdeme asi k následujícímu: Vezme se kořen stromu větví. Nastaví se globální vertex buffer pro *Body*. Nakreslí se aktualní větev (v tomto prípadě 1. úroveň *BranchTree*) za pomoci jejího index bufferu. Protože index buffer je pointer na nějakou strukturu ukazující na grafickou pamět, nemůžeme použít operátor sizeof na zjištění jeho velikosti potřebné pro vykreslování, musíme si pro každý tuto velikost někam poznamenat. Toto jsem naimplementoval dvojicí pair< vertx buffer, velikost>. Příslušné definice datových typů jsou komentované v kódu. Následně se přepne vertex buffer pro listy(dále *PSVertexBuffer*) a pomocí index bufferu pro listy v aktuální větvy se nakreslí listy aktuální větve. Následně se přejde na potomky aktuální větve(na další patro) ve stromě větví. Tento proces se pak opakuje pro každého z těchto potomků. Je to tedy klasický rekurzivní průchod stromem DFS. Tento způsob je ale značně nefektivní. Střídá se totiž kreslení *Body* a *PoitSprites*. Pokaždé se neustále přepínají vertex buffery. Cílem tedy je Nejdříve nakreslit *Body* a pak *PoitSprites*. Abychom nemuseli prochazet strom větví dvakrát, musíme mít nějaký seznam větví tak jak se vybírají při rekurzivním průchodu. Tento seznam si tedy vytvoříme při prvním průchodu stromem větví a pak už je stačí ho měnit jen v případě, že se strom větví nějak změní(např. při některé z operací systemu LoD). Tato strategie je použita pro kreslení jak *PoitSprites* tak i *Body*.
- Další optimalizací je rozdělení vykreslování listů do skupin se stejnou texturou. Počet textur(typů listů) na jednom stromě jsem omezil na 10(konstanta NUMBER_OF_PS_TEXTURES). Je to z toho důvodu, aby se minimalizoval počet přepínání mezi texturami při vykreslování. Dá se říci, že jde o věc podobnou přístupu zvanému material sorting. Toto je implementováno pomocí pole o velikosti

NUMBER_OF_PS_TEXTURES indexbufferů pro každou z větví. Znamená to tedy, že pro vykreslování listů není jeden seznam index bufferů (popsaný v předchozím odstavci), ale je jich pole o velikosti NUMBER_OF_PS_TEXTURES.

Takto jsem naimplementoval první optimalizovanou verzi. Ta sice fungovala dobře, ale pořád tam strašila ona velká redundance přítomností vertexů pro PS ve stejném vertex bufferu jako vertexy pro *Body*. To jsem vyřešil tak, že jsem tyto buffery rozdělil na buffer pro Body a druhý pro PointSprite i za cenu, že bude více bufferů, které musí GPU obhospodařovávat. Dále tato implementace obsahovala moc indexbufferů, které jsou také náročné na zprávu pro GPU. Na řadu tedy přišla další velká změna implementace. Ta spočívala v přenesení vertex i index bufferů na úroveň stromu, nikoliv větví. Ve větvích je pouze informace, kde se data větve nacházejí v bufferech ve stromě. Jsou to celočíselné offsety do vertex a index bufferů. Dále je třeba pro každou větev znát kolik je pro ni třeba nakreslit primitiv. Tyto tři informace jsem sdružil do struktur, které jsem nazval ***Node***. Jsou dva druhy. Jeden pro *Body* a druhý pro *PointSprites*, který nepotřebuje obsahovat offset do index bufferu, protože PS jsou body a tudíž se neindexují. Tato implementace už obsahuje pouze 3 buffery. 2 vertex a 1 index buffer pro celý strom.

