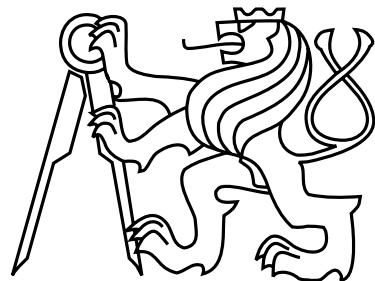


**Na tomto místě bude oficiální zadání  
práce**



České vysoké učení technické v Praze  
Fakulta elektrotechnická  
Katedra počítačů



Diplomová práce

**Realistické zobrazování modelů vegetace v reálném čase**

*Bc. Adam Kučera*

Vedoucí práce: Ing. Jiří Bittner, PhD.

Studijní program: Otevřená informatika, strukturovaný, Navazující magisterský  
Obor: Počítačová grafika a interakce

29. prosince 2011



## Poděkování

Velice rád bych touto cestou poděkoval vedoucímu práce Ing. Jiřímu Bittnerovi, PhD. za podnětné vedení práce a možnost účastnit se odborné studentské stáže na Technische Universität Wien, kde vznikaly metody animace stromů reprezentovaných plochými obrázky. Rovněž jemu a doc. Ing. Vlastimilu Havranovi, Ph.D. patří dík za možnost testovat aplikaci na výkonné školní sestavě.

I would also like to express my thanks to Mr. Ralf Habel for inspiring consultations and providing advanced leaf texture data.

Chtěl bych také poděkovat svým rodičům za to, že mi byli vždy oporou a poskytli mi vynikající zázemí.

V neposlední řadě musím ocenit i své kamarády, kteří měli trpělivost, když na ně nezbývalo tolik času, a nepřestali mi nabízet radost z přátelství.

Děkuji.



## Prohlášení

Prohlašuji, že jsem práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.

Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Liberci dne 29. prosince 2011

.....



# Abstract

Translation of Czech abstract into English.

# Abstrakt

Abstrakt v češtině

Očekávají se cca 1 – 2 odstavce, maximálně půl stránky.



# Obsah

<b>1</b>	<b>Úvod</b>	<b>1</b>
<b>2</b>	<b>Popis problému, specifikace cíle</b>	<b>3</b>
2.1	Vymezení cílů . . . . .	3
2.2	Stávající řešení . . . . .	5
2.2.1	Trojrozměrná geometrická reprezentace . . . . .	5
2.2.2	Reprezentace založené na obrázcích . . . . .	7
2.2.3	Volumetrické zobrazování a bodové mraky . . . . .	8
2.2.4	Zhodnocení stávajících řešení . . . . .	10
<b>3</b>	<b>Analýza a návrh řešení</b>	<b>11</b>
3.1	Animace a vykreslování geometrického 3D modelu . . . . .	11
3.1.1	Deformace větví . . . . .	12
3.1.2	Hierarchická deformace . . . . .	16
3.1.3	Deformace listů . . . . .	18
3.1.4	Řízení animace . . . . .	19
3.2	Zobrazování listů . . . . .	21
3.3	Úrovně detailu . . . . .	26
3.3.1	Animace v image-base modelu . . . . .	27
3.3.2	Řízení úrovně detailu . . . . .	33
3.3.3	Přechody mezi úrovněmi detailu . . . . .	34
<b>4</b>	<b>Realizace</b>	<b>37</b>
4.1	Použité technologie, frameworky a knihovny . . . . .	38
4.2	Architektura LOD . . . . .	39
4.3	Vstupní data a předzpracování . . . . .	41
4.3.1	Formát OBJT . . . . .	42
4.3.2	Textury . . . . .	43
4.3.3	Předzpracování pro LOD . . . . .	45
4.3.4	Dynamické parametry . . . . .	48
4.4	Zobrazení geometrického modelu . . . . .	49
4.5	Zobrazení nižších LOD . . . . .	51
4.6	Sezónní barvy a barevné variace . . . . .	55
4.7	Stíny . . . . .	56
<b>5</b>	<b>Testování</b>	<b>59</b>

<b>6 Závěr</b>	<b>67</b>
6.1 Možnosti vylepšení a dalšího rozvoje . . . . .	67
<b>A Seznam použitých zkratek</b>	<b>77</b>
<b>B Instalační a uživatelská příručka</b>	<b>79</b>
<b>C Obsah přiloženého DVD</b>	<b>81</b>

# Kapitola 1

## Úvod

Jedním z cílů oboru počítačové grafiky je vytvářet tak realistický obraz, jak je to jen možné. Takový úkol má ovšem vždy dvě hlediska: jak dobré je známa teorie popisující zobrazovaný jev a jak je možné tuto teorii využít v praxi. Omezení ze strany hardwaru či jiné požadavky (např. časová / paměťová náročnost) pak mohou učinit úlohu velmi těžkou i pro jevy z teoretického hlediska lehce popsatelné. Složitost vizualizace (kromě jiných) silně závisí na složitosti zobrazované scény. Zobrazování venkovních scén s velkým množstvím vegetace je považováno za obtížnou úlohu počítačové grafiky, právě kvůli vysoké složitosti takových scén. Věrné real-time renderování každého stébla trávy a každého listu stromu až do nejmenších detailů je na současném hardware nerealizovatelné, pokud ovšem nepřipustíme určitou relaxaci problému. V real-time počítačové grafice je třeba občas rezignovat na dokonalé zobrazení přesně podle příslušné teorie. V takovém případě se požaduje, aby přibližný výsledek byl i přesto velmi podobný přesnému řešení a pozorovatel pokud možno rozdíl nepoznal. Naštěstí existují metody, které zachovávají přijatelnou kvalitu a výrazně snižují nároky na výkon. Jedna z nejvýznamnějších skupin technik vychází z faktu, že detail zobrazeného objektu je limitován svou zobrazenou velikostí na výstupním zařízení. Díky konečnému (a vcelku nízkému) rozlišení obrazovky se zobrazení některé detaily do velikosti pod prahem definovaným Nyquist-Shannonovým vzorkovacím teorémem a mohou proto být vypuštěny. To může značně snížit složitost scény. Další zjednodušení scény lze zavést na základě vlastnosti lidského vnímání. Některé části scény mají malý vliv na celkový vjem pozorovatele. V rámci takových částí je možné lišit se od exaktního řešení i velmi podstatně aniž by se dojem pozorovatele výrazně zhoršil. Metoda řízení úrovně detailu (Level Of Detail - LOD) využívá právě těchto poznatků. Se využitím výkonem výpočetní techniky využívají i požadavky na kvalitu zobrazování. Nepřekvapí tak, že zobrazení pouhého statického stromu již nevyvolává v uživatelích takový dojem realističnosti, jako vykreslení stromu, který se hýbe vlivem působícího větru. Zohledněním dalšího rozsahu problému zobrazování vegetace, kterým je čas, se úloha stane ještě složitější.



## Kapitola 2

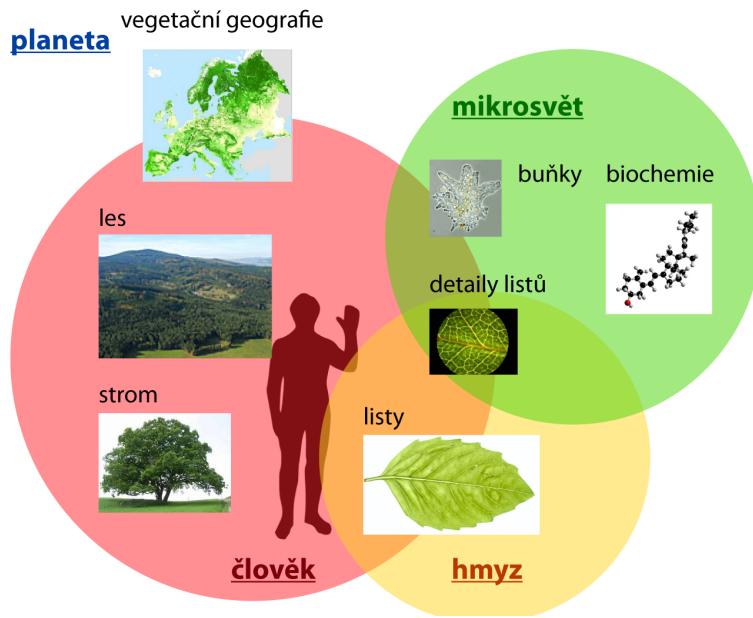
# Popis problému, specifikace cíle

Jak je již předesláno v předchozí kapitole, komplexní objekty, jako je právě například vegetace, představují v oboru počítačové grafiky zvláštní kategorii s relativně vysokou složitostí jejich zpracování. Celá úloha se rádově komplikuje, požadujeme-li zobrazování v reálném čase. Problém zobrazování vegetace lze mírně abstrahovat na úlohu zobrazování stromů. S vývojem hardware a aplikací, které umožňují uživateli interakci s virtuálním prostředím, se zvyšují požadavky na realističnost zobrazované vegetace. Je vyžadováno nejen věrné zobrazení statického stromu, ale také jeho animace. Uvědomíme-li si však, že průměrně rostlý listnatý strom může mít až miliony jednotlivých listů a uživatel požaduje zobrazení ne jednoho stromu, ale hned celého lesa o několika tisících stromů, je přímý a naivní přístup k řešení problému příliš náročný pro současný běžný hardware. Přitom se jedná pouze o prostředí, které dotváří scénu a má významově nižší prioritu než objekty zájmu uživatele (budovy, vozidla, speciální efekty, ostatní hráči - příklady ze světa počítačových her) a tedy by jeho zpracování a zobrazování nemělo vytěžovat ve větší míře dostupné prostředky počítače. V následující sekci budou vytyčeny cíle této práce a definována úroveň podrobnosti, které se chce práce přiblížit. Následující text se zaměří na rešení aktuálního stavu v oblasti real-time renderování vegetace pro vymezenou úroveň podrobnosti. V další kapitole ( Analýza a návrh řešení ) bude rozpracován teoretický základ pro implementaci softwaru zobrazujícím vegetaci v reálném čase. Kapitola Realizace popíše specifika výsledné implementace a řešení postatných problémů vyplývajících z její realizace. Ověření a zhodnocení kvalit vytvořeného software se bude věnovat kapitola Testování. Poslední kapitola diskutuje výsledky práce a nastíní možné způsoby vylepšení výsledné implementace.

### 2.1 Vymezení cílů

Hned na úvod je třeba vyjasnit, jaké úrovni podrobnosti (detailu) se tato práce bude snažit dosáhnout. Ačkoliv má pojem „úroveň detailu“ (Level Of Detail - LOD) v oboru CGI více-méně technický význam, je možné ho chápát i jinak, a to bez znalosti světa počítačů. V této práci jde o zobrazování rostlin – konkrétně stromů. Úrovní detailu, kterými můžeme pohlížet na stromy, je celá řada (viz obr. 2.1).

Pokud budeme na chvíli chápát úroveň detailu tímto způsobem, uvedeme, že tato práce se pohybuje výhradně na úrovni pohledu člověka, neboť právě virtuální simulace pohledu člověka na vegetaci je jejím předmětem. Samozřejmě je možné navrhnout i jiné systémy, které



Obrázek 2.1: Znázornění příkladu úrovní detailu.

zpřístupní uživatelům informace na ostatních úrovních detailu, ale to je již úloha z oboru vizualizace a řada postupů by byla zřejmě zcela odlišná. Souvislost tohoto přirozeného systému úrovní detailu s již zmíněným technickým pojetím ale existuje. Jak bude popsáno dále, v současné době není v možnostech běžného hardwaru zobrazovat jednotným postupem lesy o tisícovkách stromů i detail jednotlivých listů samostatného stromu dostatečnou rychlosí pro real-time aplikace. Přirozený systém úrovní detailu rovněž napovídá, jak uživatel vnímá určité pohledy. Při pohledu na celý rozsáhlý les stromů jsou tím nejpodrobnějším, co uživatel rozpozná, jednotlivé stromy, případně nejvýraznější větve, největší pozornost je zaměřena na celkový tvar lesa, tvar čáry horizontu, hustotu a druhovou skladbu jednotlivých rostlin. Pohled na úrovni jednotlivého stromu naproti tomu musí zobrazit i jednotlivé větve a listy, tvar lesa již není tak důležitou informací. Pokud přejdeme ještě blíž, uživatel by měl mít možnost pozorovat i detaily na listech. Z toho může vycházet technická stránka úrovně detailu (dále jen LOD).

Cílem této práce je navrhnout a poté realizovat software, který bude realisticky zobrazovat stromovou vegetaci v reálném čase a umožní i její animaci (působením větru). Software by měl pracovat na běžně dostupném hardware. Dále bude software poskytovat podporu pro zobrazování stromů v různých obdobích roku. Půjde především o změnu barvy listí. Práce se bude pohybovat na úrovni podrobnosti běžné pro vnímání člověka. Tedy umožní zobrazit větší skupinu stromů (les) a přitom bude možné pozorovat jednotlivé listy stromů a to včetně animace. Práce se zaměří především na způsob, jakým lze stromy na těchto úrovních podrobnosti zobrazovat v pohybu.

## 2.2 Stávající řešení

Zobrazování vegetace v reálném čase je vcelku běžný požadavek současných grafických systémů - především herních enginů. Jak ukazuje například článek [17], existuje množství různých přístupů. Mohou být veskrze rozdeleny podle toho, zda pracují přímo s prostorovou geometrií modelu (3D reprezentace), či s dvourozměrnou náhražkou (2D reprezentace). Každý přístup má svá pro i proti a existuje i mnoho způsobů, jak je kombinovat.

### 2.2.1 Trojrozměrná geometrická reprezentace

Základní verze 3D povrchové reprezentace modelu vychází z myšlenky, že pokud je těleso neprůhledné (neprůsvitné), stačí k jeho zobrazení vykreslit jen jeho povrch. Povrch tělesa je popsán množinou trojúhelníků v prostoru a jejich barvou (texturou) - případně dalšími atributy. Trojúhelník reprezentuje část geometrie skutečného modelu. Avšak je nutné si uvědomit, že možnosti současných grafických karet jsou omezené a jejich výkon pro real-time (zhruba 25 fps) se pohybuje pouze v řádech desítek milionů zpracovaných trojúhelníků (při základní pipeline). Tato hranice se ještě výrazně sníží, pokud požadujeme pokročilejší zpracování (například složitější model osvětlení, více texturovacích zdrojů, atd.). Z výše uvedených omezujících faktorů vyplývá, že přímé zobrazení např. celého lesa na úrovni vykreslování trojúhelníků, aby geometrie reprezentující jednotlivé listy, je příliš náročné i pro současný hardware. Zajímavé metody a přístupy, jak zobrazovat kvalitně a efektivně i detailní geometrii stromů včetně animace až na úrovni jednotlivých listů jsou popsány v článcích [13] a [12]. Jsou navrženy postupy, které berou v potaz rozdílné vlastnosti různých stran listů i jejich

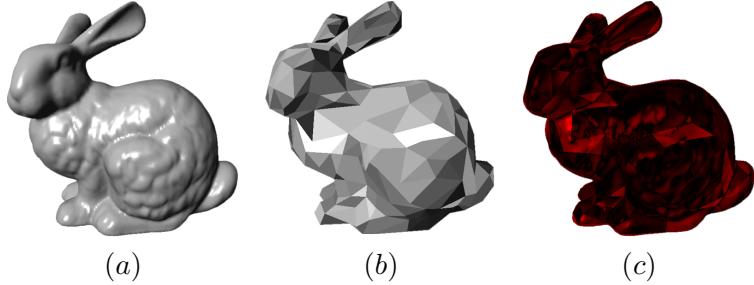


Obrázek 2.2: Možnosti realistického zobrazování vegetace, převzato z [12]

průsvitnost. Animace stromu je optimalizována pro zpracování na GPU ve vertex shaderu a pracuje na principu hierarchické deformace geometrie na úrovni jednotlivých hierarchických úrovní větví. Avšak zobrazení většího počtu stromů je i tak příliš náročné na výkon a není zde řešeno.

