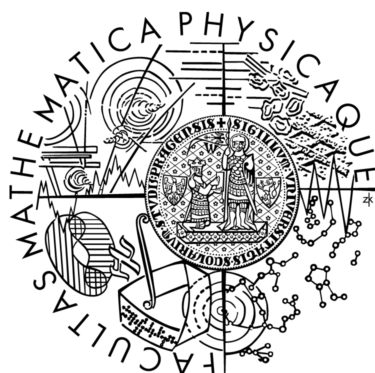
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Václav Klecanda

Modelování rostlin pomocí „Point-sprites“

KSVI

Vedoucí bakalářské práce: RNDr. Josef Pelikán, KSVI

Studijní program: Informatika, programování

2006

Rád bych poděkoval vedoucímu mé práce panu doktoru Josefu Pelikánovi za skvělé vedení a výborné podněty směrující mou práci ke zdárnému cíli. Další velký dík patří panu magistru Bohuslavu Horovi, který provedl korekturu a kontrolu typografické správnosti.

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 a příjmení:

Obsah :

KAPITOLA 1 MODELOVÁNÍ PŘÍRODNÍCH OBJEKTŮ 5

1.1 MODELOVÁNÍ PŘÍRODNÍCH OBJEKTŮ (ÚVOD) 5
1.2 LSYSTÉMY 5
1.3 POUŽITÍ LSYSTÉMŮ PŘI GENEROVÁNÍ PŘÍRODNÍCH OBJEKTŮ 6
1.4 GENEROVÁNÍ GEOMETRIE STROMU POMOCÍ PŘÍKAZŮ „ŽELVIČCE“ 7

KAPITOLA 2 PŘÍSTUPY POČÍTAČOVÉ GRAFIKY 8

2.1 LOD (LEVEL OF DETAILS) 8
2.2 POINT SPRITES 8

KAPITOLA 3 MOJE IMPLEMENTACE GENEROVÁNÍ STROMŮ 10

3.1 IMPLEMENTACE LSYSTÉMU 10
3.2 STRUKTURA SOUBORU LSYSTÉMU (.LS) 11
3.3 IMPLEMENTACE GENEROVÁNÍ POMOCÍ ŽELVIČKY 12
3.4 IMPLEMENTACE LOD 14
3.5 IMPLEMENTACE STRUKTUR PRO VYKRESLOVÁNÍ... 17
3.6 OBSAH A FUNKCE JEDNOTLIVÝCH C++ TŘÍD 20

KAPITOLA 4 MOŽNÁ ZLEPŠENÍ.. 26

4.1 MOŽNÉ PŘÍSTUPY JAK ZLEPŠIT KVALITU ZOBRAZENÍ... 26
4.2 MOŽNÉ PŘÍSTUPY ZEFEKTIVNĚNÍ VYKRESLOVÁNÍ 27

KAPITOLA 5 ZÁVĚR.. 30

KAPITOLA 6 SEZNAM ODBORNÉ LITERATURY 32

KAPITOLA 7 SCREENSHOTS 33

Název: Modelování rostlin pomocí „Point-sprites“

Autor: Václav Klecanda

Katedra (ústav): KSVI

Vedoucí bakalářské práce: RNDr. Josef Pelikán, KSVI

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

Abstrakt: V předložené práci studuji možnosti použití elementů hardwarově akcelerované grafiky zvané Point Sprites v modelování a následném vykreslování přírodních objektů, jako jsou stromy, keře, květiny. Dalším 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: Modelling natural models using „Point-sprites“

Author: Václav Klecanda

Department: KSVI

Supervisor: RNDr. Josef Pelikán, KSVI

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

Abstract: In the present work i 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 possibility 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 byly 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 omezený počet gramatik pro LSystémy (viz níže), a z těch, které jsem k dispozici měl, je nejlepší výsledek podobný právě stromu. Abychom získali konkrétní představu, je dobré si vzít například jabloň, která nejlépe odpovídá onomu nejlepšímu výsledku. V některých pasážích textu je to malinko na obtíž, protože strom, tak jak ho budu v textu používat, se bude plést s pojmenováním datové struktury. Věřím ale, že z kontextu budou patrné rozdíly v pojmech.

Text je rozdělen do několika základních celků. Obsah textu je rozdělen od obecného ke konkrétnímu. 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 (1925 – 1989), který úpravou formální gramatiky pro účely modelovaní vývoje jednoduchých mnohobuněčných organizmů vytvořil teorii, a ta se později ukázala být vhodná i pro simulaci růstu a vývoje tak vysokých biologických forem, jako jsou květiny, stromy nebo keře, tzv. *Lindenmayerovy systémy* (dále LSystém (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 (axiom) $\in$ N, což je počáteční řetězec symbolů z N;

R je množina přepisovacích pravidel tvaru $u \rightarrow v$, kde $u, v \in (N \cup T)^*$, a u obsahuje jeden neterminální symbol. Znaku u říkáme levá strana pravidla, řetězci v 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. To 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. LSystému s takovým 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 LS typu 0. Tento LS používá deterministického rozhodování, které pravidlo použije. Znamená to ovšem, že musí produkovat stále stejný řetězec znaků. To je bohužel pravda, což je velkou nevýhodou tohoto typu LS. Stromy vygenerované tímto druhem LS budou vypadat stále stejně. Tento neduh se snaží řešit tzv. stochastický LS. Ten se od typu 0 liší tím, že má více pravidel, která mají na pravé straně stejný symbol. Při přepisování se pak z těchto pravidel vybírá nedeterministicky, na příklad na základě pravděpodobnosti, kterou má každé pravidlo u sebe uvedenou a s kterou se uplatňuje. Takové LS už produkují pokaždé jiný řetězec. Jejich jazyk je tedy rozmanitější. Jsou tedy pro generování přírodních objektů vhodnější.

To je dobrá vlastnost, ale pro účel mé práce nevhodná. Představme si, že vygenerujeme nějaký osamocený strom, ke kterému se uživatel přiblíží, zapamatuje si jeho tvar a předpokládá, že když se na toto místo vrátí, bude tento strom stejný. To mu stochastický LS tak, jak je definován, neumožní. Má to ale jednoduché řešení. Stačí si zapamatovat pro každý strom ve scéně nějaký seed generátoru pseudonáhodných 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 dokáže 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, jež umí vykonávat příkazy, které jsou jí dávány. Má svůj aktuální stav, součástí kterého jsou atributy jako poloha, směr. Může jich být typicky více. Ale tyto 2 jsou základní. Příkazy pro želvičku jsou na příklad: 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 rozdvojování nasimuluje. Ten představuje s*tavový 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 2 Přístupy 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. Provádí se 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.

Používají se 2 přístupy k této technice. Jedním z nich je statický přístup. Jeho základem je vytvoření několika verzí modelu objektu dopředu. Ve vykreslovací fázi se pak jen tyto verze vyměňují podle vzdálenosti od kamery. Pro zamezení náhlé změny jedné verze za druhou (flicking) se mezi verzemi postupně prolíná.

Druhý přístup je dynamický. Tento, namísto předpřipravených verzí, postupně zjednodušuje určité části modelu za běhu, dynamicky. Tento přístup je kvalitativně lepší, avšak vyžaduje složité algoritmy na výběr části modelu pro zjednodušení. I algoritmy vlastního zjednodušování jsou poměrně komplikované. Významnou součástí takovéhoto LoD je rutina, která bude říkat, kdy 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, jež se v nich nacházejí. Poloměry těchto mezikruží by se měly zvětšovat s nějakou rychleji rostoucí funkcí.