Ale úpravy pokračovali. Tentokrát to bylo třeba protože jakmile máme buffery společné pro celý strom, nemáme možnost některou jejich část rušit a opět přidávat tak jak to vyžaduje LoD. Toto vyhazování a přidávání se děje po vrstvách *BranchTree*. Čili další verze rozdělila 1 sadu společných bufferů na tolik, kolik je úrovní *BranchTree* (konstanta NUM_LVLS). Tato verze ale fungovala značně pomaleji než ta se společnými buffery. Bylo tedy třeba zachovat myšlenku společných bufferů ale za cenu toho, že při každé snížení počtu aktivních úrovní *BranchTree* se budou muset buffery zrušit a vytvořit nové. Navíc to podmiňovalo ukládání vygenerovaných dat do objektu pro Strom, čili do systémové paměti. Toto předtím nebylo nutné, protože data mohli existovat jen na GPU. Toto je ale jedno, protože driver, když je buffer vytvořen s flagem POOL_DEFAULT, stejně vytváří kopie někde v lokální systémové paměti pro případ resetu zařízeni. Pak stačí vytvářet buffery pouze v paměti GPU a resetovací proceduru naprogramovat ručně a jsme na stejných pamětových nárocích jako předchozí verze.

Niní již máme fungující buffery, kterých není zbytečně moc, ve větvích jen nezbytně nutná data a systém připraven na LoD. Schází již jen systém, který by sdružoval věve, které se mají aktuálně vykreslit. Jinými slovy nejaký seznam větví, které se mají vykreslit ENHANCED(čili v plných detailech), jiných, které se mají vykreslit SIMPLYFIED(čili jen jako textury na *Point Spritech*). Pak seznamy větví, které obsahují jednotlivé typů listů, aby se mohli kreslit postupně všechny listu jednoho typu. To vše pro každou úroveň *BranchTree*. Toto vše by mělo být přístupně přes jednoduché rozhraní pro objekty výše v hierarchii lesa. Tím je například objekt reprezentující „druh“, který pak samotné vykreslování provádí a kreslí více podobných stromů najednou. Proto by měl mít možnost přistupovat jednoduše k jednotlivým částem stromů, které kreslí. Původně toto bylo řešeno pomocí STL kontejnerů, které jednotlivé větve sdružovaly. Toto ale nebylo příliš flexibilní řešení, protože jich bylo díky nutnosti mít pro každou kreslenou část a pro každou úrověň *BranchTree* neůnosně mnoho. Proto jsem od kontejnerů upustil a hledal řešení spíše v určitém prošívání *BranchTree*. Využil jsem toho, že *BranchTree* je strom složený ze stejných uzlů – větví. Pro každou větev jsem tedy definoval množinu poiterů, které toto prošívání budou realizovat. Tato množina je seskupena ve strukturu. Pro každou kreslenou část je v této struktuře jeden pointer. Takže *BranchTree* je díky těmto pointerům provázaný spojovým seznamy větví. Tyto seznamy jsem podle nazval ***Chains***. Podle funkce kterou plní jsou to *DrawingChain*, spojový seznam větví, u kterých se kreslí *Body*, *PSChains*, seznamy sdružující jednotlivé větve které obsahují ten který typ listu. Těch je NUMBER_OF_PS_TEXTURES a jsou reprezentovány

polem ukazatelů, kde i-tý prvek pole odpovídá ukazateli na další větev obsahující i-tý typ listu. Struktura vypadá asi takto:

Typedef struct sChainPointers_{

CBranch *NextOnLevel;   // pointer na souseda na stejné úrovni v *BranchTree*

CBranch *NextInBodyDraw; // pointer na další větev v řetězci vykreslovaných větví