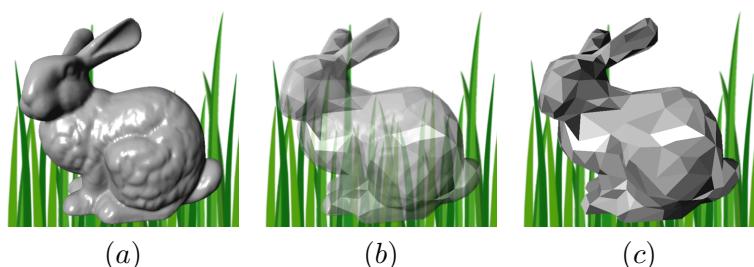
Existuje však i možnost, jak zobrazení optimalizovat. K tomu účelu se hojně využívá technik zjednodušování geometrie 3D objektu. Složitost geometrie přechází ve složitost materiálu. Pokud zůstaneme u tématu vegetace, pak si lze představit, že původní 3D reprezentace

např. stromu obsahuje geometrický popis i nejmenších detailů křivosti jednotlivých listů. V tom případě by stačila jen informace o barvě geometrických primitiv a výsledný obrázek po vykreslení by mohl být velmi kvalitní. Učíme myšlenkový krok zjednodušení geometrie tak, že jednotlivý list reprezentuje pouze jeden trojúhelník. Křivost samotného listu pak musí obsáhnout složitější informace o materiálu listu. Pokud bychom reprezentovali např. celou větev se všemi listy jedním trojúhelníkem, složitost materiálu musí zákonitě narůst příslušným způsobem, pokud požadujeme alespoň přibližně stejný výsledek zobrazení.



Obrázek 2.3: Obrazy různých stupňů detailu téhož modelu (a) a (b) a jejich rozdíl červeně (c)

Pokud metoda zobrazování a řízení úrovně detailu nepracuje dynamicky (což většinou zajišťuje takřka nepostřehnutelný přechod mezi úrovněmi), vyvstává vždy problém, jak provádět přechod mezi jednotlivými úrovněmi, který při naivním přepnutí úrovní může vést k nevítanému efektu zvanému *LOD-popping*. Jde o to, že pokud se skokově změní velká část obrazu, vnímá tuto změnu pozorovatel velmi rušivě. Klasická metoda snížení LOD-poppingu pracuje s prolínáním obrazů a řízením jejich průhlednosti. Zatímco jedna úroveň je postupně zprůhledňována a mizí, druhá se naopak stává neprůhlednou a objevuje se. Během tohoto přechodu se ovšem může stát, že se model stane průhledným, jak ukazuje obrázek 2.4. Eliminací tohoto nežádoucího efektu se zabývá článek [10]. Jde o postup, kdy je vždy jedna



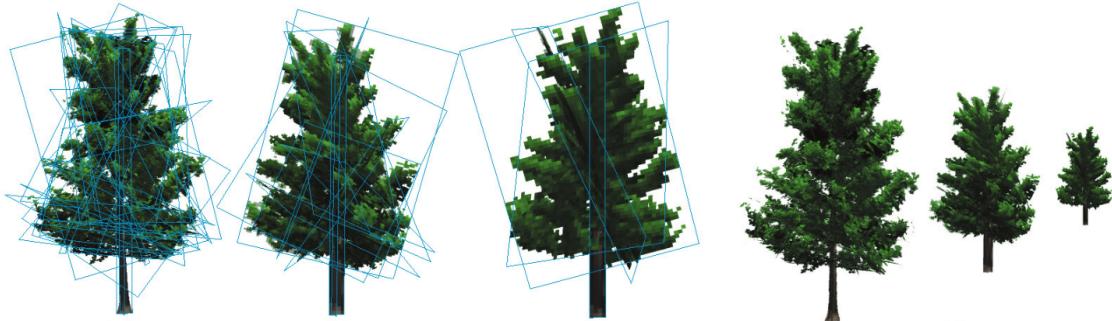
Obrázek 2.4: Znázornění problémů měkkého přechodu mezi diskrétními úrovněmi detailu.

- (a) LOD0 100% , LOD1 0%
- (b) LOD0 50% , LOD1 50%, objekt se stal průhledným, ačkoliv by neměl.
- (c) LOD0 0% , LOD1 100%

úroveň detailu zobrazena plně a neprůhledně. Tím je zaručeno, že nemůže dojít k celkovému zprůhlednění.

### 2.2.2 Reprezentace založené na obrázcích

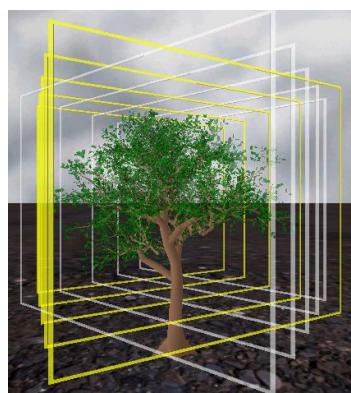
Způsob, jak optimalizovat proces zobrazování pomocí seskupení podobně orientovaných listů z celé koruny stromu do billboardů je popsán v článku [8]. Vegetace je pak reprezentována tzv. billboardcloudem. Bohužel, ačkoliv je tento přístup velmi slibný pro statické modely, pokud požadujeme věrohodnou animaci vegetace, pak nelze předpokládat, že všechny obdobně orientované listy na různých větvích se budou chovat stejným způsobem. Myšlenka seskupení



Obrázek 2.5: Různé úrovně detailu tvořené billboardy a jejich porovnání při použití (zcela vpravo), převzato z [7]

blízkých listů za účelem zjednodušení geometrické složitosti modelu je vyjádřena v článku [20], kde jsou listy dynamicky slučovány na základě jejich prostorové blízkosti a podobnosti. Ačkoliv jsou navrhované postupy daleko vhodnější pro přidání možnosti animace, jde v zásadě o dynamické řízení úrovně detailu a jako takové vyžadují relativně výrazně prostředky pro správu zjednodušovaného modelu.

Existuje i několik přístupů založených čistě na billboardingu. Metoda řezů představená v článku [15] vytváří určitý trs billboardů, které zobrazují pohledy z několika směrů a to i v několika vrstvách (viz obr. 2.6). Trs billboardů je v tomto případě vůči scéně nehybný a



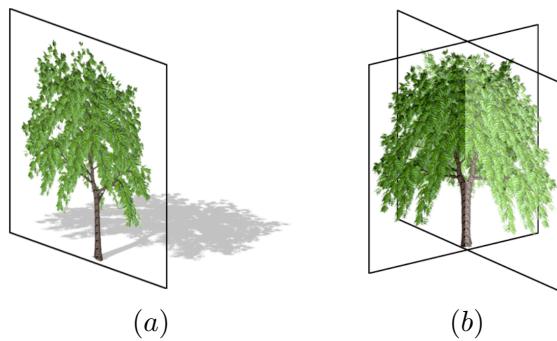
Obrázek 2.6: Reprezentace korun stromů pomocí řezů, převzato z [15]

řezy, které jsou souběžné se směrem pohledu, jsou skrývány. Výhoda použití více souběžných

billboardů spočívá v tom, že lze tímto způsobem navodit dojem určité prostorové hloubky (efekt paralaxy).

Metoda trsu rovnoběžných billboardů je rozpracována i v článku [21], kde je popsána možnost, jak efektivně řídit úrovně detailu pro tyto trsy.

Dovedeme-li zjednodušení do takové krajnosti, že původní objekt (či dokonce skupiny objektů) reprezentujeme jedním geometrickým primitivem, mluvíme o takzvaných billboardech, či impostorech (viz obr. 2.7). Díky vlastnostem lidského vnímání a zobrazovací technologie je možné nahradit určitý objekt plochým obrazem, který odpovídá pohledu na daný objekt z podobného místa. Zatímco zpracování geometrických primitiv je relativně pomalý proces,



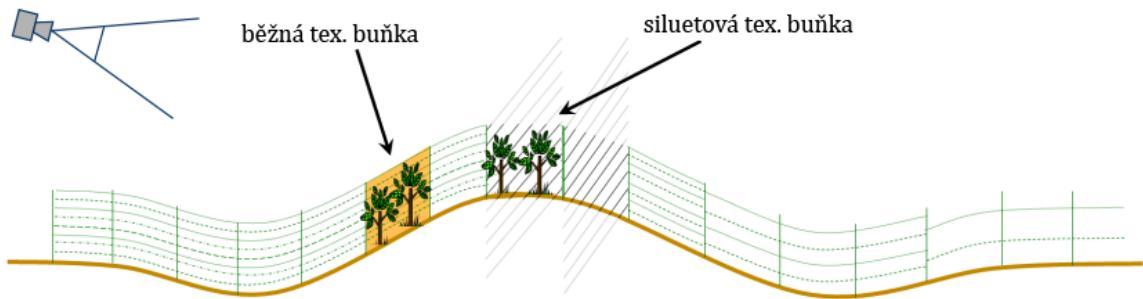
Obrázek 2.7: Různé přístupy k billboardům. Jediný plochý obraz, který se natáčí k pozorovateli (a), křížová konstrukce dvou billboardů, které zachovávají orientaci vůči scéně (b), převzato z [23]

uplatnění informací o materiálu v daném bodě může být velmi efektivní. Nutno ovšem dodat, že běžné reprezentace materiálu (obrázkové textury) trpí omezeními plynoucími z jejich principu – např. omezené rozlišení. Tedy jakési plnohodnotné nahrazení původní 3D geometrie funguje obstantně jen pro objekty ležící za určitou rozlišovací hranicí, která je dána zejména faktory jako rozlišení textur, rozlišení obrazovky, detailnost pohledu, geometrická podobnost obou reprezentací a podobně.

### 2.2.3 Volumetrické zobrazování a bodové mraky

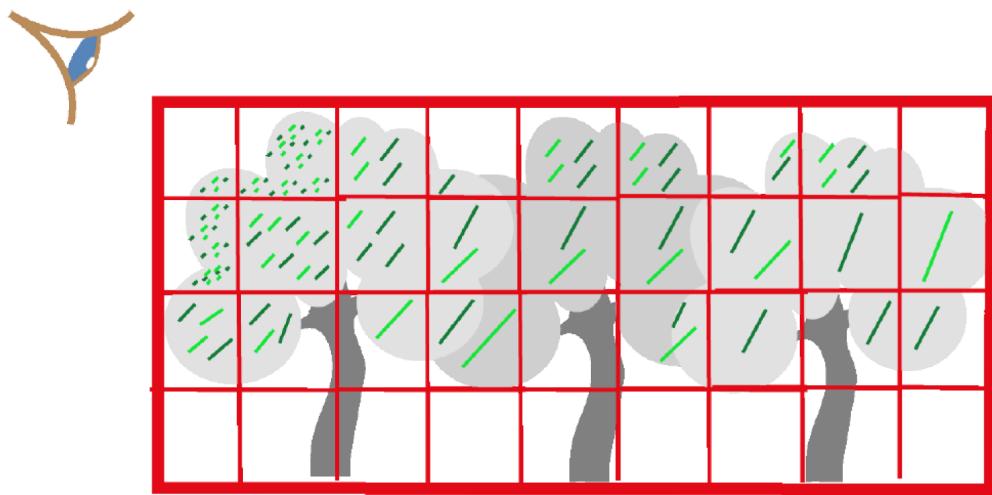
Existují však ještě další možnosti, jak reprezentovat a zobrazovat vegetaci. Metoda vycházející z principů volumetrického zobrazování (volume rendering) je popsána v článku [6]. Je vhodná zejména pro úroveň detailu typu les a vegetační geografie. Nicméně její přímé využití pro zobrazování na úrovni jednotlivých listů je s dnešním běžným hardware nemožné. Metoda pracuje s rovnoběžnými vrstvami textur, jejichž hustota je řízena aktuální úrovní detailu v daném místě.

Další možností je tzv. bodové (nebo-li *point-based*) zobrazování. Základním primitivem tohoto přístupu je prostorový bod (dále jen bod). Tato metoda je výhodná zejména pro objekty, které jsou promítnuty do malého počtu pixelů na výstupním zařízení, neboť pro jejich věrné zobrazení stačí minimálně tolik bodů, kolik pixelů na zobrazovacím zařízení zabírají. Ovšem neplatí zde, že prostorový bod je vždy promítnut do právě jednoho pixelu. Velikost a tvar, jímž je reprezentován bod, se mohou lišit. Nabízí se tedy, aby list stromu byl



Obrázek 2.8: Volumetrické zobrazování využívající rovnoběžných textur, převzato z [6]

reprezentován právě jedním bodem. Ovšem i tehdy by byl počet zpracovávaných bodů příliš veliký při zobrazování celého lesa naivním postupem. Pokročilé techniky LOD jsou navrženy v článku [11]. Prostor je rozdělen do pravidelné mřížky a v rámci vzniklých buněk je určen LOD. Na nejvyšší úrovni LOD reprezentuje skutečně každý bod specifický list, ovšem s nižší úrovni LOD jsou listy slučovány a postupně reprezentovány většími body (viz obr. 2.9).



Obrázek 2.9: Reprezentace listů jednotlivými body. Čím je zobrazovaný bod dále od kamery, tím je větší a reprezentuje více listů. Převzato z [11]

Vzdálenější koruny stromů jsou tak reprezentovány malým počtem velkých bodových primitiv, protože platí předpoklad, že tyto primitiva se zobrazí do jednoho (maximálně několika málo) výsledných pixelů. Bodová reprezentace v tomto podání funguje velmi dobře jen pro korunu stromu. Použití i pro samotný kmen a větve stromu je problematické.

#### **2.2.4 Zhodnocení stávajících řešení**

Jak se ukazuje, všechna řešení pracující až na úrovni jednotlivých listů pracují s geometrickým trojrozměrným modelem stromu (povrchová trojúhelníková reprezentace). Každá z metod zobrazující současně strom na úrovni listů i rozsáhlý les využívá nějakou formu řízení úrovňě detailu. Tím dosahují snížení složitosti scény a vyšší efektivity. Zatímco metod zaměřujících se pouze na statické zobrazení je celá řada, postupů, jak jednotlivé stromy realisticky rozhýbat je málo a omezují se na geometrické modely. Některé techniky LOD by bylo obtížné přizpůsobit, aby efektivně fungovaly i pro pohybující se stromy (volumetrické reprezentace). Jiné by bylo možné s relativně rozumným úsilím rozšířit i o tuto možnost (billboard-clouds). Metody dynamického řízení úrovňě detailu většinou představují elegantní řešení, které netrpí problémy LOD-poppingu, ale vyžaduje zvýšené prostředky na neustálé udržování aktuální úrovňě detailu modelů. Naproti tomu diskrétní LOD systémy se musí vypořádávat s přepínáním jednotlivých úrovní, ale představují jen malou výkonostní zátěž. Ať už dynamické či diskrétní LOD systémy, nejčastěji se využívá převedení složitosti geometrie do obrazu, který je pak namapován na zjednodušenou geometrii.

# Kapitola 3

## Analýza a návrh řešení

Pro účely real-time renderování grafiky se běžně využívá metod přímé rasterizace<sup>1</sup>. Existuje několik grafických API, které pro tyto účely využívají grafický hardware a dosahují tak pozoruhodného výkonu. Dvěmi nejběžnějšími API pro práci s 3D grafikou jsou v současnosti DirectX a OpenGL. Technologie DirectX je standardem v oblasti počítačových her a je vázaná na platformu Windows. Naproti tomu OpenGL je platformově nezávislá knihovna dosahující srovnatelných výsledků. Obě technologie pak pracují s GPU a využívají tzv. grafickou pipeline poskytovanou hardwarem. Z průzkumu stávajících řešení vyplývá, že realistického real-time zobrazení na úrovni listů lze v současné době dosáhnout efektivně využitím geometrické ploškové reprezentace. Pro její zpracování je optimalizován grafický hardware a takto popsané stromy lze i věrně rozpohybovat. Jeví se proto jako výhodné, využít ji pro zobrazování instancí stromů s nejvyšším stupněm detailu. Postup zpracování grafických primitiv na GPU je znázorněn na obrázku (3.1).