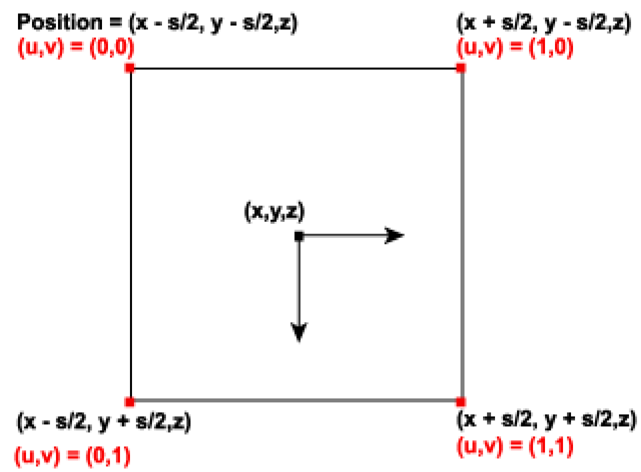
2.2 Point Sprites

PointSprite (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 grafické kartě, nebo úspora množství dat na kartu posílaných. Protože je k jeho definici potřeba jen poloha, stačí pouze jeden vektor, což jsou 3 floaty oproti 18 floatům dvou trojúhelníků nebo 12 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

Moje implementace generování stromů

3.1 Implementace LSystému

Implementován je stochastický LS, aby bylo dosaženo variability generovaných stromů, což bylo jedním z cílů 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

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

Na příklad 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) =…

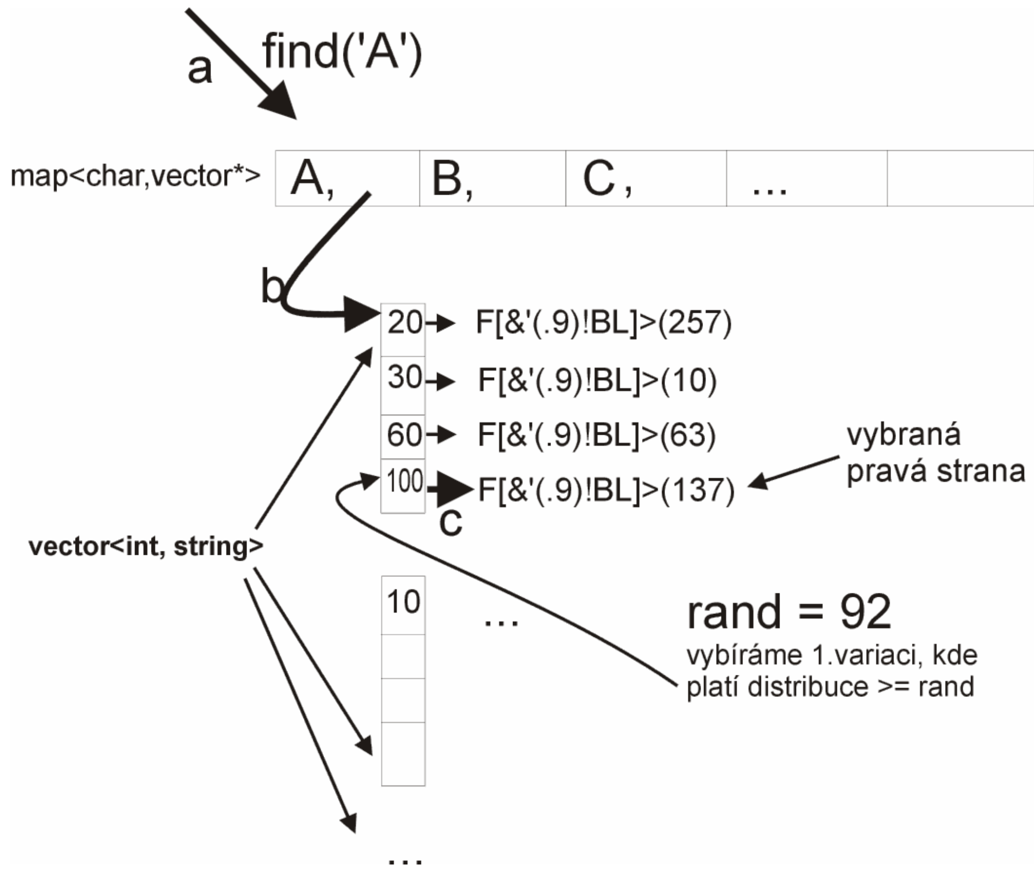
Jedinou výjimkou je axiom, který je zahrnut také mezi pravidla a který je tvaru:
X(100) = ... Hodnota distribuční funkce (100) se nesmí měnit.

Implementace takového 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 začátku 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 skončit, 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 vypadá struktura LS a příklad, jak probíhá výběr variace pravidla ‘A’. (šipky: a = vyhledání pravidla s A na levé straně v asociativním kontejneru. b = přechod na odpovídající vektor variací, c = výběr pravidla na základě náhodné hodnoty rand.)



3.2 Struktura souboru LSystému (.ls)

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

- recursionLevels
- defaultAngle
- defaultThickness
- axiom
- pravidla
- @

V souboru jsou 2 druhy elementů. Jednak jsou to funkční elementy, které jsou vlastními daty pro LSystém, a pak jsou to komentáře, které se 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. Následuje axiom, což je řetězec tvaru shodného s tvarem pravidel (distribuce rovna 100!). 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 samostatného řádku. Platí i pro komentáře. Avšak ty se mohou vyskytnout 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 geometrii, topologii, *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, jež vykonávají příkazy obsažené v řetězci, které mění atributy, ty jsou také v Cursor a 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 obnoví 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 orientační (mění polohu a orientaci želvičky), atributové (mění atributy) a řídící (oznamují na příklad větvení).

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

Tabulka 1, Seznam orientačních příkazů

<table>
<tr><td>f</td><td>Posunutí o délku úseku v aktuálním směru</td></tr>
<tr><td>F</td><td>Posunutí o délku úseku v aktuálním směru a nakreslení úseku</td></tr>
<tr><td>L</td><td>Zaznamenání nového listu</td></tr>
<tr><td>&</td><td>Otočení kolem osy X CW („sklopení hlavy“) *</td></tr>
<tr><td>^</td><td>Otočení kolem osy X CCW („zdvihnutí hlavy“) *</td></tr>
<tr><td><</td><td>Otočení kolem osy Y CCW („otočení hlavy“ vlevo) *</td></tr>
<tr><td>></td><td>Otočení kolem osy Y CW („otočení hlavy“ vpravo) *</td></tr>
<tr><td>+</td><td>Otočení kolem osy Z CW („naklonění hlavy“ vpravo) *</td></tr>
<tr><td>-</td><td>Otočení kolem osy Z CCW („naklonění hlavy“ vlevo) *</td></tr>
<tr><td>|</td><td>Otočení o 180° kolem osy Y („otočení hlavy“ o 180°)</td></tr>
<tr><td>%</td><td>Otočení kolem osy Z o 180°</td></tr>
<tr><td>$</td><td>Otočení do roviny („naklopení hlavy do roviny“ = úhel 0°)</td></tr>
</table>