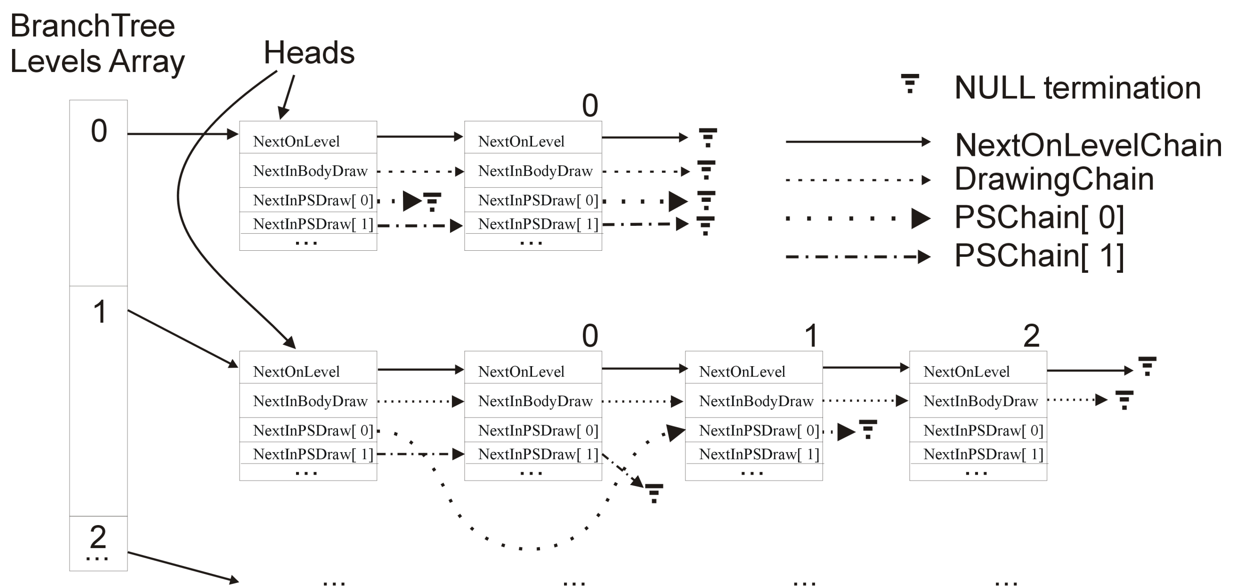
CBranch *Next Simplified;   // pointer na další zjednodušenou

CBranch *NextInPSDraw[NUMBER_OF_PS_TEXTURES]; // pole pointerů na jednotlivé typy listů

}TChainPointers;

Tyto seznamy jsou tvořeny při generování. Pro každou úroveň *BranchTree* je jedena sada *Chains*, uvozená hlavou(také objekt CBranch).

Nejlépe celou situaci osvětlí následující obrázek. Jsou na něm znázorněny jednotlivé úrovně *BranchTree*. A 4 druhy chains. NextOnLevel, Drawing a 2PS Chains. NextOnLevel Spojuje větve na stejné úrovni. *DrawingChain* spojuje ty co se budou kreslit ENHANCED. Stejně tak PS[0]Chain. Avšak ten vynechává větev 0 na úrovni 1. To je tím, že v ní nejsou žádné listy typu 0. Navíc je ukončen před koncem úrovně, takže větev 2 je také nemá. Analogicky je to s PS[1]Chain.



3.6 Obsah a Funkce jednotlivých C++ tříd

Třída CNO

Popis:

Reprezentuje model stromu.

Členské proměnné:

m_toWorld = matice reprezentující pozici a orientaci stromu ve světě

m_randSeed = inicializátor generátoru náhodných čísel(viz. generování)

m_branchRoot = kořen stromu větví

m_pointSprites, m_bodyVertBuf = vertex buffery pro listy a BODY společné pro všechny větve

m_pointSpritesToDraw = pole containerů index bufferů pro optimalizaci procházení stromem větví při vykreslování listů(popsáno výše)

Metody:

Draw = nakreslí strom

Třída CBranch

Popis:

Reprezentuje větev. Objekty této třídy tvoří strom větví

Kapitola 4

Možná zlepšení

4.1 Možné přístupy jak zlepšit kvalitu zobrazení

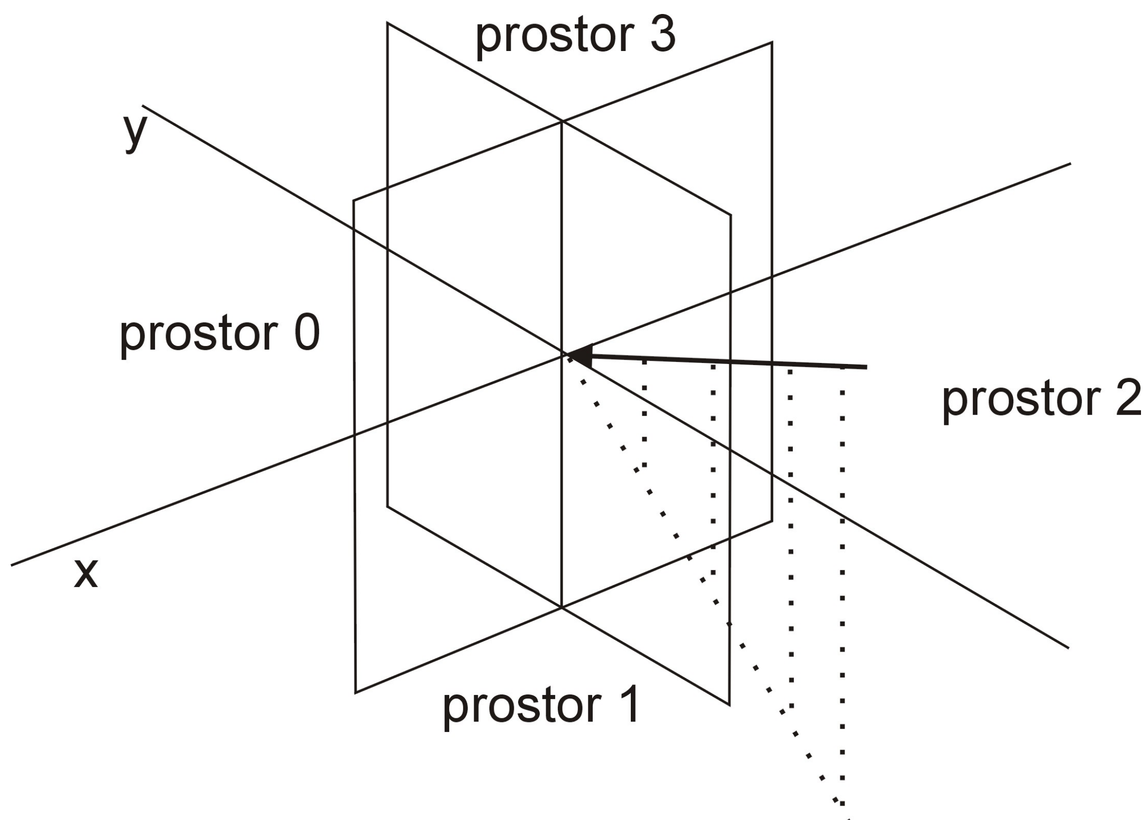
Zamyslíme-li se nad nedostatky našeho dosavadního systému, dojdeme k jedné zásadní věci, která ruší realistický vjem. Je to ono porušení toho předpokladu o podobnosti struktury a obrysu při operaci *Simplify*. Jak již bylo naznačeno, toto by se dalo eliminovat tak, že by se místo zjednodušované větve nepoužila předem připravená textura z nejakého poolu textur, ale tato by se dinamicky vygenerovala vykreslením této větve do OffScreen textury. Textura by tedy přesně odpovídala zjednodušované větvi. To ovšem ale platí pouze z toho směru, z kterého se dívala kamera v době provádění *Simlify*. Pakliže se kamera pohubuje, textura se již nemění a podmínka podobnosti struktury a obrysu je opět porušena. To nám ale již nevadí, protože se již nic nemění a nebliká.