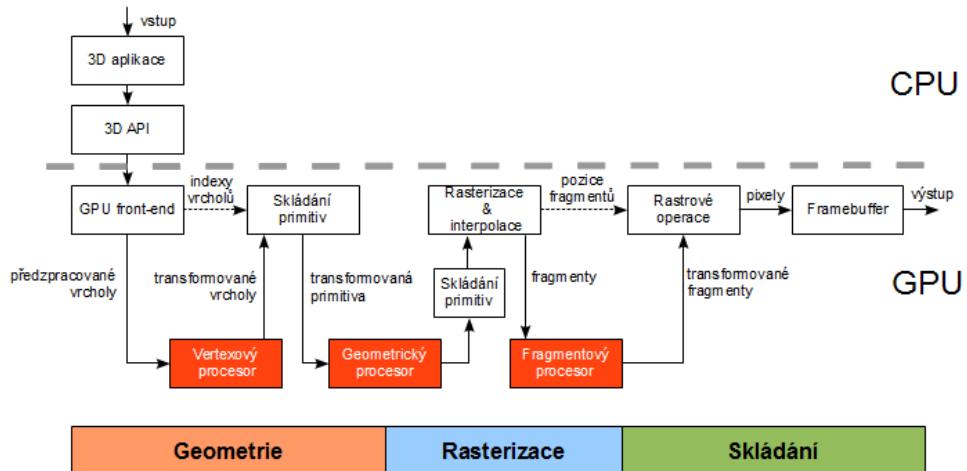
Nížší úrovně následně vytvořeného LOD-systému budou z této nejpodrobnější úrovně odvozeny a abstrahovány. Pro věrné zobrazení stromu pohybujícího se vlivem větru je třeba poznat, jak na vítr strom reaguje. Teoretické základy takové animace rozebere právě následující sekce.

### 3.1 Animace a vykreslování geometrického 3D modelu

Jelikož je model tvořen množinou trojúhelníků a ty mají přímou vazbu na části stromu (list, konkrétní větev), lze tedy vhodnou změnou jejich polohy animovat celý strom. Požadavek na provádění animace v reálném čase prakticky implikuje i potřebu využití GPU pro tyto účely. Grafická primitiva tvořící model jsou zpracovávána na GPU po vrcholech (vertexech) ve vertex shaderu, který může měnit jejich pozici. Metoda, již lze s výhodou použít, je popsaná v již zmíněném článku [13] a využívá myšlenky tzv. hierarchical vertex displacement. Důležitý je zde fakt, že skutečné větve stromů tvoří hierarchickou strukturu a transformaci nadřazené větve v bodě napojení přejímá celá skupina větví podřízených, což tvoří netriviální řetěz transformací. Tato skutečnost svazuje do určité míry výpočet polohy nadřazené a z ní

---

<sup>1</sup>V současnosti existují i přístupy využívající technik sledování paprsku (ray tracing) pracující v real-time. Pro některé scény mohou být dokonce efektivnější, nicméně jejich implementace na GPU je obtížnější a standardem v oboru real-time počítačové grafiky je stále přímá rasterizace.

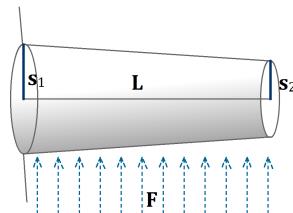


Obrázek 3.1: Grafická pipeline. Oranžově zvýrazněné bloky lze přeprogramovat a využít k vlastnímu zpracování grafiky.

vycházející větve. Poskytneme-li však vhodná data každému vrcholu, může být celý řetěz transformací určen právě pro každý jednotlivý vrchol korektně a výpočet animace se tak provádí na GPU ve vertex shaderu.

### 3.1.1 Deformace větví

Deformace tak složitých objektů, jako jsou stromy, vlivem větru je komplexní fyzikální problém. Na jeho složitosti mají zásadní vliv zejména dva faktory – netriviální mechanické vazby jednotlivých větví a turbulentní proudění vzduchu, které způsobuje výsledný pohyb. Oba faktory se navíc vzájemně ovlivňují. Aktuální pozice dané větve ovlivňuje proudění v okolí a opačně. I pokud bychom nahradili turbulentní proudění laminárním, jde o složitý problém. Samotný ohýb jedné větve lze sice relativně dobře popsat, ovšem musíme uvažovat, že každá větev je napojena na další a tím ovlivňuje její ohyb. Síly působící na listy i větve samotné vlivem proudění vzduchu se přenáší hierarchií větví až ke kmeni. Jde vlastně o složitý problém inverzní kinematiky, který pro takto složité struktury zatím nelze běžně řešit v real-time.



Obrázek 3.2: Fyzikální model pružného prutu (větve).

Aby bylo možné dosáhnout dostatečného zobrazovacího výkonu, je nutné rezignovat na úplnou fyzikální korektnost výpočtů deformace. Tuto relaxaci problému si naštěstí můžeme dovolit, neboť systém nemá ambice být fyzikálním simulátorem a předpokládá se, že uživatel se spokojí s výsledkem, který nebude výrazně porušovat jeho představu o deformaci vegetace. Aby byla animace pohybu větví co nejvěrnější, je využita deformace vycházející z Euler-Bernoulliho popisu ohybu prutu. Větev je approximována komolým kuželem s definovanou délkou  $L$  a poloměry na počátku  $s_1$  a na konci  $s_2$ . Tuhota tělesa je závislá pouze na průřezu. A na těleso působí kolmo síla  $\vec{F}$  (viz obr. 3.2 ).

Euler-Bernoulliho rovnice pak popisuje ohyb takového tělesa:

$$\frac{d^2}{dx^2}(EI(x)\frac{d^2\dot{u}(x)}{dx^2}) = F \quad (3.1)$$

V tomto vztahu představuje  $I$  plošný moment setrvačnosti a  $E$  je modul pružnosti, který je uvažován jako konstantní. Okrajové podmínky na počátku (pevný konec) jsou :

$$\dot{u} |_{x=0} = 0 \quad \frac{d\dot{u}}{dx} |_{x=0} = 0 \quad (3.2)$$

a na konci jsou:

$$\frac{d^2\dot{u}}{dx^2} |_{x=L} = 0 \quad \frac{d^3\dot{u}}{dx^3} |_{x=L} = 0 \quad (3.3)$$

Provedeme zjednodušení, kde normalizujeme rozdíl poloměrů  $r_1$  a  $r_2$  délkou  $L$  a zavedeme konstantu  $\alpha$ , která udává poměr velikosti poloměrů na začátku a na konci prutu, také poupravíme modul pružnosti  $E$  :

$$r_{1,2} = \frac{s_{1,2}}{L} \quad \alpha = \frac{r_2}{r_1} \quad E' = EL \quad (3.4)$$

Plošný moment setrvačnosti pro kruhový průřez poloměru  $r$  je dán vztahem:

$$I = \frac{\pi r^4}{4} \quad (3.5)$$

Protože uvažujeme, že poloměr prutu se mění lineárně v podélné ose prutu, plošný moment setrvačnosti  $I$  se mění takto:

$$I(x) = \frac{\pi r_1^4((\alpha - 1)x + 1)^4}{4} \quad (3.6)$$

Euler-Bernoulliho rovnice pro zmíněné okrajové podmínky a měnící se plošný moment setrvačnosti  $I(x)$  má netriviální řešení

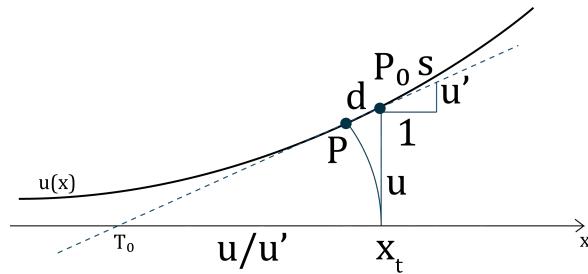
$$\begin{aligned} \dot{u}(x) &= \frac{E'F}{r_1^4} (x(\alpha - 1)(6 + x(\alpha - 1)(2x(\alpha - 1)(3 + (\alpha - 3)\alpha) + 3(4 + (\alpha - 2)\alpha))) \\ &\quad - 6(1 + x(\alpha - 1))^2 \log(1 + x(\alpha - 1)) \cdot (3\pi(1 + x(\alpha - 1))^2(\alpha - 1)^4)^{-1} \end{aligned} \quad (3.7)$$

Tento vztah je však příliš složitý pro efektivní a rychlé vyhodnocení. Důležitý je postřeh, že funkce  $\dot{u}$  je lineárně závislá na velikosti síly  $|\vec{F}|$ . Fitováním nalezneme funkci s velmi podobným průběhem ve tvaru:

$$\mathbf{u}(\mathbf{x}) = \mathbf{c}_2 \mathbf{x}^2 + \mathbf{c}_4 \mathbf{x}^4 \quad (3.8)$$

Koeficienty  $c_2$  a  $c_4$  reprezentují hodnoty  $E'$ ,  $\alpha$  a  $r_1$ . Ohybová funkce  $u(x)$  tedy určuje, jaká bude výchylka bodu kolmo na podélnou osu prutu ve vzdálenosti  $x$  od počátku větve (předchozí normování zajišťuje, že větev zde má vždy jednotkovou délku). Vytváří tak ohybovou křivku. Derivaci ohybové funkce v bodě  $x$  označíme  $u'(x)$ .

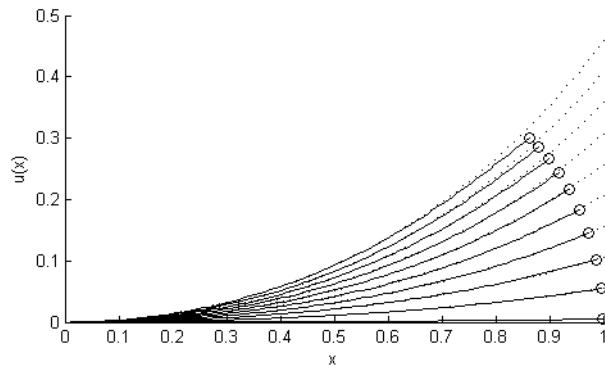
Ovšem bod na ohýbaném prutu se ve skutečnosti nepohybuje po přímce kolmé k podélné ose, nýbrž opisuje složitou křivku danou i faktum, že délka prutu se při deformaci nemění. Pokud by však byl použit přímo tento vztah pro deformaci, docházelo by k viditelnému efektu prodlužování prutu (větve) s většími výchylkami. Z toho důvodu je nutné zavést určitou korekci, jež bude ve výsledku zachovávat stejnou délku prutu pro libovolnou výchylku. Pokud bychom chtěli zachovat přesnou délku, bylo by nutné vždy vyřešit křivkový integrál, což je příliš časově náročné, a proto se spokojíme s nepřesnou, avšak dostačující a především rychlou metodou. Vyjdeme z předpokladu, že pro korekci délky stačí posunout bod  $P_0$  o určitou vzdálenost  $d$  po přímce tečné v tomto bodě s ohybovou křivkou. Získáme tak výsledný bod  $P$ .



Obrázek 3.3: Korekce délky ohybové funkce  $u(x)$ . Pro přehlednost byl ze zápisu vypuštěn parametr  $x_t$ . Místo  $u(x_t)$  je zapsáno pouze  $u$  a obdobně pro další.

Uvažujeme-li o deformaci jako o rotaci, pak na základě podobnosti trojúhelníků a zachování délek  $|T_0x_t|$  a  $|T_0P|$  můžeme zformulovat následující korekční vztahy (postup nazývejme „korekce délky“):

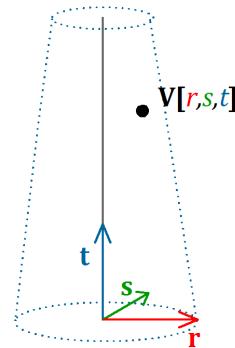
$$\begin{aligned} s(x) &= \sqrt{1 + u'^2(x)} \\ d(x) &= \frac{u(x)}{u'(x)}(s(x) - 1) \\ \vec{p} &= \vec{p}_0 + \frac{1}{s(x)} \begin{pmatrix} -d(x) \\ u(x) \end{pmatrix} \end{aligned} \quad (3.9)$$



Obrázek 3.4: Původní ohybová funkce  $u(x)$  pro různé výchylky (tečkovaně) a ohybová funkce po provedení korekce (plná čára). Kolečka vyznačují koncové body korigované křivky.

Jak je patrné z obrázku 3.4, chyba způsobená popsanou metodou je relativně malá a nemění charakter původní ohybové křivky.

Výše popsaným způsobem získáme však ohybovou funkci pracující ve 2D. Větev je ale třeba deformovat ve 3D. Za tímto účelem jsou aplikovány dvě výše popsané deformace v navzájem kolmých osách  $(r,s)$ .



Obrázek 3.5: Souřadný systém větve tvořený orthonormálními bázovými vektory  $\vec{r}$ ,  $\vec{s}$  a  $\vec{t}$ .

Jak vyplývá z Euler-Bernoulliho rovnice, síla ohýbající prut má lineární účinek na ohybovou funkci – stačí tedy funkci  $u(x)$  vynásobit silou  $\vec{F}$ , ohybová funkce závislá na síle působící na větev může být vyjádřena jako:

$$f(x) = |\vec{F}| \cdot u(x) \quad (3.10)$$

Pokud připustíme, že síla určující výchylku je určitým řídícím signálem  $A$ , pak bude celková deformace řízena vztahem (indexy udávají, k jaké ose se daná funkce vztahuje):

$$u_{r,s}(x) = A_{r,s}u(x) \quad u'_{r,s}(x) = A_{r,s} \frac{u'(x)}{L} \quad (3.11)$$

Konečný tvar ohybové funkce pracující ve 3D je tudíž:

$$P = P_0 + \begin{pmatrix} -d_r(x)/s_r(x) - d_s(x)/s_s(x) \\ u_r(x)/s_r(x) \\ u_s(x)/s_s(x) \end{pmatrix} \quad (3.12)$$

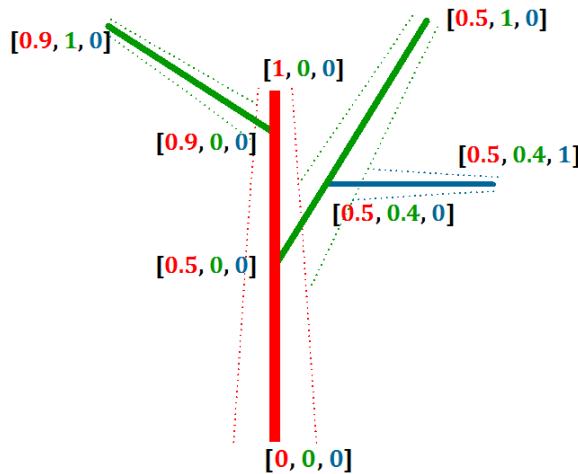
Pro korektní zobrazování celé hierarchie větví je potřeba ještě správně transformovat normálu a tangentu v daném bodě  $P$ . Aby bylo možné korektně vyjádřit potřebné vektory, je nutné znát Jakobián  $J_t$  popsané transformace. Bohužel právě jeho výpočet nelze efektivně provádět v real-time. Místo toho využijeme mnohem jednodušší Jakobián  $J_u$  původní ohybové funkce (bez délkové korekce) a vyhodnotíme ho pro délkově korigovaný bod  $P$ .

$$J_u = \begin{pmatrix} 1 & 0 & 0 \\ u'_r(x - d_r(x)/s_r(x)) & 1 & 0 \\ u'_s(x - d_s(x)/s_s(x)) & 0 & 1 \end{pmatrix} \quad (3.13)$$

Tangentu tedy vypočteme jako  $\vec{t} = \text{norm}(J_u \vec{t}_0)$ , normálu pak jako  $\vec{n} = \text{norm}(J_u^{-T} \vec{n}_0)$

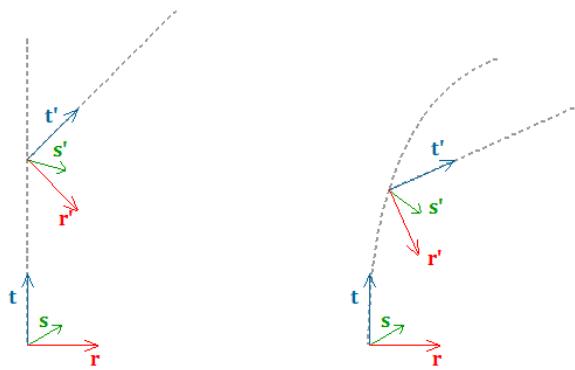
### 3.1.2 Hierarchická deformace

Předchozí kapitola popisuje, jak lze dosáhnout vcelku realistického ohybu osamocené větve. Je však třeba korektně propagovat deformaci do všech větví napojených. Pro rychlý a efektivní výpočet polohy každého vrcholu je nutné složit transformace vyplývající z topologie stromu (napojení větví a jejich ohyb). Pro každý vrchol je nutné mít přístupná data o relevantní části topologie, která ho ovlivňuje. K tomu stačí větve oddělují ve smyslu parametru  $x$  ohybové funkce (normovaná podélní vzdálenost na větvi). Každému vrcholu zavedeme tedy vektor hodnot  $x$ , kde  $i$ -tá složka vektoru odpovídá  $i$ -té hodnotě  $x$  na cestě z kořene.