Tabulka 2, Seznam řídících příkazů

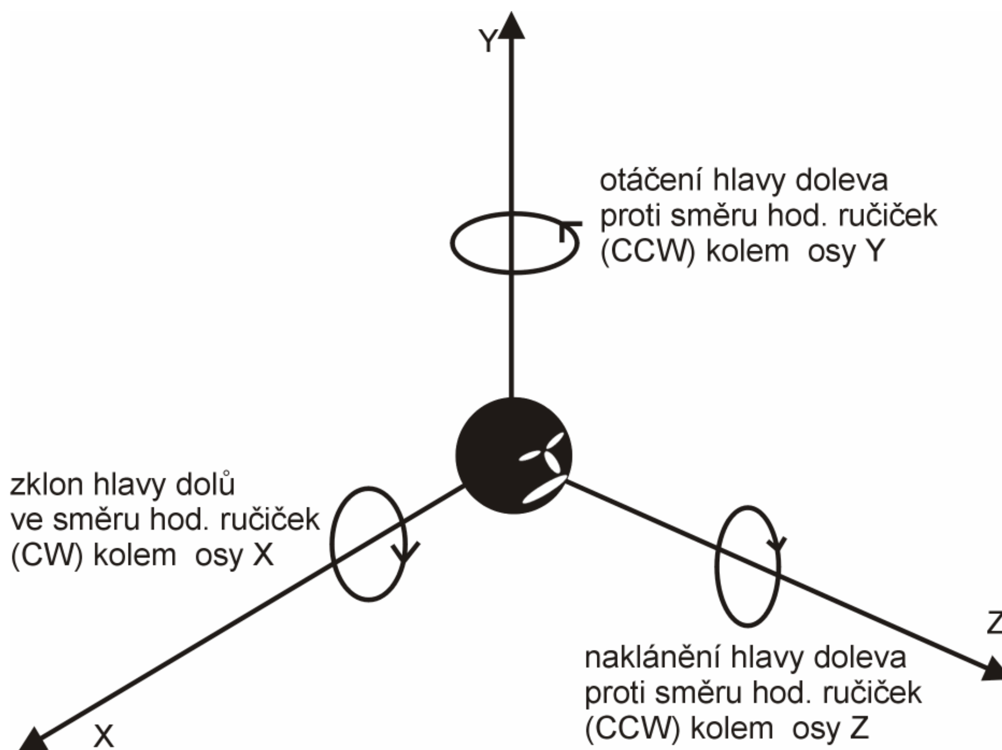
[	Uložení aktuální stav na zásobník přidáním nového Cursoru „rozdvoj se“
]	Smazání vrcholu zásobníku (obnovení předchozího stavu, „obnov stav“)

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 parametru je jakékoliv reálné číslo. Opět vše musí být bez mezer jako v souboru s LSystémem. Na příklad +(67) = otočení kolem osy Z CV o úhel 67°.

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



3.4 Implementace LoD

Základní myšlenkou 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 zblízka, 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 pozorujeme, že tam jsou, mají nějakou barvu, ale jejich obrysy už nerozeznáme. Pro věrohodný a realistický pohled nám stačí pouze to, když pozorovaný objekt – strom – má pořád stejnou barvu a korunou prosvítá světlo. Významným se nakonec stane jen tvar stromu.

Tento tvar determinuje několik prvních větví, které se oddělují od kmene. Je tedy determinován už na začátku svého vývoje. Toho 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 nich další, menší větve, a tak dál, posledním prvkem jsou listy. Takový strom budu nazývat *BranchTree* („strom větví“). Tato reprezentace je pro potřeby takového systému, který dovoluje simulovat 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;
- za druhé je to možnost definovat listy jako *PointSprites*;
- za třetí je to modifikovatelnost. Protože máme strom složen z jednotlivých větví, lze jednoduchými úpravami docílit efektů jako je například pohyb 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 vzdálenosti rozeznáváme a které jsou pro pozorovatele měřítkem kvality modelu. Ovšem s rostoucí 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 scénáři.

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, větší množství není žádoucí i z důvodu velkého počtu těchto index bufferů, protože by to bylo spojeno s velkou režií na grafické kartě. Proto jsem shora 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 další režii pro průchodové algoritmy a i pro správu paměti, kterou by toto patro zabíralo. Proto jsem další 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 jsou později vykreslovány jako *PointSprites*.

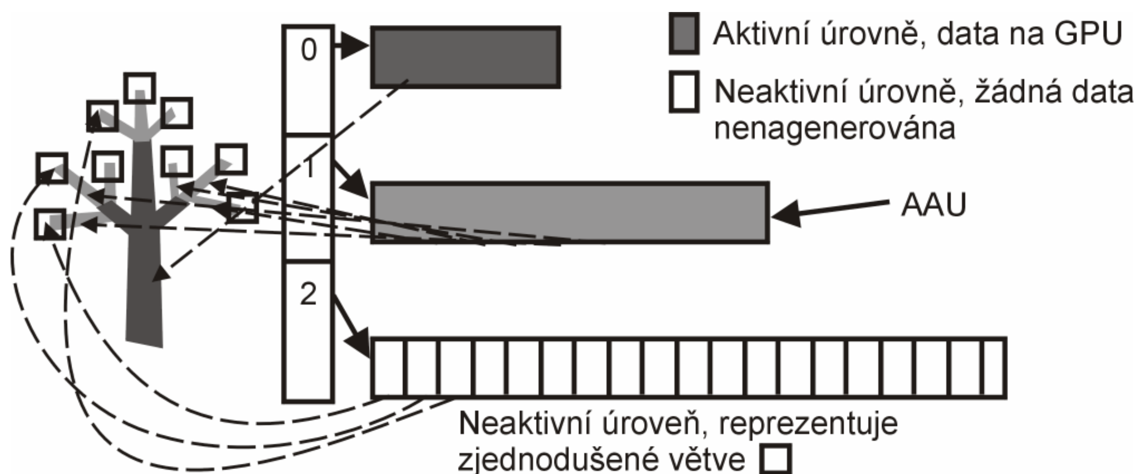
Pro další potřeby je vhodné nazvat příslušné operace LoD zjednodušení *Simplify*, kdy se z komplikované části modelu (větev s listy) vytvoří jednoduchá plocha. Tato plocha ale musí mít stejnou barvu (což je důležité), podobnou strukturu a prosvítá skrz ni světlo. Protichůdná operace je opětovná komplikace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

Protože můj LoD systém je úzce provázán s *PointSprites*(PS), operace *Simplify* převádí geometrii větviček a listů (opět PS) na plochu, která je ve formě dvojrozměrné textury a jež se pak mapuje na PS. Slovo převádí je malinko zavádějící. Ve skutečnosti se vymění složitější geometrie za texturu, ta se vybere (náhodně) z nějakého poolu textur. To ovšem porušuje požadavek na podobnost plochy (textury) s větví a s 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 k *Simplify* dochází, vadí. Kdybychom toto chtěli eliminovat, museli bychom tuto texturu generovat, na příklad vykreslením do *OffScreen textury*. To ale přináší řadu problémů, 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.