To, že se textura již nemění a tudíž vlastně nebere v potaz úhel, z jakého je pozorována(je to vlastně billboard) může vypadat dost divně při procházení okolo stromu kolem dokola. To může být ovšem napraveno opět tím, že když se pozice kamery změní o nějakou konstatntu, OffScreen textura se vygeneruje znovu z aktuálního pohledu. Tato metoda ale předpokládá to, že je z čeho generovat. Tzn. musíme mít ještě k dispozici data celé větve a to pocelou dobu života *Simplified* textury. Uvážíme, že máme-li více úrovní

BranchTree a zjednodušujeme po úrovních, musíme tedy držet v pamětech(jak Systemové, tak i GPU) celý strom a LoD zaměrit tedy jen na zjednodušení vykreslování. Nikoliv na snížení hardwarových nároků(paměť). Spočítáme-li si že strom, který v plně kvalitě má 500kB jen v paměti GPU a kreslíme zcénu, která obsahuje 1000 stromů, zabere nám 0.5GB paměti GPU. Což je neúnosná hodnota. Dalším nezanedbatelným objemem dat na GPU jsou právě ony vygenerované textury. Protože pro každou větev máme jednu, je to další neůnosný objem dat.

Jisté nedostatky v zobrazování ale přetrvaly. Například, to, že když regenerujeme textury z každým větším pohybem kamery, tak se nám opět skokově změní, což je pro oko uživatelovo dost rušivé. Nastává to také když provádíme opět operaci *Enhance*. Dá se to opět řešit prolínáním starého a nového obrazu. To ale zase vyžaduje existenci obou obrazů současně, což je zase další velká porce hardwarových prostředků. Je to tedy metoda, která čeká na lepší hardware, nebo na detailnější rozpracování, aby mohla být implementována.

Jistým vylepšením této metody může být implementace následující úvahy. Máme-li zjednodušovat, dělejme to nejdříve vzadu(z aktuálního pohledu) stromu, kde to nemá tak rušivý dopad jako na obrysech a vpředu. Když už nemáme žádné větve pro *Simplify* vzadu, používáme ty vpředu a až nakonec na obrysech. To vyžaduje použití něčeho jako adresáře prostoru. Prostor si rozdělíme na několik segmentů(např. 4, rozříznutím krychle okolo stromu dvěma řezy ve svislém směru). Pak již při generování modelu musíme znát, které větve jsou v kterém segmentu a dát jim tuto informaci jako atribut a zařadit je do nějaké struktury podle tohoto atributu. Pak když vybíráme větve pro operaci *Simplify*, vezmeme vektor pohledu od kamery, tímto naadresujeme náš adresář a tak víme, které větve jsou vzadu, vpředu, vlevo a vpravo od kamery.



Dalším vylepšením realističnosti je simulace třepotání listů ve větru. Díky statičnosti PS toto není možné realizovat natáčením PS v prostoru. Lze to ale simulovat střídáním dvou