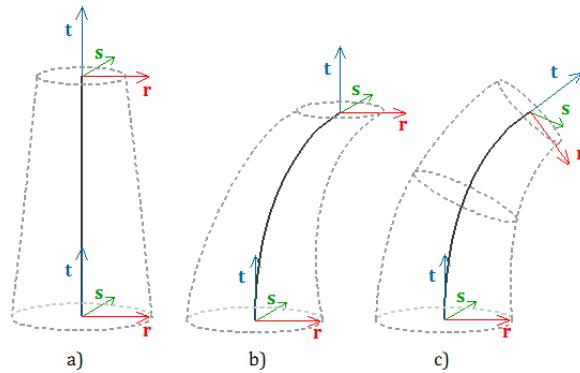
Obrázek 3.6: Vyjádření hierarchie pomocí vektoru s hodnotami  $x$ .

Dále musí mít vrchol přiřazeny souřadnice  $r, s, t$  v souřadné soustavě větve. Aby se při



Obrázek 3.7: Souřadný systém větve a jeho transformace při ohybu nadřazené větve

ohýbání větví nezplošťovala, je nutné provádět transformaci souřadného systému větve pro každý vrchol. Zde je možnost tuto korekci neprovádět a zrychlit tak výpočet na úkor kvality výsledného ohýbu.



Obrázek 3.8: Transformace souřadných systémů při ohybu. (a) výchozí poloha, (b) ohyb bez transformace souřadného systému, (c) ohyb se správnou transformací

Jak je patrné z obrázku 3.8, při malých deformacích je chyba v podstatě zanedbatelná. Při výpočtu polohy bodu  $\vec{p}_i$  na středovém paprsku větve je vhodné použít následujících iterativních vztahů:

$$\vec{p}_i = \vec{p}_0 - \frac{\vec{t}d_r(x) - \vec{r}u_r(x)}{s_r(x)} - \frac{\vec{t}d_s(x) - \vec{s}u_s(x)}{s_s(x)} \quad (3.14)$$

$$x_{i,r,s} = x - \frac{d_{r,s}(x)}{s_{r,s}(x)} \quad (3.15)$$

$$\vec{t}_i = \vec{t}_0 + (u'_r(x_{i,r})\vec{r} + u'_s(x_{i,s})\vec{s})(\vec{t} \cdot \vec{t}_0) \quad (3.14)$$

$$\vec{n}_i = \vec{n}_0 + (u'_r(x_{i,r})(\vec{r} \cdot \vec{n}_0) + u'_s(x_{i,s})(\vec{s} \cdot \vec{n}_0))\vec{t} \quad (3.15)$$

Pokud výpočet transformace polohy bodu bude probíhat od kořene hierarchie, dokud v

ní není dosaženo úrovně, na které se daný vrchol nachází, pak budou postupně aplikovány všechny relevantní deformace. Na počátku je  $\vec{p}_0$  původní poloha bodu v souřadném systému objektu (obdobně i tečna  $\vec{t}_0$  a normála  $\vec{n}_0$ ). V každé další iteraci (postupu o úroveň výš v hierarchii) je třeba počáteční vektory  $\vec{p}_0$ ,  $\vec{t}_0$  a  $\vec{n}_0$  nastavit na výsledek z předchozí iterace. Hodnota  $x_{D,r}$  a  $x_{D,s}$  představuje hodnotu parametru  $x$  po provedení korekce délky. Transformace souřadného systému v rámci jedné větve vyžaduje přepočet nového bázového vektoru  $\vec{t}$  podle vzorce pro výpočet tečny a nových bázových vektorů  $\vec{r}$  a  $\vec{s}$  podle vzorce pro výpočet normály.

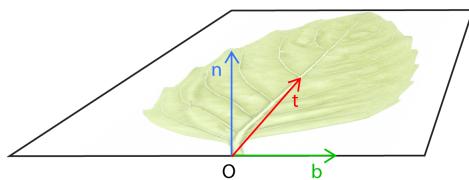
Následující tabulka shrnuje, jaká data jsou potřeba pro výpočet deformace:

pro vrchol	pro celou větve
skutečná poloha	koeficienty $c_2$ , $c_4$ , délka $L$
poloha vzhledem k větvi	definice souřadného systému větve
hodnota x na větvi	řídící signály ohýbu
odkaz na větvi	odkaz na nadřazenou větve

Tabulka 3.1: Data potřebná k výpočtu hierarchické deformace větve

### 3.1.3 Deformace listů

Listy jsou součástí hierarchie stromu a musí proto respektovat i příslušný řetěz deformací této hierarchie. Kromě toho ovšem přidávají i vlastní deformaci, která je svou podstatou odlišná od deformace větví. Uplatňuje se tu jak podélná deformace, tak krut, který nastavuje plochu listu do rovnovážné polohy vzhledem k mechanickým vlastnostem a působícímu větru. Jelikož považujeme listy za velice lehké a v rámci své plochy za nedeformovatelné, je tedy



Obrázek 3.9: Souřadný systém listu.

deformace listu approximována jako orientace plochy listu vůči věti, ze které vyrůstá. Jde tedy o relativně primitivní transformaci souřadného systému listu, který vytvoříme způsobem naznačeným na obrázku 3.9.

Deformace listu pak může probíhat takto:

$$\begin{aligned}\vec{t}_t &= \vec{t}_0 + \vec{n}_0 * A_x \\ \vec{n}_t &= \vec{t}_t \times \vec{b}_0 \\ \vec{n}_r &= \vec{n}_t + \vec{b}_0 * A_y \\ \vec{b}_r &= \vec{t}_t \times \vec{n}_r \\ \vec{t}_r &= \vec{t}_t\end{aligned}$$

$$P_d = O_{xyz} + P_x * \vec{b}_r + P_y * \vec{t}_r \quad (3.16)$$

Bod  $O$  je místo, kde se list napojuje na větev souřadnice  $P_x$  a  $P_y$  jsou dány relativně, vůči tomuto bodu v soustavě  $\vec{n}\vec{b}\vec{t}$ .

### 3.1.4 Řízení animace

Animace by měla uživateli navodit dojem, že se strom hýbe působením větru. Uvažujme, že celý pohyb je způsoben hledáním rovnovážné polohy, která závisí na mechanických vlastnostech stromu a vnějších sil (gravitace a vítr). Tím, že vnější síla v podobě působení větru je značně proměnlivá, dochází k neustálému neuspořádanému kývání kolem tušené rovnovážné polohy. Pro účely této práce dále uvažujme, že má vítr určitou složku směrového proudění a složku turbulentního proudění. Zmíněné nahodilé kývání zřejmě způsobuje turbulentní složka, zatímco složka směrového proudění má spíše za následek celkovou změnu rovnovážné polohy (pozorovatel vnímá, že se celý strom ohýbá).

Pozorujeme-li reálný strom, pak pro určité rozmezí malé síly větru nelze určit, kterým směrem vítr vane. V takovém případě tedy výrazně převládá vliv turbulentní složky. Lze však sledovat, že čím je daná větev větší, tím je frekvence těchto výkyvů nižší a amplituda naopak vyšší. Velikost větve odpovídá vcelku dobře její úrovni v hierarchii větví.

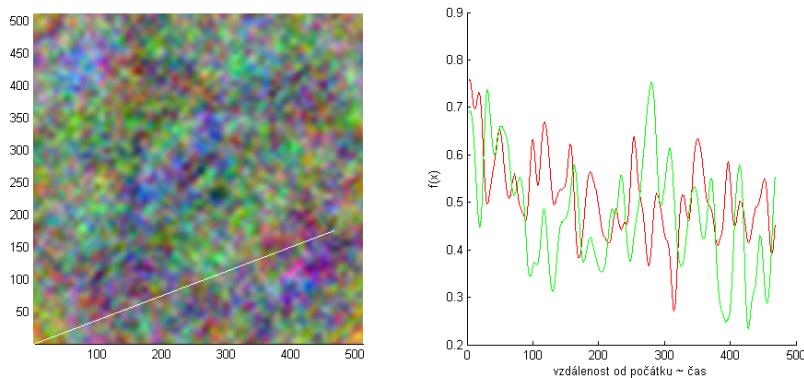
Naproti tomu silnější vítr způsobí celkové ohnutí stromu do směru, kam vane. Převládá složka směrového proudění. Frekvence výkyvů jednotlivých větví se zvyšuje. Do určité meze roste i amplituda. Po jejím překročení amplituda klesá. V případě extrémní síly větru je již nahodilý pohyb větví vlivem působících sil minimální.

Popis deformací z kapitoly Deformace větví umožňuje jednoduše zohlednit obě zmínované působící složky. Vyjdeme-li ze vztahu (3.10) a rozvineme sílu  $\vec{F}$  do tvaru:

$$F_d = |\vec{W}_{turb}| + \vec{d} \cdot \vec{W}_{lin} \quad (3.17)$$

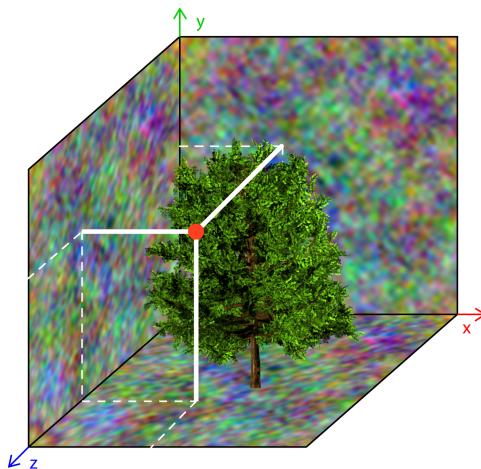
bude možné promítnout sílu směrové složky větru  $\vec{W}_{lin}$  a turbulentní složky  $\vec{W}_{turb}$  do směru  $\vec{d}$  resp. do směrů  $\vec{r}, \vec{s}$ . Aby bylo možné dosáhnout kýzeného efektu způsobeného turbulentní složkou, je třeba v čase měnit příslušným způsobem sílu  $\vec{W}_{turb}$ . Obě složky lze chápout jako řídící signály animace (směrová složka může zůstávat konstantní). Jsou-li tyto signály měněny plynule, je plynulá i výsledná animace.

Jednotlivé větve je však třeba řídit vzájemně nezávislými signály. Proto je nutné generovat pro každou větev dvousložkový šum, který bude ideálně aperiodický a pro větve unikátní. S přihlédnutím k možnostem grafického hardwaru se jako velmi výhodná jeví strategie využívající jedinou šumovou texturu pro generování celé třídy řídících šumových funkcí. Vyhodnocením vzorku dat na určité pozici získáváme vlastně funkční hodnotu. Bude-li se pozice v textuře plynutím času posouvat po přímce definované směrovým vektorem  $\vec{m}\vec{v}$  (popřípadě i počátečním bodem  $o$ ), získáme postupně funkční hodnoty průběhu požadované šumové funkce. Vzorek dat pak může obsahovat více kanálů (např. RGB, jak je u obrázků běžné) a získáme takto i více různých šumových funkcí současně, jak je patrné na obrázku 3.10 .



Obrázek 3.10: Dvě generované šumové funkce pro červený a zelený kanál (vlevo) získané odečítáním hodnot na bílé úsečce v šumové textuře (vpravo)

Pro pozorovatele jsou jednotlivé řídící funkce větví nezávislé – turbulentní složka je pro pozorovatele natolik chaotická, že nemůže lehce určit vazby. Naproti tomu pro listy, které svým chováním vlastně přímo vzorkují turbulentní proudění, je dobré dodržet určité prostorové vazby. Pozorovatel totiž dokáže rozeznat poryv větru docela dobře podle pohybu listů. Listy v blízkém okolí musí na poryv reagovat podobně. Z toho důvodu zavedeme zjednodušený model turbulentního pole (viz obrázek 3.11).



Obrázek 3.11: Model 3D turbulentního pole pro animaci listů

K tomu lze využít jedinou šumovou texturu. Pozici dotazu do turbulentního pole posuneme o vektor  $-\vec{W} \cdot t$ , kde  $t$  je čas. Získáme tak 3 hodnoty  $A_{xy, xz, yz}$ , ze kterých vypočteme

vážený součet.

$$A^l = A_{xy}^p \left(1 - \frac{|W_z|}{|\vec{W}|}\right) + A_{yz}^p \left(1 - \frac{|W_x|}{|\vec{W}|}\right) + A_{xz}^p \left(1 - \frac{|W_y|}{|\vec{W}|}\right) \quad (3.18)$$

Výsledkem je hodnota, která je největším dílem tvořena z roviny, jež nejlépe odpovídá směru větru.

## 3.2 Zobrazování listů

Listy rostlin představují z hlediska počítačové grafiky zajímavý objekt. Jejich vizuální vlastnosti se liší mezi jednotlivými druhy rostlin. Zároveň je často velmi markantní rozdíl mezi rubovou (horní) a lícovou (spodní) stranou. Zatímco z vrchu jsou některé listy vysoce lesklé díky různým voskovým vrstvičkám, zespodu jsou často spíše matné. Některé listy mají na povrchu miniaturní chloupek, které jim dávají hedvábný vzhled. Dalšími faktory jsou pak tvar, tloušťka a barva listu.



Obrázek 3.12: Fotografie skutečných listů, kde se projevuje jejich průsvitnost, zdroj [19]

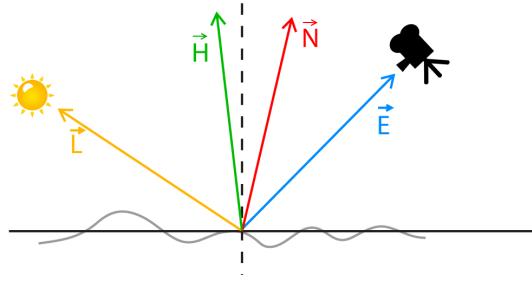
Metod a postupů, jak zobrazovat listy v real-time existuje několik. Většina přístupů se ale spokojuje s tím, že bere v potaz pouze povrchové vlastnosti listu a simuluje tak například členitost jeho povrchu či běžné odrazové vlastnosti. Metoda popsaná v [12] ovšem zahrnuje i průsvitnost listů, která hraje zásadní roli zejména na spodní straně listu (viz obr. 3.12). Osvětlení listu lze zformulovat do následujícího vztahu

$$L = L_D + L_I + L_E, \quad (3.19)$$

kde výsledné osvětlení  $L$  z jednoho světelného zdroje se skládá z příspěvku přímého ( $L_D$ ) a nepřímého ( $L_I$ ) osvětlení a složky  $L_E$ , kterou list emituje. V následujícím textu bude rozebrán pouze příspěvek přímého osvětlení. Zbylé budou approximovány ambientní složkou. Přímé osvětlení lze pak schematicky rozepsat jako:

$$L_D = L_{diffuse} + L_{specular} + L_{ambient} + L_{translucent}, \quad (3.20)$$

kde první tři členy představují příspěvek odraženého světla, zatímco poslední člen  $L_{translucent}$  se vztahuje k průsvitnosti listu.



Obrázek 3.13: Situace na povrchu listu. Vektor  $\vec{N}$  definuje normálu v bodě dopadu paprsku.  $\vec{E}$  udává směr k pozorovateli a  $\vec{L}$  ke světlu. Vektor  $\vec{H}$  je definován na ose úhlu mezi  $\vec{L}$  a  $\vec{E}$ . Šedivá křivka představuje povrch.