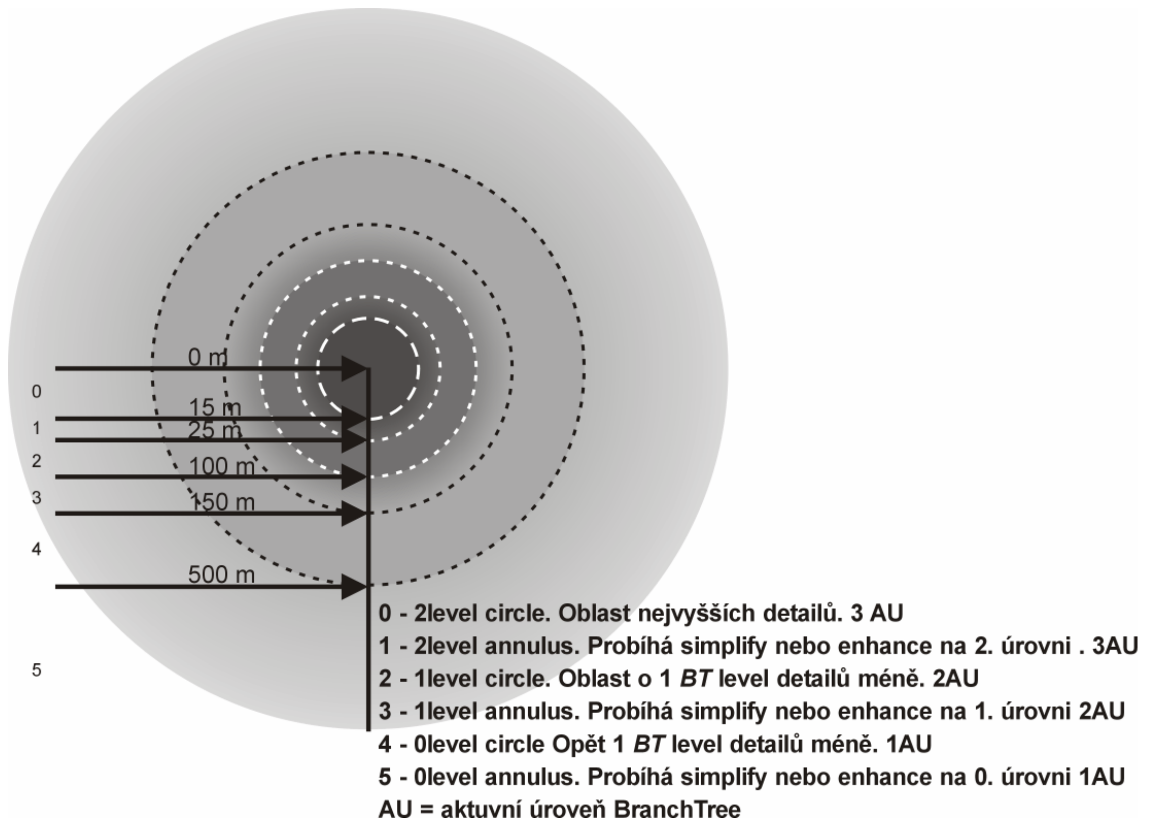
Výběr větví, na které se provede *Simplify* se provádí sekvenčně a z aktuálně nejvyšší aktivní větve (aktuální aktivní úroveň, AAU). Takže je třeba pro ni mít seznam větví, které jsou už zjednodušené (SIMPLIFIED) a které ne (ENHANCED). Aktivní úrovní je úroveň, která má svá data aktuálně na grafické 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. Operace snižující počet aktivních je *DecreaseActiveLevels*. Tato operace zruší data na kartě, pošle nová data, mezi kterými už není námi zrušená úroveň (operace *ReleaseLevel*), smaže úroveň o 1 výše (operace *DeleteLevel*) a ustanoví nižší aktivní úroveň za AAU. Naopak při prázdném seznamu SIMPLIFIED (tj. AAU nemá žádnou větev zjednodušenou) a žádosti o další *Enhance* se provádí operace *IncreaseActiveLevel*. Ta přidává další aktivní úroveň, která je prohlášena za aktuální AAU. Součástí této operace je generování dat (geometrie) úrovně o 1 výše než AAU při současném vygenerování další úrovně (zatím ještě neaktivní) a jejím provázáním *SimplifiedChain*. Následně se zruší data na kartě a pošlou se nová, o přidanou AAU 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 jednotlivými úrovněmi (viz dále). Dále vidíme, že rozdílem mezi aktivní a neaktivní úrovní je to, že aktivní disponuje vygenerovanými daty (prošla operací *Generate*, která generuje geometrii, viz Interpretace Želvičkou) a neaktivní nikoliv. Protože součástí *Generate* je i vytvoření nové, neaktivní, úrovně, můžou se neaktivní úrovně postupně mazat a zůstat může jen jedna (ta, které bude potřeba 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. Aktivní úrovně pak odpovídají částem stromu, které jsou vykreslovány s plnými detaily. Neaktivní pak těm, které jsou zjednodušeny do textury (*PointSprites*). Proto je důležité ji při vytváření provázat *SimplifiedChain*.



Další částí, kterou popíši, je rutina. Ta 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*. Přidána jsou navíc mezikruží, ve kterých probíhá proces přechodu mezi jednotlivými úrovněmi detailu

(operace *Simplify* nebo *Enhance*, podle směru pohybu kamery vzhledem ke stromu), čili mezi počtem aktivních úrovní *BranchTree*. Vše je vidět na obrázku:



3.5 Implementace struktur pro vykreslování

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

- z důvodu rychlosti není v každé větvi informace o 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 větvích jsou tedy jen ukazatele do tohoto společného vertex bufferu. Stejně tak je tomu i u geometrie vlastních větví jako součásti kmene (*BODY*). Slovem *Body* budu nazývat vlastní kmen a větve (bez listů) jakožto útvary trojrozměrné, pro jejichž kreslení jsou 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 normály a texturovací souřadnice. V každé větvi je pak jen index buffer s topologií. Toto je velice redundantní řešení. Bylo však motivováno nezanedbatelnou myšlenkou, že grafické akcelerátory mají rády velké a hlavně

celistvé objemy dat. Dále jsem použil pro Body a PointSprites rozdílné vertex buffery, aby k žádným redundancím nedocházelo a obětoval tak menší množství index a vertex bufferů na kartě spojených s režií jejich udržování a správy za cenu menšího množství dat v nich;

- další optimalizací je změna průchodu *BranchTree*. Představíme-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 aktuální větev (v tomto případě 1. úroveň *BranchTree*) za pomoci jejího index bufferu. Protože index buffer je pointer ukazující na grafickou paměť, 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 větvi se nakreslí listy aktuální větve. Následně se přejde na potomky aktuální větve (na další patro) ve stromu větví. Tento proces se pak opakuje pro každého z těchto potomků. Je to tedy klasický rekurzivní průchod stromem DFS. Tento způsob je ale značně neefektivní. Střídá se totiž kreslení *Body* a *PoitSprites*. Pokaždé se neustále přepínají vertex buffery. Cílem tedy je nejdříve nakreslit *Body* a pak *PoitSprites*. Abychom nemuseli procházet 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 ho stačí měnit jen v případě, že se strom větví nějak změní (např. při některé z operací systému 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. To 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 jejich pole o velikosti NUMBER_OF_PS_TEXTURES.

Takto jsem naimplementoval první optimalizovanou verzi. Ta sice fungovala dobře, ale neustále se vyskytovala 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 tu cenu, že bude více bufferů, které musí GPU obhospodařovávat. Dále tato implementace obsahovala moc indexbufferů, které jsou také náročné na správu 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 nemusí 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.

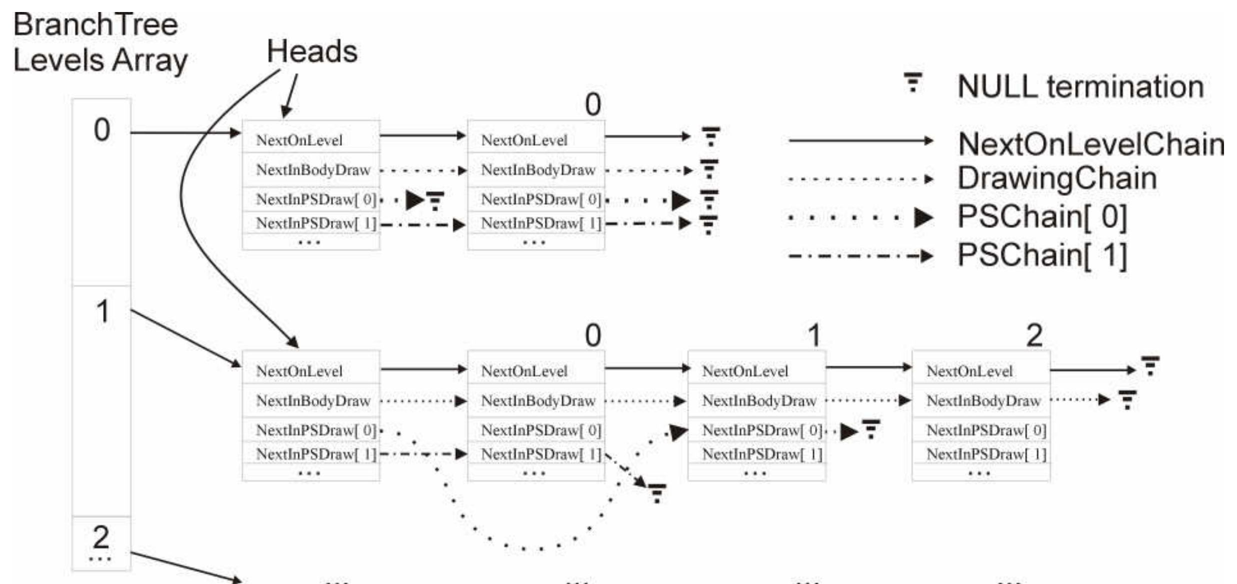
Ale úpravy pokračovaly. Tentokrát to bylo třeba proto, že jakmile máme buffery společné pro celý strom, nemáme možnost některou jejich část rušit a opět přidávat tak, jak to vyžaduje LoD. Toto vyhazování a přidávání se děje po vrstvách *BranchTree*. Čili další verze rozdělila 1 sadu společných bufferů 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 každém 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 mohla existovat jen ve formě bufferů na kartě. To ale nepřináší žádné další paměťové nároky, protože driver, když je buffer vytvořen s flagem POOL_DEFAULT, stejně vytváří kopie někde v lokální systémové paměti pro případ resetu zařízení. Pak stačí vytvářet buffery pouze v paměti GPU a resetovací proceduru naprogramovat ručně, a jsme na stejných paměťových nárocích jako předchozí verze.