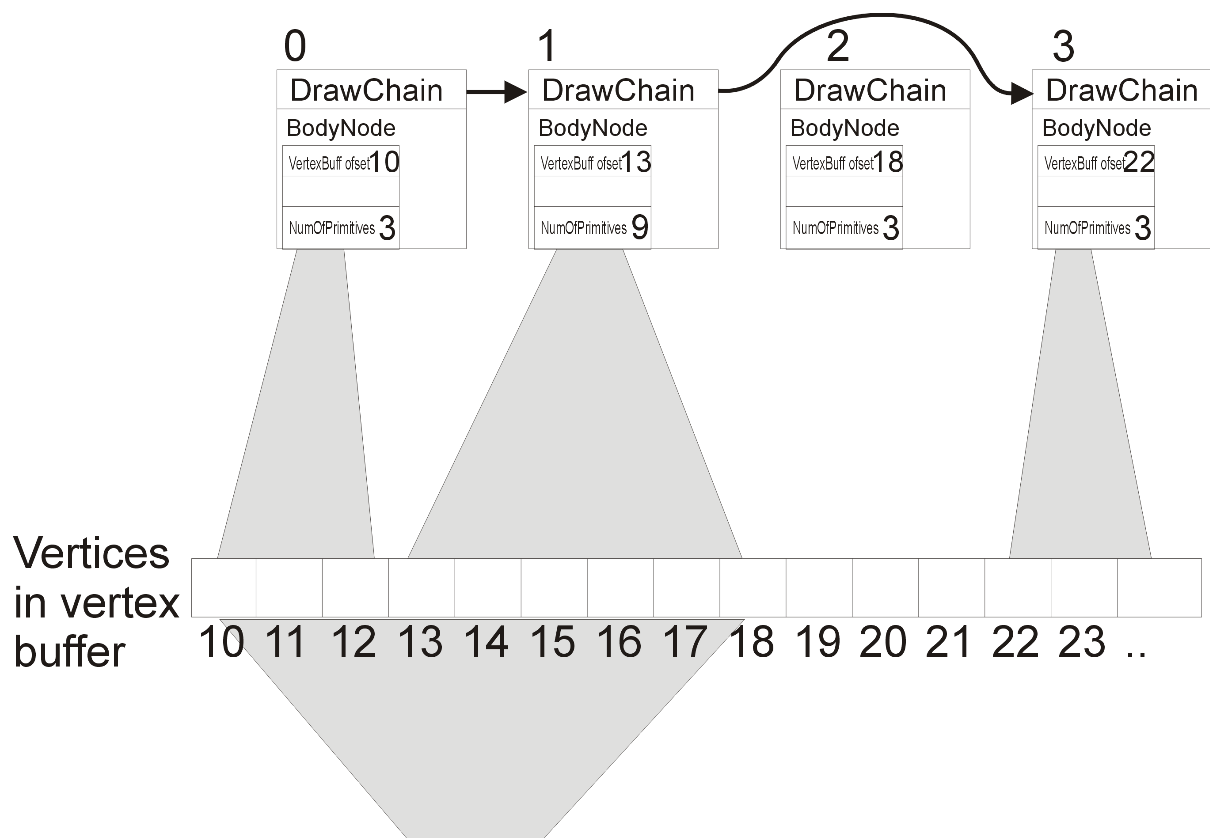
či více textur v čase. Toto nepřinese žádné zpomalení díky vysokým nárokům na hardware, protože stačí modifikovat vlastnosti pouze na úrovni *DrawChains*, kterých je pouze 10(NUM_OF_PS_TEXTURES).

4.2 Možné přístupy jak zefektivnit vykreslování

Během implementace jsem vymyslel několik způsobů jak vykreslování ještě urychlit. Pro nastínění možných způsobů je třeba si uvědomit, kde jsou slabiny našeho systému. Pak je možné se zamýšlet na tím, jak je odstranit, nebo alespoň zmírnit jejich dopad.

Prvním neduhem je časté střídání textur. V případě, že máme více stromů stejného druhu v lese, a kreslíme každý zvlášť vystřídáme každou z textur pro listy tolikrát, kolik máme v lese stromů tohoto druhu. Zlepšením je nějak tyto stromy sdružit a vykreslovat každý druh textury najednou. Toto združení je možné realizovat zapousdtřením všech stromů jednoho druhu do jednoho objektu, který je bude obsahovat (CKind – „druh“). Dále bude obsahovat použité textury, protože stromy stejného druhu používají i stejné textury. Les pak bude vypadat jako nějaký kontainer těchto sdružujících objektů. Nejlepší je použít Stl:Map, aby bylo možné rychle vyhledat, případně rozhodnout, zda druh přidávaného stromu už je ve scéně a tudíž má svůj sdružující objekt, nebo je třeba založit nový. Vykreslování je potom sekvenční průchod přes tento kontainer. Pro každý „druh“ v tomto kontaineru se vykreslí všechny části stromů, které obsahuje již najednou bez zbytečného přepínání textur(Např. nejdříve *Body*, potom PS1, PS2, …, kde PSi je i-tý druh PS).