K vyjádření difuzní a spekulární složky využijeme popisu z obrázku 3.13. Difuzní složku lze určit jako příspěvek světla vážený úhlem mezi přicházejícím paprskem světla  $\vec{L}$  a normálou povrchu  $\vec{N}$  v místě jeho dopadu:

$$L_{\text{diffuse}} = \vec{N} \cdot \vec{L} \quad (3.21)$$

Výpočet spekulární složky je oproti tomu složitější. Nevyužívá Phongova osvětlovacího modelu, ale nahrazuje ho odvozeninou osvětlovacího modelu Cook-Torrance popsávaného v [4] a [3]. Tento model pracuje s BRDF jež operuje nad proměnnými  $\vec{N}$ ,  $\vec{L}$ ,  $\vec{E}$ ,  $\vec{H}$  (význam viz obr. 3.13),  $\sigma$  udávajícím drsnost povrchu a indexem lomu  $n$  světla na povrchu listu.

$$BRDF_{\text{spec}}(\vec{N}, \vec{L}, \vec{E}, \vec{H}, \sigma, n) = \frac{R(\sigma, \vec{N}, \vec{H}) \cdot F(n, \vec{H}, \vec{E}) \cdot G(\vec{N}, \vec{L}, \vec{E}, \vec{H})}{(\vec{N} \cdot \vec{L})(\vec{N} \cdot \vec{E})\pi} \quad (3.22)$$

Člen  $R(\sigma, \vec{N}, \vec{H})$  představuje vliv hrubosti povrchu a může být vyjádřen jako:

$$R(\sigma, \vec{N}, \vec{H}) = \frac{1}{\sigma^2 \cdot (\vec{N} \cdot \vec{H})^4} \cdot e^{\left( \frac{1}{\sigma^2} - 1 \right)} \quad (3.23)$$

Naproti tomu, člen  $F(n, \vec{H}, \vec{E})$  popisuje lesklou složku:

$$\begin{aligned} c &= \vec{H} \cdot \vec{E} \\ g &= \sqrt{n^2 + c^2 - 1} \\ F(n, \vec{H}, \vec{E}) &= \frac{1}{2} \left( \frac{g - c}{g + c} \right)^2 \left[ 1 + \left( \frac{c(g + c) - 1}{c(g - c) + 1} \right) \right] \end{aligned} \quad (3.24)$$

Konečně člen  $G(\vec{N}, \vec{L}, \vec{E}, \vec{H})$  popisuje geometrii v bodě dopadu paprsku:

$$G(\vec{N}, \vec{L}, \vec{E}, \vec{H}) = \min \left( 1, \frac{2 \cdot (\vec{N} \cdot \vec{H}) \cdot (\vec{N} \cdot \vec{E})}{(\vec{E} \cdot \vec{H})}, \frac{2 \cdot (\vec{N} \cdot \vec{H}) \cdot (\vec{N} \cdot \vec{L})}{(\vec{E} \cdot \vec{H})} \right) \quad (3.25)$$

Složka způsobená průsvitností listu má dominantní vliv, pokud se světelný zdroj nachází za rovinou listu vzhledem k pozorovateli. K jejímu vyjádření je použit BSSRDF model. Jeho konstrukcí a určením pro různé listy se venuje [12] a tato problematika zde nebude dále rozebírána. Zmiňovaná funkce  $L_t$  závisí na pozici  $\vec{x}_0$ , směru k pozorovateli  $\vec{E}$  a směru ke světlu  $\vec{L}$  a může být napsána v diskrétní formě takto:

$$L_t(\vec{x}_0, \vec{E}, \vec{L}) = \rho_t(\vec{x}_0, \vec{E}) A_p \sum_{\vec{x}_i} T(r, d(\vec{x}_i)) E(\vec{x}_i, \vec{L}), \quad (3.26)$$

kde  $\rho_t(\vec{x}_0, \vec{E})$  reprezentuje propustnost,  $A_p$  zastupuje plochu texelu,  $T(r, d(\vec{x}_i))$  je dynamické konvoluční jádro díky závislosti na tloušťce listu  $d(\vec{x}_i)$  a konečně  $E(\vec{x}_i, \vec{L})$  je přenosová funkce osvitu (irradiance transport function). Jde v zásadě o proces konvoluce obrazové informace a můžeme tedy odlišit konvoluční část.

$$L_t(\vec{x}_0, \vec{E}, \vec{L}) = \rho_t(\vec{x}_0, \vec{E}) L_t^C(\vec{x}_0, \vec{L}) \quad (3.27)$$

$$L_t^C(\vec{x}_0, \vec{L}) = A_p \sum_{\vec{x}_i} T(r, d(\vec{x}_i)) E(\vec{x}_i, \vec{L}), \quad (3.28)$$

Právě konvoluční část tvořená hemisférickou funkcí  $L_t^C$  je v real-time příliš obtížné vypočítat, a proto se předpočítá pro každý texel. Reprezentována pak je pomocí tzv. *Half Life 2* bází (HL2b). Jde o konstrukt, který umožňuje jednoduše vyjádřit reprezentovanou funkci pro daný hemisférický směr. Bázové vektory HL2b jsou následující:

$$\begin{aligned} \vec{H}_1 &= \left( -\frac{1}{\sqrt{6}}, -\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{3}} \right) \\ \vec{H}_2 &= \left( -\frac{1}{\sqrt{6}}, \frac{1}{\sqrt{2}}, \frac{1}{\sqrt{3}} \right) \\ \vec{H}_3 &= \left( \sqrt{\frac{2}{3}}, 0, \frac{1}{\sqrt{3}} \right) \end{aligned} \quad (3.29)$$

Tyto vektory pak definují tři kosínové bázové funkce na polokouli (hemisféře):

$$\mathcal{H}_i(\vec{d}) = \sqrt{\frac{3}{2\pi}} \vec{H}_i \cdot \vec{d} \quad (3.30)$$

Hemisférickou funkci  $f(\vec{d})$  pro směr  $\vec{d}$  lze tedy přepsat pomocí těchto bázových funkcí následovně:

$$f(\vec{d}) = \sum_{i=1\dots 3} h_i \mathcal{H}_i(\vec{d}), \quad (3.31)$$

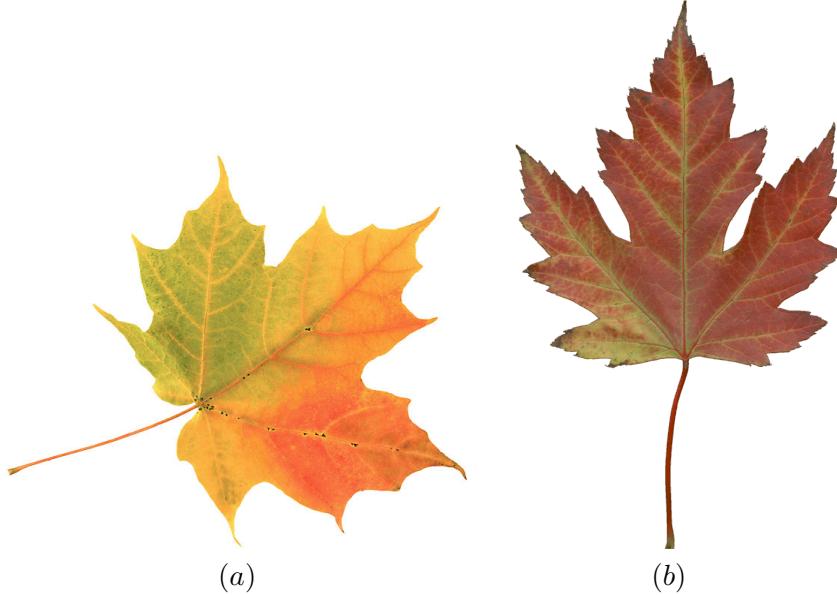
kde  $h_i$  jsou souřadnice vzhledem k bázovým funkcím.

Výsledný vztah rekonstruující původní  $L_t$  využívající HL2b pro předpočítanou konvoluční funkci pak vypadá takto:

$$L_t^R(\vec{x}_0, \vec{L}) = L_D \rho(\vec{x}_0) \sum_{i=1\dots 3} h_i(\vec{x}_0) \sqrt{\frac{3}{2\pi}} \vec{H}_i \cdot \vec{L} \quad (3.32)$$

Uvážíme-li, že  $L_D \rho(\vec{x}_0)$  a  $h_{1\dots 3}(\vec{x}_0)$  lze uložit do textur (pro pozici  $\vec{x}_0$ ), lze tím pádem průsvitnost vypočít na základě pouze dvou dotazů do textur.

Posledním krokem je přizpůsobit výše popsané postupy tak, aby bylo možné dynamicky měnit barvu listů. Barva se liší jednak v rámci jednoho listu, jednak mezi listy. Sezónní změny barvy listu se často projevují různě v různých částech listu. Zatímco okrajové části mohou být už červené či žluté, středové části mohou zůstávat stále zelené. Toto chování ovšem není cílem napodobit. Důraz bude kláden na celkovou barevnost listu, která hraje roli při pohledu z větší vzdálenosti.



Obrázek 3.14: Fotografie skutečných listů, barevné variace v ploše listu

Vyjdeme-li ze vztahu 3.20, a budeme-li ho chápat jako jednotlivé skalární podíly intenzit světla, můžeme vytvořit následující vztah pro zahrnutí barevnosti listu:

$$L_D = L_{\text{diffuse}} \mathcal{D} + L_{\text{specular}} \mathcal{S} + L_{\text{ambient}} \mathcal{A} + L_{\text{translucent}} \mathcal{T}, \quad (3.33)$$

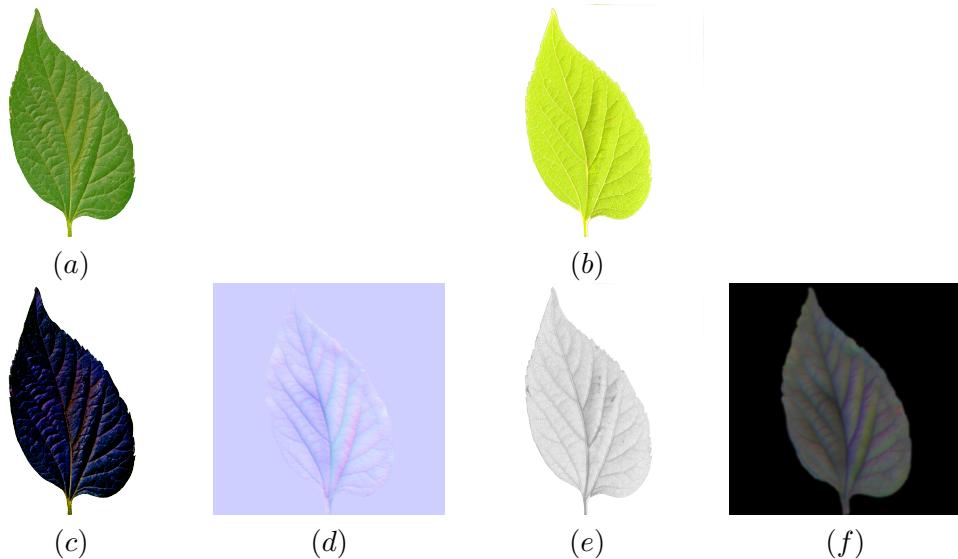
přičemž  $\mathcal{D}$  je difuzní barva v daném bodě,  $\mathcal{S}$  je barva odlesku daná barvou světelného zdroje a materiélem,  $\mathcal{A}$  je barva ambientního příspěvku světla a  $\mathcal{T}$  je barva vzniklá průchodem světla listem.

Pro konkrétní bod  $x$  a sezónu  $s$  lze barvy vyjádřit jako:

$$\begin{aligned} \mathcal{D}(x, s) &= (c_v(x) + c_s(s)) \cdot \mathcal{M}_{\text{diffuse}} \\ \mathcal{A}(x, s) &= (c_v(x) + c_s(s)) \cdot \mathcal{M}_{\text{ambient}} \\ \mathcal{S}(x, s) &= \mathcal{M}_{\text{specular}} \\ \mathcal{T}(x, s) &= (c_t(x) + c_s(s)) \cdot \mathcal{M}_{\text{translucent}}, \end{aligned} \quad (3.34)$$

kde  $\mathcal{M}_{(\dots)}$  je pro daný materiál na světelný zdroj konstanta,  $c_v(x)$  je prostá barva listu, která je definovaná texturou,  $c_s(x)$  je barevná odchylka daná sezónními změnami barvy (dovolíme

si zjednodušení: sezónní barevná odchylka je pro celý list stejná ). Člen  $c_t(x)$  vyjadřuje barvu po průchodu světla listem, která je rovněž definovaná texturou. Na tomto místě je dobré přiznat, že správnější by bylo uvažovat pro různé barevné variace listu dané sezónou i různé barvy průsvitné složky. Řešní by se tím ovšem zkomplikovalo a i popsaný přístup funguje relativně dobře. Předpokládá se, že každá ze dvou stran listu bude vyhodnocována odděleně s jinými parametry (různé textury, příp. materiály). Na obrázku 3.15 jsou zdrojové textury pro jednu stranu listu. Pro listy jsou ovšem k dispozici původně jiná zdrojová data.

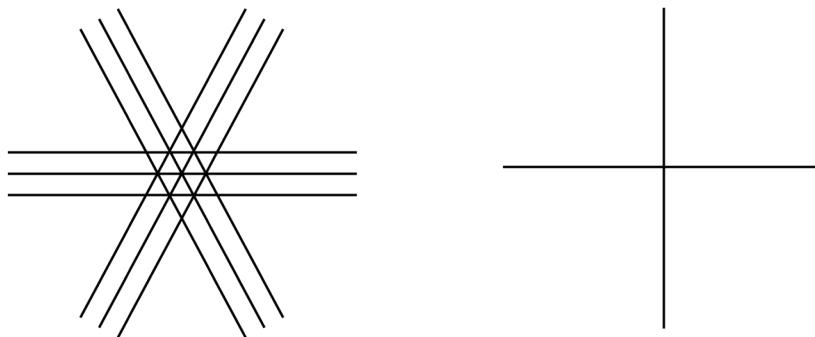


Obrázek 3.15: Zdrojové textury pro výpočet osvětlení listů. (a) původní barevná mapa, (b) původní mapa průsvitnosti, (c) mapa barevných odchylek od základní barvy listu (pro názornost zvětšeny), (d) normálová mapa, (e) mapa průsvitnosti, (f) mapa koeficientů Half Life 2 bázových funkcí

Původní barevná mapa obsahuje prosté barvy listu (např. fotografie) - jde vlastně o záznam podoby listu pro jednu sezónní barvu a tu neumožňuje přímo a jednoduše dynamicky měnit. Mapu barevných odchylek můžeme získat z původní barevné mapy odečtením základní barvy listu (později přičtena jako sezónní barva listu). Obdobné předzpracování je nutné provést s původní mapou průsvitnosti, na kterou je výhodnější nahlížet jako na mapu intenzit.