Nyní již máme fungující buffery, kterých není zbytečně moc, ve větvích jen nezbytně nutná data a systém je připraven na LoD. Schází již jen systém, který by sdružoval větve, které se mají aktuálně vykreslit. Jinými slovy nějaký seznam větví, které se mají vykreslit ENHANCED (čili v plných detailech), jiné, které se mají vykreslit SIMPLYFIED (čili jen jako textury na *PointSprites*). Potom seznamy větví, které obsahují jednotlivé typy listů, aby se mohli kreslit postupně všechny listu jednoho typu. To vše pro každou úroveň *BranchTree*. Toto vše by mělo být přístupné 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. Nebylo to ale příliš flexibilní řešení, protože jich bylo nutno mít pro každou kreslenou část a pro každou úroveň *BranchTree* neúnosně 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 pointerů, 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 spojovými 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 jedna 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, jež 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

<u>Třída CScene</u>

<u>Popis:</u>

Reprezentuje scénu. Obsahuje jednotlivé druhy. Jejich správu má na starost asociativní kontejner map, který druh asociuje s jeho jménem.

<u>Důležité členské proměnné, metody, definice typu:</u>

```
// definice typu kontejneru
typedef map< string, CKind* > TKindMap;

// přidává strom na základě jména jeho druhu na zadanou polohu
void AddTree (string, D3DXVECTOR3 &).
```

<u>Třída CKind:</u>

<u>Popis:</u>

Reprezentuje druh. V každém tomto objektu jsou stromy jednoho druhu. Existují tak objekty CKind např. pro břízu, javor apod. Objekty tohoto typu jsou uvnitř kontejneru v CScene.

<u>Důležité členské proměnné, metody, definice typu:</u>

```
// definice typu pro les. Vector stromů
typedef vector< CNO*> FOREST;

// zjišťuje zda se některý strom nedostal do jiného kruhu nebo mezikruží (viz LoD), případně na to zareaguje nastavením stromu odpovídajícího statusu
void Update( D3DXVECTOR3 &);

// podle statusu vykonává operace LoD
void TickUpdate( D3DXVECTOR3 &);

// při přidávání je třeba zajistit, aby se vygeneroval správný počet aktivních úrovní. Podle toho v jakém kruhu, mezikruží se na začátku strom nachází
void InitTree( CNO *, D3DXVECTOR3 &);

// vykreslí všechny PointSprites všech stromů v objektu.Postupně podle typu listu. Optimalizuje tím vykreslování, protože se tolik nestřídají textury
bool DrawPS( CNO *);

// vykreslí Simplified Chains všech stromů v objektu
bool DrawSiplifiedBranches( CNO *);
```

<u>Třída CNO</u>

<u>Popis:</u>

Reprezentuje model stromu. Obsahuje *BranchTree* a plus další pomocné informace, např. kde (na jakém offsetu) začínají data jednotlivých BranchTree levels, protože tato data jsou v kontejnerech sdílených jednotlivými úrovněmi. Obsahuje také buffery (Vertex, Index), které se následně odesílají na GPU.

<u>Důležité členské proměnné, metody, definice typu:</u>

```
// hlavy úrovní BranchTree
CBranch m_LevelHeads[NUM_LVLS];

// Matice s pozicí a orientací vůči světu
D3DXMATRIX m_worldMatrix;

// kontejnery na data větví. Vertexy BODY, indexy, vertexy PointSprites
BODYVECTORS m_bodyVertices;
VECTORINDECES m_bodyIndeces;
VECTORS m_PS;
```

```
// nakreslí BODY (větve bez listí) celého stromu. Projde všechny aktivní
úrovně.
bool Draw (LPDIRECT3DDEVICE9, bool);

// provádí operaci DisposeLevel
void DisposeLevel (int);

// provádí operaci delete level
void DeleteLevel (int);

// to samé s operací Simplify
void Simplify (void);

// to samé s Enhance
void Enhance (void);

// provádí Simplify dokud je co v AAU
void ForceSimplify (void);

// to samé s Enhance
void ForceEnhance (void);
```

Třída CBranch:

Popis:

Reprezentuje větev. Objekty této třídy tvoří BranchTree. Obsahují matice s polohou a orientací, dále svůj seed pro generátor. Dále obsahují informace nutné pro opětovné vygenerování, což jsou vlastně atributy „želvičky" v okamžiku, kdy se dostala na tuto větev. Uložené jsou v char pro úsporu místa.

Důležité členské proměnné, metody, definice typu:

```
// BODY Node
typedef struct ibnode_ {
        unsigned int pToVert;           // offset do vertex bufferu
        unsigned int pToIndx;           // do index bufferu
        unsigned short count;           // počet primitiv (trojúhelníků) na
vykreslení
} sIBNode;

// PointSprite Node
typedef struct psnode_ {
        unsigned int startVertex;       // offset do vertex bufferu PointSprites
        unsigned short count;           // počet PointSprites na nakreslení
}sPSNode;

// typ pro potomky
typedef vector< CBranch *> CHILDREN;
```

```
// výčet možných statusů větve
typedef enum status_ {
        NOTGENERATED,  // nevygenerováno
        ENHANCED,           // nakresleno v plných detailech
        SIMPLYFIED,         // zjednodušeno do textury na PointSprite
        SIMPLIFIED_BUT_NOT_DRAWN,         // nevykreslováno pří
vykreslování potomků
} STATUS;

// zapouzdřuje jednotlivé Chains
typedef struct sChainPointers_ {
        CBranch *NextOnLevel;       // soused v BranchTree úrovni
        CBranch *NextInBodyDraw;// další v DrawBody Chain
(ENHANCED)
        CBranch *NextSimplified;    // Simplified Chain
        // další v Chains pro jednotlivé typy listů
        CBranch *NextInPSDraw [NUMBER_OF_PS_TEXTURES];
} TChainPointers;

// nese informaci o poloze a orientaci vzhledem k modelové souřadné
soustavě
D3DXMATRIX m_modelMatrix;
```

Třída LSystem:

Popis:

Reprezentuje vlastní LSystém. Obsahuje definice typu pro správu pravidel. Procedury pro načítání/ukládání z/do souboru i do edit boxů. Stejně tak i na vlastní přepisovací proces.