Dalším je mnohonásobné volání vykreslovací metody DrawPrimitive objektu reprezentujícího device DirectX, které tento vykreslující příkaz zařadí do fronty někde uvnitř driveru a když tato je plná, všechny příkazy se odešlou na GPU. Vše je ale doprovázeno přechodem z uzivatelského režimu fo režimu jádra a zpět. Tyto přechody trvají ale nezanedbatelně dlouho. Toto lze eliminovat tím, že sejednotlivá volání seskupí do jednoho. Protože v *DrawChain* máme informace jako offest do vertex buffer, počet vykreslovaných primitiv mužeme volání pro vykreslování zhustit do jednoho. Podmínkou je, aby vykreslovaná primitiva ležela ihned za sebou ve vertexbufferu. Tuto podmínku kazí to, že někde v tom kterém *DrawChain* jsou mezery způsobené tím, že některá větev již prošla operací *Simplify*. Ty primitiva, které jí reprezentují se při vykreslování již neuplatní tudíž nám porušuje kontinuitu dat pro spojení vykreslovacích volání. Následuje obrázek. Větev 2 již prošla *Simplify* a zkazí nám zpojení vykreslení všch 4 větví do 1 volání. Ale 0 a 1 spojit můžeme.



Spojit kreslení větve 0 a 2 v jedno volání
DrawPrimitive(start = 10, count = 9)

Každopádně tímto způsobem vykreslování urychlíme. V nejhorším případě si nepomůžeme. Tento případ nastane, střídají-li se v DrawChain větve, které jsou ENHANCED a SIMPLYFIED. Což v mé implementaci není, protože větve pro *Simplify* se vybírají postupně. Takže se dá očekávat, že bychom si tímto vylepšením pomohli.Tyto úvahy je možné implementovat pouze, pokud nechceme mít pro každou větev její vlastní modelovou matici nutnou např. pro simulaci ohýbání větví větrem. Pak pak je nutné již při fázi generování generovat vertexy s pozicí vzhledem k celému stromu. Pak je možné tímto způsobem vykreslování optimalizovat. V případě, že pro každou větev máme její vlastní modelovou matici, vertexi jsou generovány s pozicí vzhledem k bázy této větve a při vykreslování je nutné tuto matici před každým voláním poslat na GPU. Potom není možné tyto ůvahy implementovat.

Kapitola 5

Závěr

Na závěr se sluší udělat jisté zhodnocení. Budu tedy hodnotit cíle mé práce, kterými jsou "možnosti použití Point Sprites ve vykreslování přírodních objektů a jejich zapojení do LoD". Podle mého soudu je použití Point Sprites dobrý nápad, pokud netrváme na vysoké

realističnosti modelů. Výhodný je nepochybně už z hlediska úspory hardwarových prostředků. Tato úspora je opravdu velká. Má ale i své nevýhody. Největší z nich je ona neflexibilita. To, že se nedají natáčet v prostoru a tím simulovat různorodé natočení listů ve stromě. Listy tak vypadají všechny stejně. Toto řeší použití více druhú listů(jinak natočených, barevných), ale různorodost, tak jak ji známe z přírody nenasimulujeme. Další nevýhodou je nemožnost aplikovat světlo na PS. Při jejich renderingu je třeba osvětlování vypnout. Není to možné už z toho důvodu, že nemají normálové vektory a jsou neustále nasměrovány na kameru. Není tak možné simulovat na stromech efekty, které vytváří např. slunce při své pouti oblohou.

Navíc se dájí PS pohodlně kombinovat s LoD. Prostým vyměněním za texturu, která reprezentuje větší část modelu. Tyto části se dají hierarchicky uspořádávat a dovolit tak velkou modifikovatelnost a flexibilitu.

Seznam odborné literatury

[1]. Przemyslaw Prusinkiewicz, Aristid Lindenmayer: The Algorithmic Beauty of Plants, Springer, 1990

[2]. DirectX documentation, MSDN

[3]. www.two-kings.de. DirectX tutorials