### 3.3 Úrovně detailu

V předchozích kapitolách předpokládáme zpracování a zobrazování geometrického modelu vegetace. Pro nižší úrovně detailu (LOD) lze využít metod, které složitou geometrii typicky nahrazují zobrazením primitivní geometrie (např. čtverce), na které je aplikována textura (obrázek) navozující dojem, že se jedná o původní objekt. Zaznamenáme-li pohledy na objekt z několika směrů, lze s určitou rozumnou tolerancí ke snížení kvality zobrazení pomocí těchto pohledů libovolný pohled na objekt bez nutnosti zobrazení a zpracování jeho plné geometrické reprezentace. Úspora potřebného výkonu spočívá jak v malém počtu zpracovávaných vrcholů geometrie, tak v ušetření řady rasterizačních operací stejně jako v menším počtu zpracovávaných fragmentů.<sup>2</sup> Výsledná kvalita zobrazeného objektu je ovlivněna počtem předgenerovaných pohledů, jejich kvalitou a také způsobem, jak zkonstruovat obraz pro pohled ze směru, pro který neexistuje předgenerovaný obraz. Známé jsou metody billboardingu využívající pro osově souměrnou geometrii jediného pohledu, který je natáčen kolmo k pohledu virtuální kamery.<sup>3</sup> Relativně běžná je metoda využívající jakýchkoli trsu billboardů. Objekt je nahrazen množinou různě orientovaných geometrických primitiv, jak je patrné z obrázku



Obrázek 3.16: Konstrukce jednoduchého trsu billboardů: (pohled zvrchu) vyšší LOD tvořený třemi skupinami billboardů po 3 řezech (vlevo), nižší LOD tvořený pouze dvěma kolmými billboardy (vpravo)

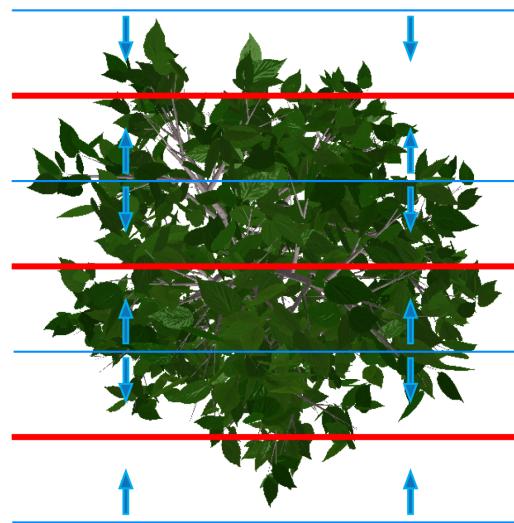
Na rozdíl od metod skutečného billboardingu, trsy billboardů si zachovávají svou orientaci vůči světovým souřadnicím ve scéně. Tento koncept lze vylepšit pro zobrazování objektů, jako jsou stromy tím, že přidáme další rovnoběžné billboardy. Struktura koruny stromu je dosti členitá a při pohybu kolem stromu se uplatňuje paralaxa a dochází k překrývání větví a listů. K tomu dochází i v případě pohybu samotného stromu. Budeme-li uvažovat v určitých

<sup>2</sup> předpokládá se, že jednotlivé listy jsou vykreslovány v náhodném pořadí a tím pádem je barva určitého pixelu několikrát přepisována

<sup>3</sup>existuje několik možností natáčení obrázku např:

- rovnoběžně se stínítkem kamery
- kolmo k pohledu kamery

částech průhledné obrázky použité v trsu rovnoběžných billboardů, pak lze podobných efektů docílit. Sadu rovnoběžných billboardů lze chápat jako různé řezy geometrií. Do každého řezu se promítne určité okolí tak, aby celkově všechny rovnoběžné řezy pokrývaly rovnoměrně celou původní geometrii. Řezy lze předgenerovat pro statický strom a používat je následně pro každou instanci daného stromu ve scéně. Předpokládá se rovněž, že v rámci jednoho řezu bude vytvořen soubor informací potřebný k pozdějšímu zobrazení a animaci (např. barevné, normálové a hloubkové mapy). Čím je objekt od pozorovatele vzdálenější, tím méně se tyto



Obrázek 3.17: Tvorba řezů pro vícevrstvé billboardy. Jednotlivé textury řezů jsou znázorněny červeně.

efekty uplatňují a tím menší má objekt percepční váhu, proto lze v rámci zvýšení výkonu snižovat počet obrázků (textur) v trsu. Protože se orientace trsu řezů vůči scéně nemění, mohou vznikat nepříjemné artefakty tím, že směr pohledu bude téměř rovnoběžný s rovinou některého z řezů. Z toho důvodu je dobré zajistit, aby se takové řezy vůbec nezobrazovaly.

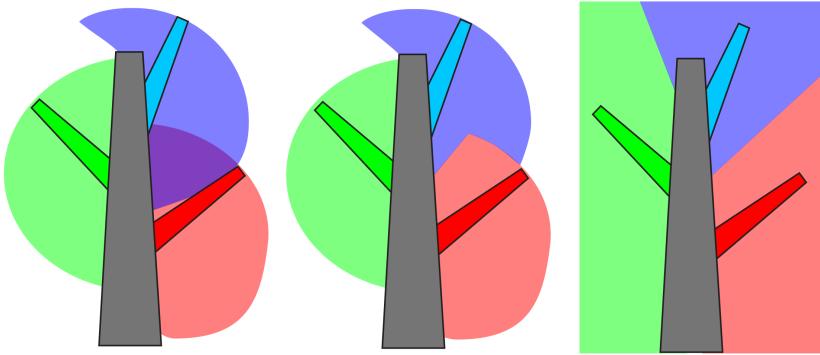
Pro velmi vzdálené stromy se již neprojevuje efekt paralaxy skoro vůbec a pozorovatel vnímá pouze tvar obrysu stromu a jeho celkovou barevnost. Z toho důvodu lze tyto velmi vzdálené stromy nahradit zjednodušenou verzí LOD, která je představována pouze dvěma navzájem kolmými řezy.

### 3.3.1 Animace v image-base modelu

Bohužel z povahy konstrukce trsu řezů dochází ke ztrátě velikého množství informací, které jsou podstatné pro provedení animace v rozsahu a kvalitě odpovídající postupu popsaného v kapitole 3.1. Větve jsou většinou zakryty listy. Jednotlivé listy se také překrývají a tak například o listech, které nejsou vidět nelze získat žádné informace. Naštěstí nemají modely zobrazené touto metodou vysokou percepční váhu (jde o nižší LOD) a lze si tudíž dovolit snížení kvality a přistoupit k animaci zcela jinak. Základním předpokladem je, že deformace

se provádí pouze v dvourozměrném prostoru textury jednoho řezu. Zatímco původní deformace je realizovatelná ve vertex shaderu, tento postup využije s výhodou možností fragment shaderu. Vyplývá z toho však nepříjemné omezení, které mění původní problém, který lze shrnout otázkou: „*Kam se posune tento bod?*“ (nazývejme **přímá transformace**), na problém typu „*Jaký bod se přesune do tohoto místa?*“ (nazývejme **zpětná transformace**). Fragment shader totiž neumožňuje měnit aktuálně zpracovávanému fragmentu pozici, na kterou bude zapsán ve výstupním bufferu. Uvážíme-li, že se na určitou pozici může po transformaci zobrazit i několik fragmentů, bylo by pro korektní zobrazení nutné projít každý texel textury provést přímou transformaci a zjistit, zda není dosaženo aktuální pozice. Takové řešení je ovšem značně nevhodné.

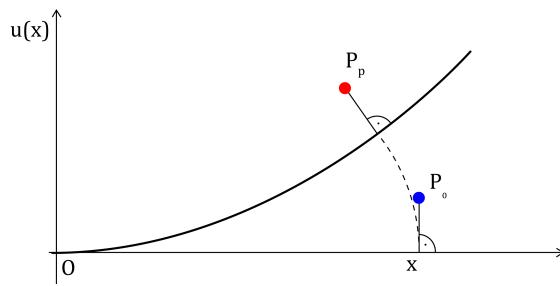
Jestliže je třeba provádět v rámci jediné textury různé transformace, které přísluší různým zobrazeným větvím, pak je třeba znát, jaká transformace se v daném bodě bude provádět. Musí být tedy předem jasné, jaká větev se do daného bodu může zobrazit. Současně s generováním textur pro jednotlivé řezy, je možné připravit i texturu, která bude tuto informaci obsahovat. Jednoduchým řešením, které lze i efektivně implementovat na GPU, je vytvoření oblastí, které odpovídají buňkám Voronoiova diagramu – tedy pro daný bod nalezneme nejbližší bod, kde je zobrazena větev ve výchozím stavu. Tento postup můžeme nazvat „**propagace dat**“.



Obrázek 3.18: Schematicky znázorněné oblasti, kam se mohou větve deformovat. Černě vytaženy obrys výchozí polohy zobrazených větví (vlevo) reálný ohyb – oblasti se překrývají, (uprostřed) oblasti bez překryvů, (vpravo) oblasti Voronoiova diagramu

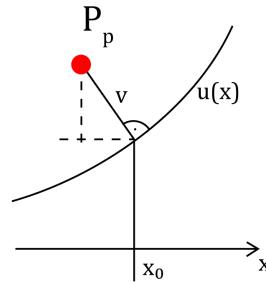
Pokud tedy známe, jaká větev se může do daného bodu  $P$  deformovat, známe také parametry této deformace. Z důvodů zachování koherence animace (větve se pohybují zhruba stejně) využijeme metodu deformace vycházející z ohýbu trojrozměrného modelu. Trojrozměrný ohyb je prováděn v souřadném systému větve ( $S_b$ ). Po převedení bázových vektorů  $(\vec{r}_b, \vec{s}_b, \vec{t}_b)$  do souřadného systému řezu ( $S_p$  – tedy vektoru  $\vec{r}_p, \vec{s}_p, \vec{t}_p$ ) je možné postupovat prakticky stejně. Uplatníme myšlenkový model, kdy bod  $P_0$  mimo větev (představovaná osou x) bude při deformaci udržovat svou relativní pozici na normále k větvi (viz obrázek 3.19). Potřebujeme tedy zjistit, ke kterému bodu větve je bod  $P_p$  ukotven.

Problém představuje určení parametru  $x$  ohýbové funkce  $u(x)$  pro bod  $P_p$  (deformovaný z  $P_0$ ), který zmíněnou polohu na větvi definuje. Pro přesné určení je nutné zjistit nejbližší



Obrázek 3.19: Myšlenkový model ohybu pomocí zpětné transformace

bod na ohybové funkci a pro ten pak zjistit jeho vzdálenost na křivce od počátku.

Obrázek 3.20: Zjišťování nejbližšího bodu na ohybové křivce  $u(x)$ 

Bod  $x_0$  lze určit z následujících rovnic, kde hledáme minimum pro vzdálenost  $v$ :

$$\begin{aligned} \frac{dv}{dx_0} &= 0 \\ v &= \sqrt{(x_0 - P_{px})^2 + (u(x_0) - P_{py})^2} \\ \frac{d\sqrt{(x_0 - P_{px})^2 + (c_2 x_0^2 + c_4 x_0^4 - P_{py})^2}}{dx_0} &= 0 \end{aligned} \quad (3.35)$$

Řešení je omezeno podmínkou:

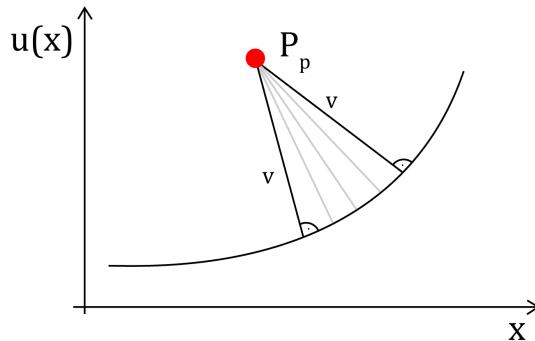
$$P_{px} \neq \pm \sqrt{\frac{-c_2 \pm \sqrt{c_2^2 - 4c_4 P_{py}}}{2c_4}} \quad (3.36)$$

a transformuje se na problém hledání kořenů polynomu 7. stupně:

$$4c_4^2 x^7 + 6c_2 c_4 x^5 + (2c_2^2 - 4P_{py} c_4) x^3 + (1 - 2P_{py} c_2) x - P_{px} = 0 \quad (3.37)$$

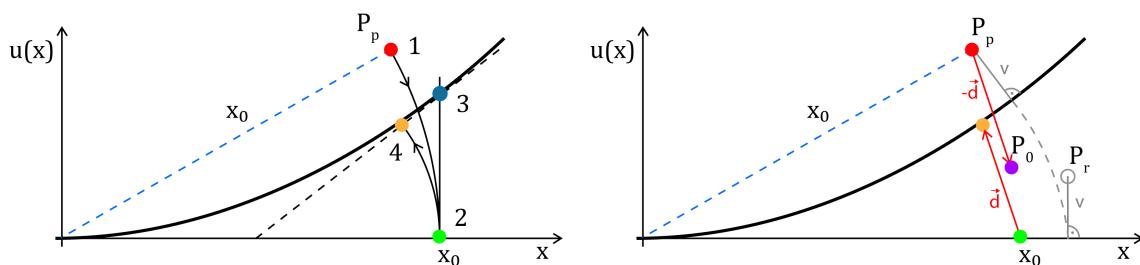
Tento polynom nemá triviální řešení a může se stát, že řešení je i několik – což odpovídá situaci, kdy se více bodů z křivky deformuje do stejného místa (viz obrázek 3.21 ).

Krom obtíží s řešením kořenů polynomu (3.37) se tak přidává i určitá nejednoznačnost. Stále je však ještě potřeba vyřešit vzdálenost takto nalezeného bodu na ohybové křivce. Uvážme-li, že takový složitý postup by se měl provádět pro každý zobrazovaný fragment textury, dostaneme se do patové situace. Zde je třeba rezignovat na dodržení shodného po-



Obrázek 3.21: Znázornění problému mnohoznačnosti při určování, který bod je řídící pro zpětnou deformaci bodu  $P_p$ . U reálného stromu dojde v tomto případě ke kolizi takových větví.

stupu jako v případě přímé deformace geometrie. Počáteční bod  $O$  příslušné větve převedeme do souřadného systému textury  $S_p$  a zjistíme projektovanou délku  $L$  větve. **Hledanou hodnotu  $x$  pak nahradíme vzdáleností  $|OP|$ , kterou normujeme délkou  $L$  a omezíme na rozsah  $\langle 0; 1 \rangle$ .** Pro takovou hodnotu  $x$  určíme deformační vektor  $\vec{d}$  (posunutí bodu) a aplikujeme v opačném směru  $-\vec{d}$  než při přímé transformaci.



Obrázek 3.22: Postup zpětné transformace: (vlevo) – postup určení deformace: Bod  $P_p$  (1), pro který určujeme deformaci je ve vzdálenosti  $x_0$  od počátku (2). Zjistíme hodnotu ohybové funkce v bodě  $x_0$  (3). provedeme korekci délky (4). (vpravo) – určíme vektor posunutí  $\vec{d}$  mezi body  $x_0$  a (4) (v obrázku vlevo) odečtením od bodu  $P_p$  se dostáváme na pozici předobrazu  $P_0$ . Šedivě je naznačen bod  $P_r$ , jež je skutečným předobrazem respektujeme-li myšlenkový model.

Je patrné, že chyba způsobená takovou hrubou approximací  $x_0$  je relativně malá pro body  $P_p$ , ležící přímo na ohybové křivce, ale roste tím víc, čím je od ní bod vzdálenější. Použitím vztahů (3.9) aplikovaným na projektované bázové vektory  $\vec{r}_p$ ,  $\vec{s}_p$  (zastoupené vektorem  $\vec{b}_p$ ) dostaneme upravené vztahy pro korekce délky  $\vec{c}_{\vec{b}}$ ,  $\vec{c}_{\vec{b}}$ .

$$\vec{c}_{\vec{b}} = \vec{t} + \vec{b} \cdot f'_{\vec{b}}(x) \cdot \frac{d_{\vec{b}}(x)}{s_{\vec{b}}(x)} \quad (3.38)$$

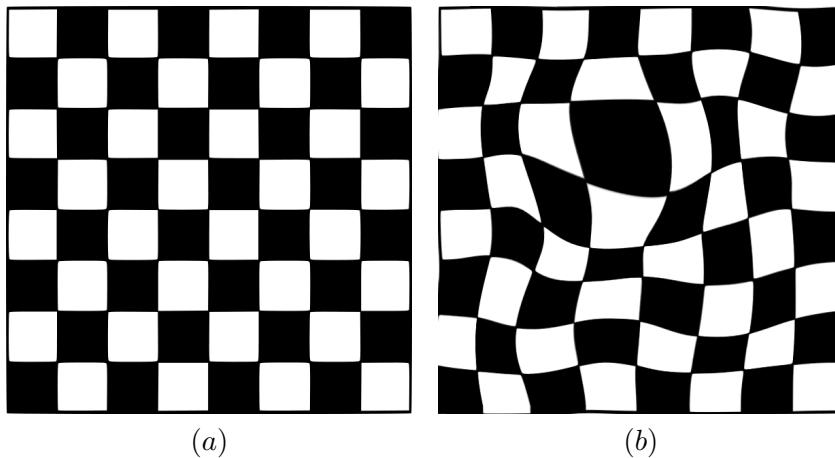
Deformaci v prostoru textury lze tedy popsat následovně:

$$\vec{P}_0 = \vec{P}_p - (u_{\vec{r}}(x) \cdot \vec{r}_p + u_{\vec{s}}(x) \cdot \vec{s}_p - (\vec{c}_{\vec{r}} + \vec{c}_{\vec{s}})) \quad (3.39)$$

Zpětnou deformaci lze provádět i hierarchicky a to v pořadí od nejnižší úrovně větví v hierarchii až po kmen (tedy opačně, než v přímé transformaci). Je však nutné vyvážit náročnost takového postupu a jeho přidanou hodnotu. Například je možné deformaci provádět pro několik nejvyšších úrovní hierarchie. S opačným postupem tvorby hierarchické deformace v image-based modelu souvisí i problém správné aplikace směrové složky větru. Ve vztahu (3.17) je použit vektor  $\vec{t}$  ovlivněný nadřazenými deformacemi. Ten však při opačném postupu zpětné deformace není znám. Předpokládáme-li provádění zpětné deformace pouze ve dvou úrovních (kmen + větve první úrovně) a rozumě malé výchylce, pak lze za vektor  $\vec{t}$  dosadit nedeformovaný vektor  $\vec{t}_p$ , místo  $\vec{W}$  použijeme  $\vec{W}_p$  (směr větru v souřadné soustavě řezu). Vzniklá chyba není natolik vážná, aby představovala překážku u modelů s nižší percepční vahou, jakými nižší stupně LOD bezpochyby jsou.