Důležité členské proměnné, metody, definice typu:

```
// definice buňky ve vektoru variací pravidla
typedef pair< short, string> varCell;

// definice kontejneru variací jako vektoru buněk
typedef vector< varCell> variationType;

// definice typu kont. pro jednotlivá pravidla
typedef map< char, variationType*> rulesType;

// vrátí řetězec vzniklý n-násobným přepisováním neterminálu
string getNRewrites (char neterminal, char n);

// načte LSystém z uvedeného souboru
bool Load (char*);
```

Třída CursorStack:

Popis:

Tento objekt vykonává veškerou práci spojenou s generováním geometrie modelu. Provádí i některé operace LoD, které potřebují generování geometrie. V proceduře GenerateBranch se provádí generovaní pomocí „želvičky“ za pomoci zásobníku stavů.

Důležité členské proměnné, metody, definice typu:

```
// definice typu zásobníku na stavy
typedef stack< Cursor*> TCURSOR_STACK;

// následující metody provádějí operace tak, jak jsou popsány v odstavci o
LoD
bool IncreaseLevel (CNO *);

bool DecreaseLevel (CNO *);
bool GenerateBranch (CBranch*, CNO*, bool);

// vytvoří z kontejnerů ve stromě (CNO) a pošle data na GPU
bool CreateBuffers (CNO *, int);

// vygeneruje vertexy podstav kvádrů, kterými se kreslí BODY
void DrawSegment (void);

// přidá do kontejnerů ve stromě nová data po operaci IncreaseLevel
void AppendNewDataToTree (CNO *, int);
```

Třída Cursor:

Popis:

Reprezentuje stav kreslící „želvičky“. Obsahuje proměnné, které stav obsahují, a metody, které s ním pracují a vykonávají tak vlastně příkazy „želvičce“. Objekty tohoto typu jsou v zásobníku stavů v CursorStack.

Důležité členské proměnné, metody, definice typu:

```
// následující proměnné jsou vlastně atributy „želvičky“
// poloha a orientace
D3DXMATRIX m_matrix;

// délka úseků, tloušťka, defaultní úhel (viz LSystém)
float m_length;
float m_thickness;
float m_defAngle;

// aktuální hodnoty úhlů jednotlivých orientací
float m_yaw;
float m_pitch;
float m_roll;
```

```
// některé metody vykonávající příkazy pro „želvičku”
void GoForward (void);
void pitch (float);
void yaw (float);
void roll (float);

// některé metody vykonávající příkazy změny atributů „želvičky”
void ChangeLength (float m);
void ChangeThickness (float m);
```

Kapitola 4

Možná zlepšení

4.1 Možné přístupy jak zlepšit kvalitu zobrazení

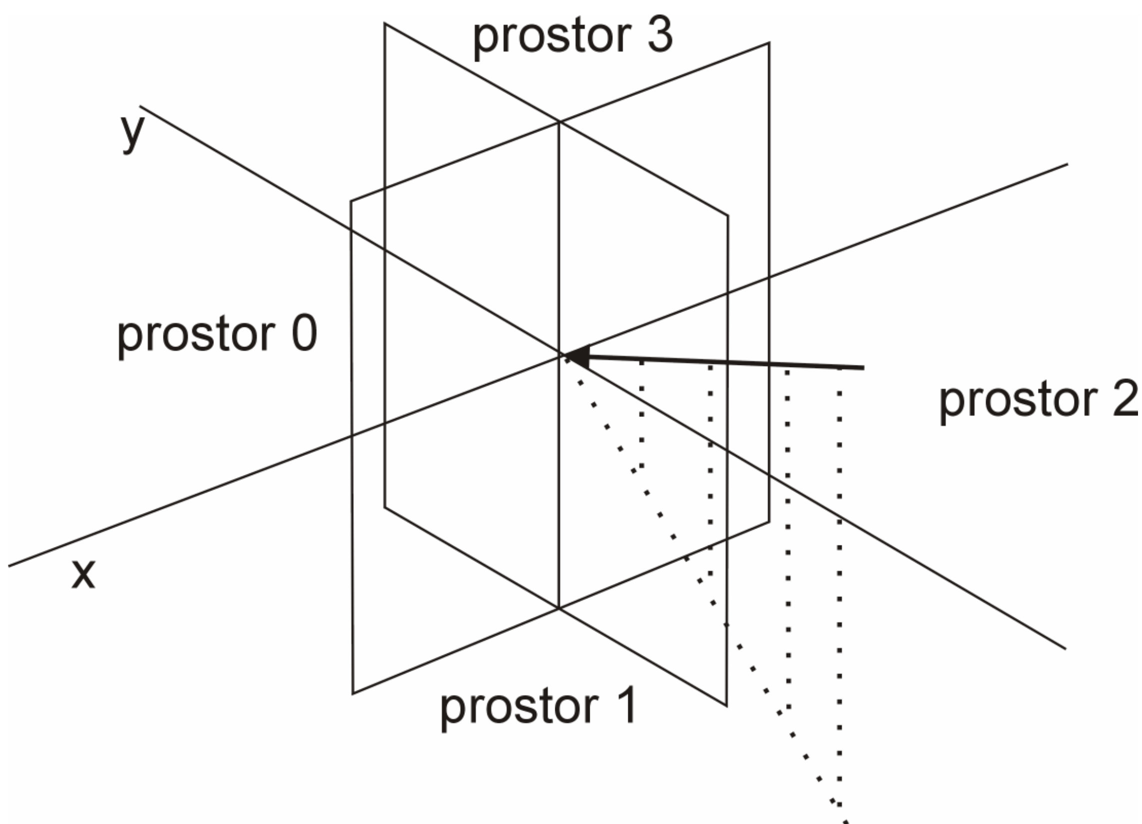
Zamyslíme-li se nad nedostatky našeho dosavadního systému, dojdeme k jedné zásadní věci, která ruší realistický vjem. Je to ono porušení předpokladu o podobnosti struktury a obrysu při operaci *Simplify*. Jak již bylo naznačeno, to by se dalo eliminovat tak, že by se místo zjednodušované větve nepoužila předem připravená textura z nějakého poolu textur, ale ta by se dynamicky 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 pohybuje, 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 konstantu, OffScreen textura se vygeneruje znovu z aktuálního pohledu. Tato metoda ale předpokládá, že je z čeho generovat. Tzn., že musíme mít ještě k dispozici data celé větve a to po celou dobu života *Simplified* textury. Uvážíme-li, že máme více úrovní *BranchTree,* a zjednodušujeme po úrovních, musíme tedy držet v pamětech (jak Systémové, tak i GPU) celý strom a LoD zaměřit tedy jen na zjednodušení vykreslování, nikoliv na snížení hardwarových nároků (paměť). Spočítáme-li si, že strom, který má v plné kvalitě 500 kB jen v paměti GPU, a kreslíme scénu, která obsahuje 1000 stromů, zabere nám 0.5GB paměti GPU. To je neúnosná hodnota. Dalším nezanedbatelným objemem dat na GPU jsou právě ony vygenerované textury. Protože pro každou větev máme jednu, je to další neúnosný objem dat.

Jisté nedostatky v zobrazování ale přetrvaly. Na příklad to, že když regenerujeme textury s každým větším pohybem kamery, tak se nám opět skokově změní, to působí pro uživatelův pohled dost rušivě. Nastává to také vždy, když provádíme opět operaci *Enhance*. Situaci můžeme 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. Méně rušivě působí zjednodušení odvrácených partií stromu (z aktuálního pohledu), než na obrysech a vpředu. Když už nemáme žádné odvrácené větve pro *Simplify*, používáme ty vpředu a až nakonec ty 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 vědět, jaké větve jsou v kterém segmentu, 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. Nepřinese to žádné zpomalení díky vysokým nárokům na hardware, protože stačí modifikovat vlastnosti pouze na úrovni *DrawChains*, kterých je pouze 10 (NUM_OF_PS_TEXTURES).