Popsaná metoda má předpoklad dobré fungovat pro větve tvořící siluetu stromu v daném řezu. U takových větví se předpokládá, že jejich podélný vektor je zhruba rovnoběžný s rovinou řezu. Současně je třeba si uvědomit, že i fáze propagace dat má určitá omezení a funguje lépe pro siluetové větve. Jak se však ukazuje, právě tyto větve hrají největší roli při pozorování stromu s nižší percepční vahou. Hrají i podstatnou roli při přechodu mezi úrovněmi LOD.

Zbývá vyřešit pohyb jednotlivých listů. Pohybem větví nižších úrovní hierarchie a listů vzniká vysokofrekvenční šum v prostoru obrazu. Tento šum pozorovatel vnímá, ale neroteznává již, jestli je způsoben pohybem větví, či samotných listů. Lze ho tedy napodobit použitím jediné deformace. Pro šum vyšších frekvencí je zároveň těžké rozlišit, zda jde čistě o pohyb, či jen změny barvy. Představíme-li si například, že list je zobrazen do jediného pixelu, pak pouhým natáčením listu zjevně měníme barvu zmíněného pixelu (barevný šum). Nerotační pohyb listu může způsobit obarvení jiného pixelu (pohybový šum). Pohyb listů může pak ústít v určité nepravidelné chvění elementárních fragmentů obrazu (případně větších oblastí – podle toho, kolik fragmentů zobrazuje stejný list). Pohybovou složku šumu lze



Obrázek 3.23: Příklad dvourozměrné deformace obrazu technikou displacement-mappingu. Originál (a) a deformovaný obraz (b).

vcelku dobře napodobit metodou zvanou **displacement-mapping**. V základní verzi získává z řídící textury informace o posunutí v prostoru deformovaného obrazu – v tomto případě barevné textury. Vhodnými změnami řídící textury (posun souřadnic, kombinace různých textur) lze plynule deformovat výsledný obraz.

Šumový displacement-mapping je možné zapsat například takto:

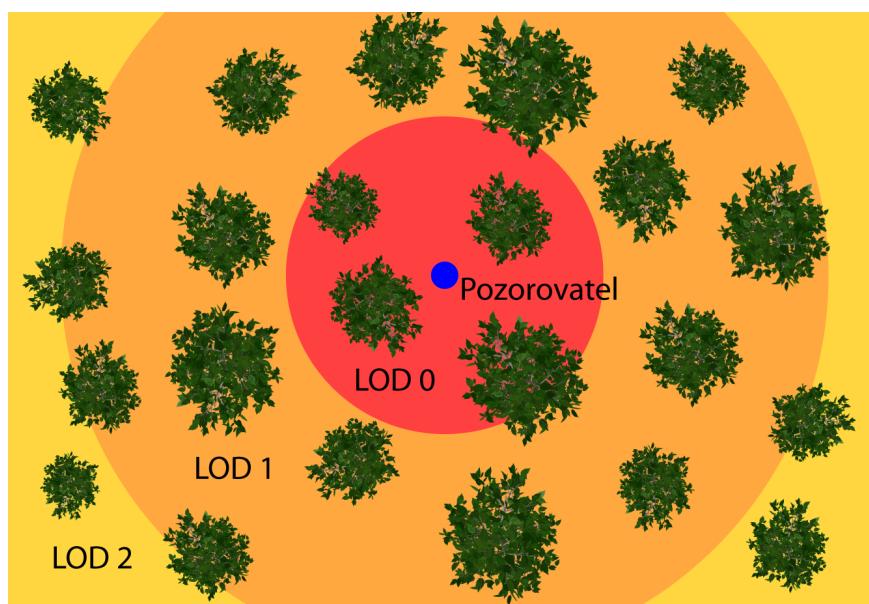
$$\text{color} = T_{\text{color}}(T_{\text{def}}(\vec{o}_t + t \cdot \vec{d}_1) + T_{\text{def}}(\vec{o}_t + t \cdot \vec{d}_2)), \quad (3.40)$$

kde  $T_{\text{def}}(\vec{x})$  je texturovací funkce, jež zobrazuje souřadnice  $\vec{x}$  na jiné, a  $T_{\text{color}}(\vec{x})$  přiřazuje daným souřadnicím  $\vec{x}$  barvu. Jak je patrné, deformace závisí na obsahu deformační textury. Pokud bude obsahovat jen plynulé přechody, bude i deformace spojité.

Barevnou složku šumu je možné vytvořit například tak, že pro každý zobrazený list v řezu měníme jeho natočení ke světlu (normálu) a následně provádíme standardní vyhodnocení Phongova osvětlovacího modelu. Je tedy nezbytné, aby tato transformace normál proběhla pro konkrétní list koherentně - nejlépe stejně. V opačném případě by mohl výsledný šum mít mnohem vyšší frekvenci, než by bylo vhodné.

### 3.3.2 Řízení úrovně detailu

V předchozích kapitolách jsou navrženy metody, jak zobrazovat vegetaci (stromy) ve třech různých stupních detailu. Nejvyšší (označme ji  $LOD_0$ ) využívá podrobné geometrické reprezentace a poskytuje nejlepší kvalitu zobrazení pro pohledy zblízka. Přesvědčivě tak lze zobrazit i jednotlivé pohybující se listy stromu.  $LOD_0$  je však vcelku náročná na výkon a není možné ji uplatnit plošně na všechny stromy rozsáhlé lesní scény. Pro stromy s nižší percepční vahou (není třeba je zobrazovat tak podrobně) byly navrženy image-based metody (označme  $LOD_1$  a  $LOD_2$ ). Ty poskytují sice nižší kvalitu zobrazení (na úrovni zobrazení a pohybu celého stromu), ale jsou méně náročné na výkon.



Obrázek 3.24: Znázornění zón, kde se instance stromů vykreslují danou metodou.

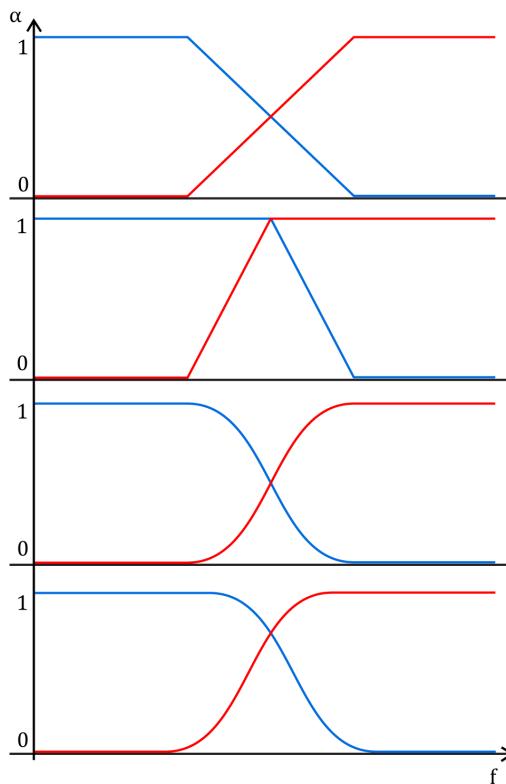
Pro určení přibližné percepční váhy se běžně využívá vzdálenostní kritérium. Blízké stromy v okolí pozorovatele jsou tedy zobrazeny metodou  $LOD_0$ , od určité vzdálenosti  $v_1$  se využívá metody  $LOD_1$  a obdobně pro vzdálenost  $v_2$  a metodu  $LOD_2$ , přičemž platí, že  $0 < v_1 < v_2$ .

Jedná se tedy o diskrétní třístupňový LOD systém. Určení vzdálenosti k jednotlivým instancím stromu lze provádět buď v prostoru (vhodné např. aplikace typu letecký simulátor), nebo pouze v horizontální rovině (aplikace s pohledem omezeným na pohled z okolí úrovně terénu). Vzdálenosti  $v_1$  a  $v_2$  je třeba přizpůsobit konkrétním modelům vegetace a scéně obecně, stejně jako optimalizaci z hlediska výkonu, na který mají podstatný vliv.

### 3.3.3 Přechody mezi úrovněmi detailu

Kvalitu metody vytvoření a řízení LOD lze posuzovat jak z hlediska optimalizace výkonu, tak i podle toho, nakolik si je pozorovatel vědomý, že k nějaké optimalizaci vůbec dochází. Bohužel, jak již bylo v předchozím textu naznačeno, diskrétní metody LOD se mohou potýkat s nežádoucím jevem zvaným LOD-popping, který prozrazuje, že se scéna mění, aniž k tomu je z hlediska pozorovatele zřejmý důvod. Nejnaivnějším způsobem jak přecházet mezi úrovněmi detailu je skokové přepnutí. Právě skoková změna na sebe poutá pozornost a tak nejde o dobré řešení. Vhodnější je jednotlivé úrovně plynule prolínout. Toho lze docílit buď technikou zvanou *morphing*, která plynule převádí tvar jednoho tělesa na druhé transformací geometrie, nebo prostým prolnutím výsledných obrazu díky řízení průhlednosti. Ukazuje se však, že přístup známý jako *cross-fade* může způsobovat určité nechtěné problémy. Cross-fade mezi objekty  $A$  a  $B$  na základě proměnné  $f \in \langle 0; 1 \rangle$  je možné symbolicky popsat následovně:

$$R = f \cdot A + (1 - f) \cdot B \quad (3.41)$$



Obrázek 3.25: Různé metody přechodů mezi LOD. Odshora: cross-fade, fade in background, soft cross-fade, shifted soft cross-fade.

Může se ale stát, že se objekt stane částečně průhledným, ačkoliv průhledný být nemá (viz obr. 2.4). Tento jev lze eliminovat použitím pozměněné metody, kterou označíme *fade in background*. Je tím zaručeno, že alespoň jedna úroveň detailu bude vždy neprůhledná a tím pádem nebude skrz objekt vidět. Řízení přechodu mezi úrovněmi  $A$  a  $B$  závisející na proměnné  $f$  lze symbolicky vyjádřit takto:

$$R = \min(1, 2(1 - f)) \cdot A + \min(1, 2f) \cdot B \quad (3.42)$$

Pro další vylepšení je možné uvažovat měkké přechody, kde nebude průhlednost (či neprůhlednost) utlumována lineárně, ale například podle následujícího předpisu:

$$\begin{aligned} \text{smooth}(x) &= \begin{cases} 1 & \text{pro } x \leq 0 \\ -1 & \text{pro } x \geq \pi \\ \cos(x) & \text{pro } x \in (0; \pi) \end{cases} \\ R &= \frac{1}{2} * \left[ \text{smooth} \left( \frac{f - s}{1 - s} \cdot \pi \right) + 1 \right] \cdot A + \left[ 1 - \frac{1}{2} * \left( \text{smooth} \left( \frac{f}{1 - s} \cdot \pi \right) + 1 \right) \right] \cdot B \end{aligned} \quad (3.43)$$

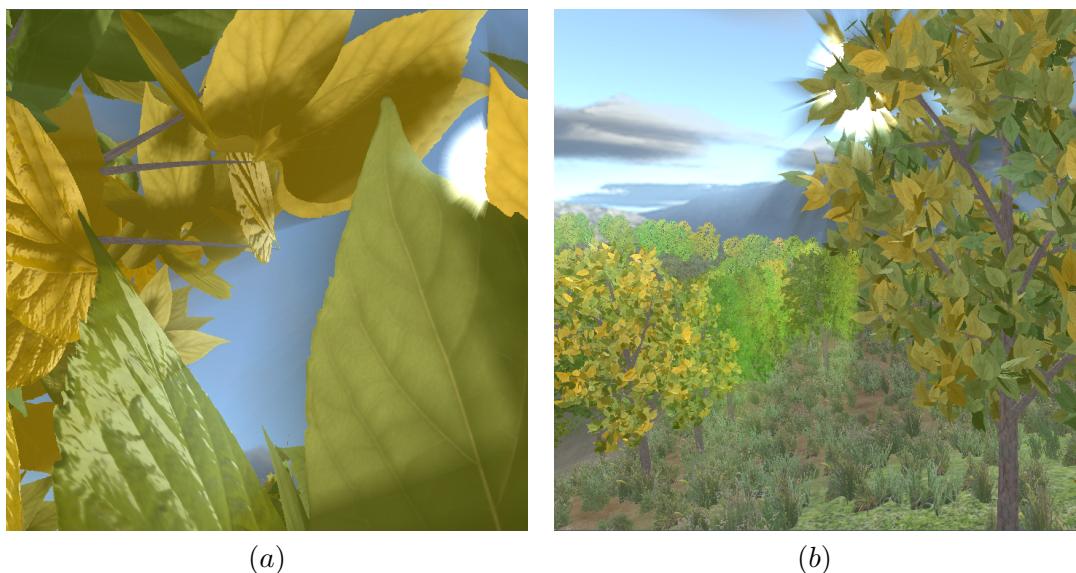
Proměnná  $f$  ovlivňuje fázi přechodu a parametr  $s$  ovlivňuje vzájemné posunutí přechodových křivek.



# Kapitola 4

## Realizace

V předchozí kapitole byly teoreticky rozebrány nejpodstatnější a nejdůležitější části práce. Ovšem i tak zbývá řada problémů, které je nutné v rámci realizace dořešit.



Obrázek 4.1: Výsledné zobrazení. Detail listů (a), kde je patrná průsvitnost i lesklost. Rozsáhlejší scéna s více stromy (b).

V následujícím textu bude stručně popsáno, jejich řešení. Budou zmíněny technologie a knihovny, které byly pro realizaci použity, a popsána vstupní data výsledného programu, jejich zpracování a použití. Opomenuta nezůstane ani celková architektura systému a konkrétní řešení zobrazování všech úrovní detailu (LOD).

## 4.1 Použité technologie, frameworky a knihovny

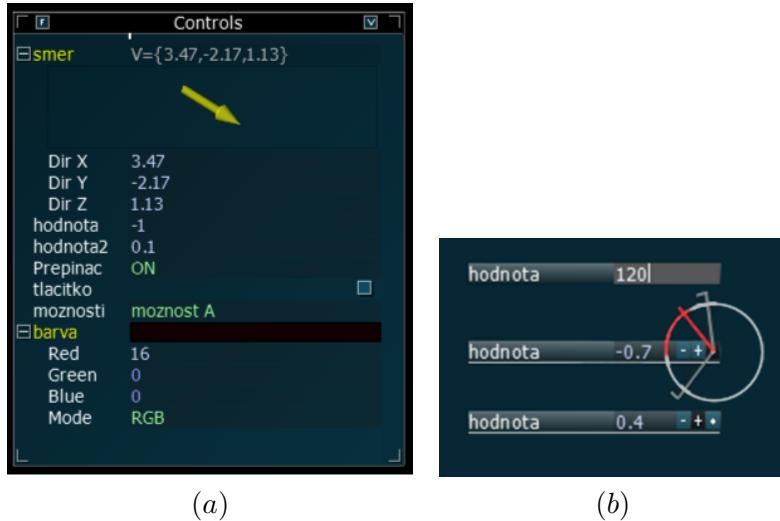
Výsledná aplikace byla implementována v jazyce **C/C++** a využívá knihovnu **OpenGL** [1]. Ta poskytuje efektivní aparát pro práci s grafickým hardware a umožňuje využít většiny jeho nejnovějších vlastností a funkcí. Její výhodou je platformová nezávislost. Stejný kód by tedy mělo být možné přeložit jak pro Windows, tak pro systémy Unixového typu. Následně bylo využito několika dalších knihoven zjednodušujících práci se základní OpenGL:

**GLFW** [9] sada knihovních funkcí určených pro snadné vytváření oken, OpenGL kontextů a mapování vstupních zařízení. Zachovává platformní nezávislost a uplatňuje se zejména ve fázi inicializace aplikace. Pomocí této knihovny je vytvořeno okno potřebných vlastností (podporuje i multisampling).