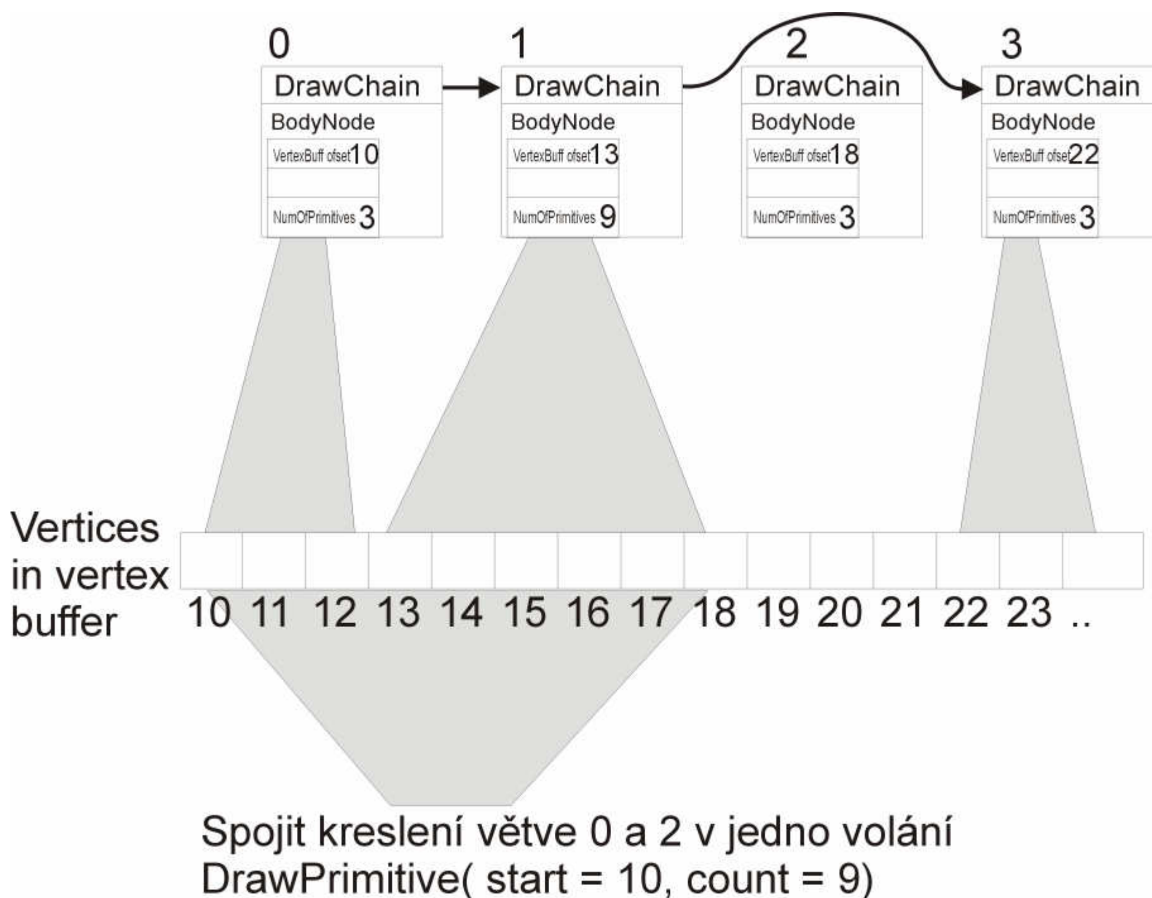
4.2 Možné přístupy zefektivnění vykreslování

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 sdružení je možné realizovat zapouzdřením 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 kontejner 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 kontejner. Pro každý „druh“ v tomto kontejneru 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 uživatelského režimu do režimu jádra a zpět. Tyto přechody trvají nezanedbatelně dlouho. Lze je eliminovat tím, že se jednotlivá volání seskupí do jednoho. Protože v *DrawChain* máme informace jako offset do vertex buffer, počet vykreslovaných primitiv, můžeme volání funkcí realizující vykreslování (DrawPrimitive) 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*. Ta primitiva, které jí reprezentují, se při vykreslování již neuplatní, tudíž nám porušují kontinuitu dat pro spojení vykreslovacích volání. Následuje obrázek. Větev 2 již prošla *Simplify* a zkazí nám spojení vykreslení všech 4 větví do 1 volání. Ale 0 a 1 spojit můžeme.



Každopádně tímto způsobem vykreslování urychlíme. V nejhorším případě si nepomůžeme. Tento případ nastane, střídají-li se v *DrawChain* větve, které jsou ENHANCED a SIMPLYFIED. To 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 tehdy, pokud nechceme mít pro každou větev její vlastní modelovou matici nutnou např. pro simulaci ohýbání větví větrem. Pak je nutné již při fázi generování generovat vertexy s pozicí vzhledem k celému stromu. Tímto způsobem je možné vykreslování optimalizovat. V případě, že pro každou větev máme její vlastní modelovou matici, vertexy jsou generovány s pozicí vzhledem k bázi této větve a při vykreslování je nutné tuto matici před každým voláním poslat na GPU. Potom není možné tyto úvahy implementovat.

Kapitola 5

Závěr

Na závěr se sluší udělat jisté zhodnocení a rekapitulaci. Má práce byla pokračováním předchozího ročníkového projektu. Jeho cílem bylo vyzkoušet použití *PointSprites* při vykreslování přírodních objektů. Výsledkem byl systém, který nejdříve přírodní objekt vygeneruje, aby ho mohl poté zobrazit. Základem takového systému je LSystém, což je vlastně gramatika s určitými modifikacemi pro použití při generování objektů, které známe z přírody. LSystém pomocí přepisování definovaného pravidla generuje řetězce znaků. Protože LSystém vygeneruje pouze řetězec znaků, je třeba ho nějak interpretovat, pravděpodobně tak, aby se z něj stala geometrie nějakého trojrozměrného modelu. K tomu slouží „vykreslovací želvička“, což je entita definovaná svým stavem, jakým je např. poloha a orientace v prostoru. Pro umožnění větvení je přidán zásobník na tyto stavy. S jeho pomocí můžeme docílit libovolně rozvětvené struktury.

Pro vykreslování listů modelu jsou použity právě *PointSprites*. To jsou vlastně čtverce orientované v prostoru vždy směrem ke kameře, na které lze aplikovat texturu. Proto je možné si je představit jako Billboardy. Mají oproti nim několik výhod. Jsou natočeny ke kameře automaticky, takže to nemusí řešit program, znamenalo by to i určitou časovou ztrátu. Další výhoda je v tom, že jsou definovány pouze jedním bodem, což přináší nemalé úspory hardwarových prostředků, především paměti a kapacity sběrnice. Výhody jsou ale zastřeny nevýhodou spočívající v nemožnosti s nimi jakkoli manipulovat v prostoru. Mohou se pouze škálovat a měnit jejich textura. Výsledkem ročníkového projektu je program pro editaci souborů s LSystémem, který ihned zobrazuje modifikace jeho parametrů. Pro ty je k dispozici sada ovládacích prvků. Včetně textboxů pro úpravu pravidel LSystému. V oddílu Screenshots je jeden z tohoto programu v době po skončení ročníkového projektu i se zobrazeným modelem stromu.

Cílem této práce je pak tento program rozšířit. První rozšíření se týká vylepšení rozmanitosti generovaných modelů. Protože LSystém z programu ročníkového projektu nepoužívá žádných náhodných elementů, generuje stále stejné řetězce, ze kterých vznikají stále stejné modely. To můj LSystém odstraňuje použitím náhody, na základě generátoru pseudonáhodných čísel. Takovýto LSystém se nazývá stochastický a generuje pokaždé jiný řetězec. To má za následek generování pokaždé jinak vypadajících modelů a simuluje rozmanitost jedinců stejného druhu v přírodě.

Další rozšíření se týká systému LoD (Level of Detail). Ten se používá pro zjednodušování vykreslovaných modelů s rostoucí vzdáleností ve scéně. Objekty v pozadí divák vnímá mnohem méně intenzivněji než ty v popředí. Proto mohou být kresleny s řádově menším množstvím detailů a tím je umožněna mnohem vyšší rychlost vykreslování. Proto může být ve scéně více takovýchto modelů a GPU je bude stíhat všechny nakreslit.

Tento systém jsem založil na tom, že strom nebo i jiný přírodní objekt je hierarchická struktura. Je vlastně předlohou pro datovou strukturu zvanou (jak jinak) strom. Skládá se z úrovní větví. Kořen stromu je také větev. Když do větve zahrneme i její listy, získáme tak homogenní strukturu. Ta je i základem mých datových struktur. Hierarchičnost těchto struktur je obrovskou výhodou pro spojení se systémem LoD. Jednotlivé úrovně tak odpovídají úrovním detailu. Je výhodou i pro spojení s *PointSprites*. Totiž úrovně, které nemusí být kresleny detailně, se nakreslí pomocí několika PointSprites s modifikovanou velikostí podle toho, jaké úrovni stromu zjednodušená úroveň odpovídá. Listy se kreslí na malých *PointSprites*, větší celky stromu, jako na příklad celé větve, pak na větších *PointSprites*.

Protože mám datové struktury rozdělené do úrovní, musel být patřičně modifikován i LSystém a jeho atributy, což si následně vyžádalo i modifikaci ovládání programu, a to o větší počet *RecursionLevels*. Tato čísla odpovídají počtu přepisování LSystému na jednotlivých úrovních. Správně zvolit hodnoty těchto atributů je základem dobře vypadajícího výsledku. Screenshot z nové verze programu je v sekci Screenshots

Podle mého názoru je použití *PointSprites* dobrým nápadem, pokud netrváme na vysoké realističnosti modelů. Výhodný je nepochybně už z hlediska úspory hardwarových prostředků. Má ale i své nevýhody. Největší z nich je ona neflexibilita – nedají se natáčet v prostoru a tím simulovat různorodé natočení listů. Listy tak vypadají všechny stejně. Toto řeší použití více druhů listů (jinak natočených, barevných), ale různorodost, tak jak ji známe z přírody nenasimulujeme. Další, asi nejzásadnější nevýhodou, je nemožnost aplikovat světlo na *PointSprites*. 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, nebo různá nasvícení jednotlivých listů z různých pohledů a ani jiné efekty související se světlem a osvětlením.

Seznam odborné literatury

[1] *Algorithmic Botany* [online], URL: <www.algorithmicbotany.org>

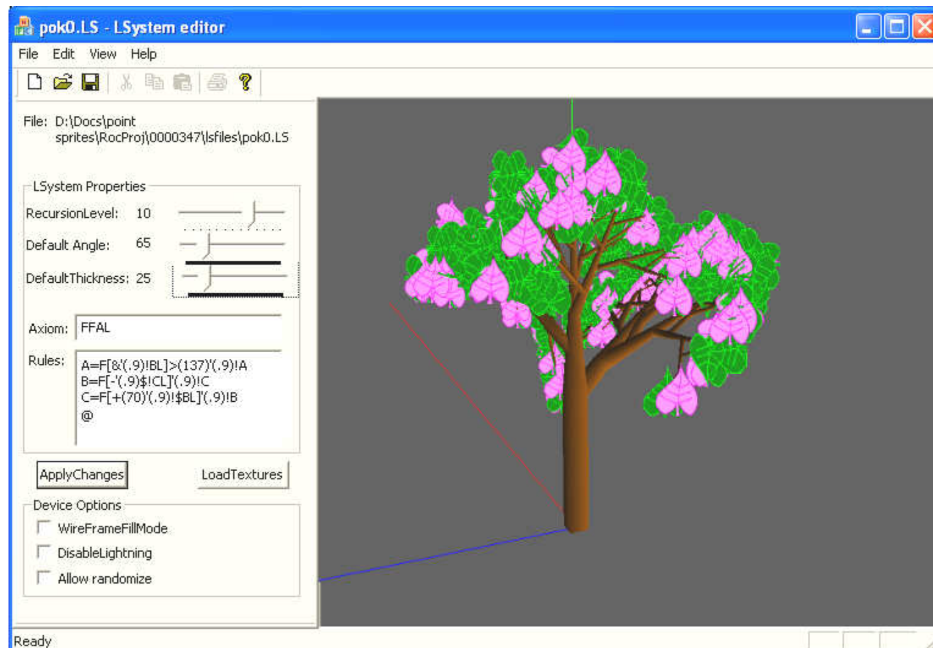
[2] *DirectX SDK documentation* [online],
URL: <www.microsoft.com/windows/directx>

[3] *Microsoft MSDN* [online], URL: <www.microsoft.com/msdn>

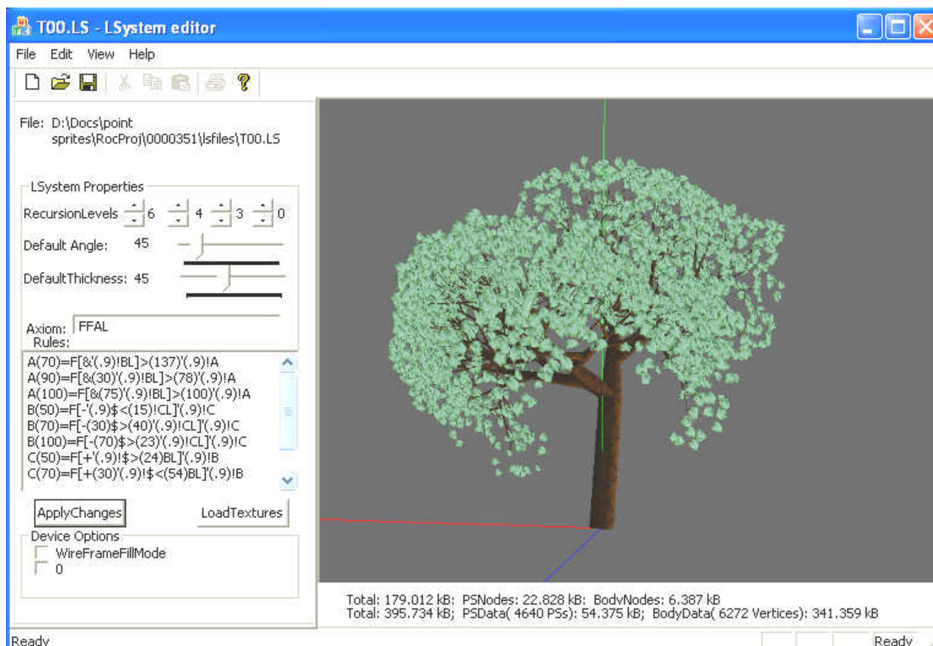
[4] *DirectX tutorials* [online], URL: <www.two-kings.de>

Kapitola 6 Screenshots

Takto vypadal program v závěrečné fázi Ročníkového projektu. Díky LS typu 0 byla struktura stromu po každém vygenerování stále stejná. A vznikaly pravidelnosti, jak je možné vidět na první větvi, která se odděluje od kmene.



Další screenshot je již z finální verze bakalářského projektu. Je vidět, že listy působí určitým umělým dojmem. Je to díky absenci osvětlení.



Následuje dvojice stromů bez listí pro demonstraci jinak vypadajících instancí při použití stejného LSystému.