**GLUT** [22] sada nástrojů pro efektivnější práci. Obsahuje funkce pro vytváření oken a kontextů, mapování vstupů i pro práci s grafickými prvky.

**GLEW** [14] knihovna pro práci s rozšířením (extenzemi) OpenGL. Zahrnuje zejména funkce pro zjišťování, zda ovladač grafického hardware danou extenzi podporuje.

Další knihovnou, která byla využita, je **LODEpng** [16]. Jde o jednoduchý dekodér a enkodér obrazového formátu PNG, který je distribuován formou jednoho .cpp a jednoho .h souboru. Je s výhodou využit pro načítání obrázků do textur.



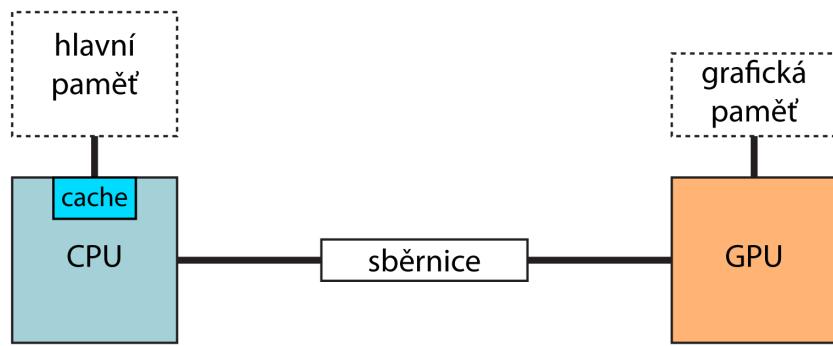
Obrázek 4.2: AntTweakBar, (a) ukázka základních ovládacích prvků (shora): ovladač směru, celočíselná skalární hodnota, desetinná skalární hodnota, dvoustavový přepínač, tlačítko, seznam možností, výběr barvy. (b) různé způsoby zadávání hodnot (shora): přímé zadání, zadání pomocí tzv. RotoSlideru, pomocí krokovacích tlačítek

Pro snadnou tvorbu grafického uživatelského rozhraní (dále jen GUI) byla využita knihovna **AntTweakBar** [5]. Ta umožňuje jednoduchou tvorbu grafických ovládacích prvků, které se nejčastěji používají při práci s 3D grafickou aplikací (viz obr. 4.2). Těmito ovládacími prvky

pak lze snadno bezprostředně a přímo ovlivňovat chování zobrazované scény - v tomto případě stromů.

## 4.2 Architektura LOD

Realizace systému vykreslování a řízení úrovní jednotlivých stromů reflektuje možnosti současných grafických karet. Jelikož je objem dat zpracovávaný v každém snímku relativně velký a současně se z velké části nemění, lze tyto neměnná data přesunout do paměti GPU a tím i zefektivnit jejich zpracování. Ušetří se tím zbytečné přenosy po sběrnici mezi hlavní pamětí a pamětí grafické karty. Tento přístup je v dnešní době standardem a OpenGL ho samozřejmě podporuje ve formě tzv. *Vertex Buffer Objects* (VBO), což je struktura obsahující data příslušná vrcholům jako jejich atributy (např.: pozice, normála, barva, atd.).



Obrázek 4.3: Různé druhy pamětí: hlavní paměť při CPU a grafická paměť na GPU. Přenos dat po sběrnici může mít negativní vliv na výkon.

Tato data mohou být umístěna jak v hlavní paměti, tak i v paměti GPU. Požadavek na umístění těchto dat lze vyjádřit při jejich zápisu do VBO příkazem

```
glBufferData(..., const GLvoid * data, GLenum usage)
```

Parametr `usage` definuje povahu dat. Např. hodnota `GL_STATIC_DRAW` informuje OpenGL o tom, že data se nebudou měnit a je tedy vhodné přesunout je do paměti GPU. Podle konkrétní implementace OpenGL a možností grafického hardware je tedy tedy tento přesun proveden, či je umístění dat jinak optimalizováno. Naproti tomu hodnota `GL_STREAM_DRAW` naznačuje, že data budou v každém snímku změněna a přenos mezi CPU a GPU nelze ušetřit. Proto budou v takovém případě data raději v hlavní paměti.

Kromě VBO existuje ještě obdobný konstrukt pro uložení indexů indexované geometrie - tzv. *Element Buffer Objects* (EBO). Indexovaná geometrie je výhodná zejména v případě, že se atributy vrcholů nemění a vrcholy jsou použity v rámci tvorby geometrie vícekrát (např. vrchol je společný pro mnoho trojúhelníků).

Protože se předpokládá zobrazení více instancí téhož stromu, jeví se jako přirozené využít techniky *instancování na GPU*, která vytváří jednotlivé instance dynamicky až v rámci

hardware. Tato technika se vyplácí zejména pro zpracování a zobrazování většího počtu instancí. Jeví se tedy jako vhodné, využít ji pro zobrazování zejména instancí s nižším LOD (LOD1 a LOD2), kterých je typicky řádově více než instancí nejvyššího detailu. Myšlenka instancování na GPU je prostá. Pokud se instance vzájemně liší pouze několika globálními parametry (pozice a orientace ve scéně, příp. další atributy) není třeba každou instanci vykreslovat zvláštním příkazem. Místo toho je typicky využit příkaz:

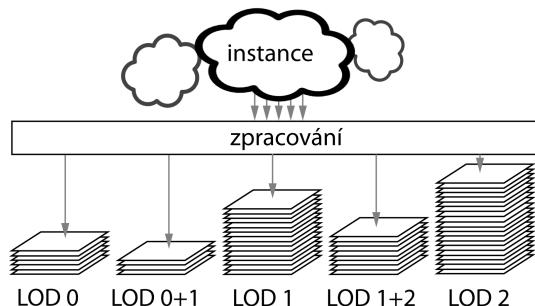
```
glDrawElementsInstanced(..., GLsizei pocetInstanci)
```

Ten vykreslí zadaný počet instancí připojené indexované geometrie. Jelikož je však třeba jednotlivé instance správně umístit do scény a přiřadit jim i další specifické parametry, je třeba předat konkrétní instanci příslušná data. K tomu účelu se využívá princip tzv. *instančních atributů*. Narozdíl od běžných atributů, které přísluší jednotlivým vrcholům (typicky normála, tangenta, texturovací souřadnice apod.), instanční atributy jsou stejné pro všechny vrcholy dané instance, ale liší se mezi instancemi. Toto chování lze zajistit příkazem:

```
glVertexAttribDivisor(GLuint attribDesc, GLuint divisor)
```

Parametr **divisor** určuje, pro kolik instancí bude atribut popsáný **attribDesc** stejný. Hodnota 0 vrací chování zpět na standardní - tedy atributy se různí pro jednotlivé vrcholy. Instanční atributy je dobré uložit taktéž do VBO. Jelikož implicitně se všechny instance vykreslí díky použití stejných dat na stejnou pozici ve scéně, je nutné ve vertex shaderu transformovat všechny vrcholy dané instance příslušným způsobem. Z toho důvodu se jako jeden z instančních atributů předává i instanční transformační matice. Pořadí, v jakém jsou předány instanční atributy, určuje i pořadí vykreslování jednotlivých instancí.

Pro realizaci aplikace pracující s LOD jsou výše zmíněné možnosti zásadní. Jelikož je nutné v každém snímku určit pro každou instanci v jakém LOD se bude vykreslovat a to závisí v tomto případě na vzdálenosti od pozorovatele, jsou instance zpracovávány postupně a jsou přiřazeny do jedné ze zobrazovacích front (viz obr. 4.4). Některé instance pak lze vyřadit



Obrázek 4.4: Fronty instancí. LOD0 jsou instance v nejvyšším stupni detailu. Fronta LOD0+1 představuje instance v přechodu mezi úrovněmi LOD0 a LOD1. Obdobně i pro LOD2.

ze zobrazovacího procesu třeba proto, že neleží v prostoru aktuálního pohledu<sup>1</sup> (např. leží za pozorovatelem).

---

<sup>1</sup>stromy v těsném okolí pozorovatele ale mohou vrhat stín, a proto jsou zachovány

Instancím, které jsou pro daný pohled právě v přechodu mezi různými stupni LOD, je v rámci zpracování určena fáze přechodu a podle [10] jsou zobrazeny. V pseudokódu lze tento proces popsat následujícím způsobem:

```
function makeTransition(float phase, Tree instance){
    float alpha1;
    float alpha2;
    if (phase<0.5){
        alpha1 = 1;
        alpha2 = 2*phase;
        disableDepthBufferWrite();
        instance.drawLOD_B(instance, alpha2);
        enableDepthBufferWrite();
        instance.drawLOD_A(instance, alpha1);
    } else {
        alpha1 = 2*(1-phase);
        alpha2 = 1;
        instance.drawLOD_B(instance, alpha2);
        disableDepthBufferWrite();
        instance.drawLOD_A(instance, alpha1);
        enableDepthBufferWrite();
    }
}
```

Kvůli průhlednosti, která hraje důležitou roli u nižších úrovní detailu, a která není implicitně řešena v rámci OpenGL, je nutné provádět vykreslování v pořadí od vzdálených instancí k blížším. Z toho důvodu je třeba provádět v rámci zpracování rychlé řazení instancí před vykreslením každého snímku<sup>2</sup>. Důležitý je fakt, že mezi následujícími snímky je zachována určitá koherence. Instance, které jsou v seřazeném pořadí v jednom snímku blízko sebe, budou nejspíš kdesi blízko sebe i ve snímku následujícím. S výhodou tedy lze použít částečného řazení s nižší (lineární) časovou složitostí. V tomto případě je využito jednoho průchodu algoritmu známého jako *bubble-sort* (dále jen *lazy-sort* neboli *líné řazení*). Toto řazení funguje na úrovni jednotlivých instancí a poskytuje uspokojivé výsledky. Fronty instancí LOD1 a LOD2 jsou následně vykreslovány pomocí instančního zobrazování. Ostatní fronty, u kterých se předpokládá, že neobsahují tolik položek, jsou vykreslovány po jednotlivých instancích.

### 4.3 Vstupní data a předzpracování

Aplikace načítá celou řadu dat, se kterými dále pracuje. V první řadě je nutné načíst popis stromu, dále pak textury použité pro jeho zobrazení. Načtená data je v řadě případů nutno předzpracovat, aby je bylo možné využít. Právě popisem vstupních dat a jejich předzpracováním se bude zabývat následující text.

---

<sup>2</sup>předpokládáme-li změnu pozice a pohledu pozorovatele

### 4.3.1 Formát OBJT

Jelikož běžné modely stromů neobsahují implicitně informace o topologii stromu, která je nezbytná pro provedení animace, byl navržen formát vstupního souboru OBJT, který zahrnuje právě tyto informace. Soubor formátu OBJT je vstupním souborem popisujícím model stromu. OBJT je textový formát obsahující 4 druhy záznamů (oddělených standardním oddělovačem řádků):

- pojmenování stromu na začátku souboru entitou `name = "%string%"`
- entita popisující větev

```
B %int% {                                // identifikátor větve
    l    %int%                         // úroveň v hierarchii
    d    %float%                        // délka větve
    z    %float% %float% %float%        // pozice začátku větve
    k    %float% %float% %float%       // pozice konce větve
    r    %float% %float% %float%       // bázový vektor r
    s    %float% %float% %float%       // bázový vektor s
    t    %float% %float% %float%       // bázový vektor t (podélný)
    p    %int%                          // identifikátor rodičovské větve
    x    %float%                        // v kolikátině rodičovské větve se větev odpojuje
}
```

- entita popisující list

```
L %int% {                                // identifikátor listu
    r    %float% %float% %float%      // bázový vektor r
    s    %float% %float% %float%      // bázový vektor s
    t    %float% %float% %float%      // bázový vektor t (podélný)
    p    %int%                          // identifikátor rodičovské větve
    x    %float%                        // v kolikátině rodičovské větve se list odpojuje
}
```

- řádkový komentář začínající `// %string%`

Ve výše uvedeném popisu formátu jsou použity zástupné řetězce za celá čísla (`%int%`), za desetinná čísla (`%float%`) a za znakové řetězce bez oddělovače konce řádků (`%string%`). Všechny souřadnice se předpokládají v souřadném systému celého stromu (též *object space*).

Data ze zdrojového OBJT jsou načtena, větve jsou vytvořeny jako kužely a listy jako ploché čtverce (quads). Vrcholy a jejich atributy jsou následně uloženy do VBO. Těmito daty jsou:

- pozice v souřadném systému větve
- normála a tangenta v souřadném systému rodičovské větve
- texturovací souřadnice
- vektor hodnot x (viz obr. 3.6)
- souřadnice do datové textury pro příslušnou větev (její obsah bude popsán dále)

### 4.3.2 Textury

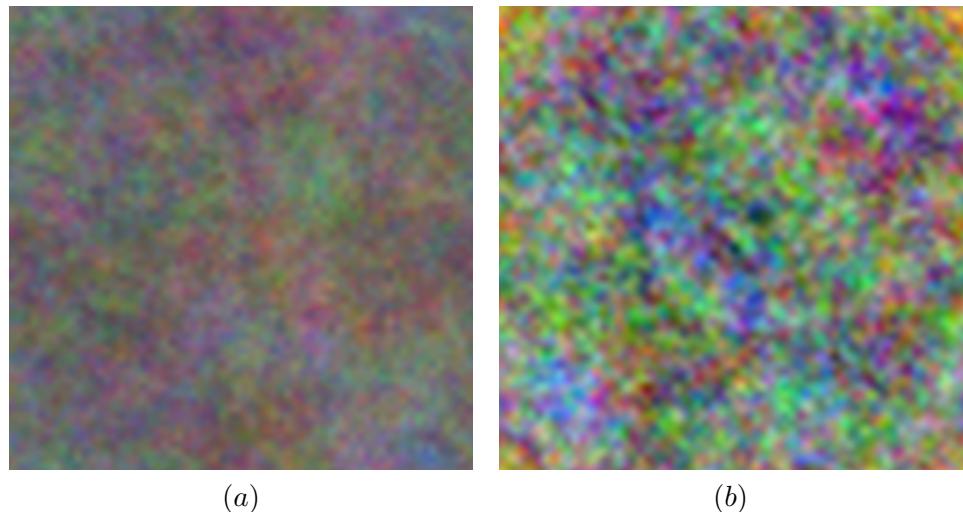
Ačkoliv je v celé aplikaci použito velké množství textur (pro skybox, krajinu, atd.), zde budou zmíněny pouze ty nejdůležitější, které se vážou přímo k tématu zobrazování stromů.

Pro zobrazení 3D modelu stromu je použito několik textur, které jsou načítány při spuštění aplikace. Kromě textur použitých pro zobrazení listů (viz obr. 3.15) je pro obarvení kmene a větví načtena textura jejich struktury (viz obr. 4.5).



Obrázek 4.5: Zdrojová textura barvy kmene a větví.

Dále jsou načteny šumové textury řídící animaci (obr. 4.6)



Obrázek 4.6: Šumové textury animace větví (a) a listů (b)

Přímo zásadní je však textura, v níž jsou uložena data potřebná pro provedení deformace ve vertexovém procesoru. Tuto texturu je nutné pro daná vstupní data popisující strom vytvořit. Pro její sestavení je třeba převést zejména bázové vektory větve ( $\vec{r}, \vec{s}, \vec{t}$ ) do souřadného

systému rodičovské větve (převedené vektory  $\vec{r}_b$ ,  $\vec{s}_b$ ,  $\vec{t}_b$ ). Zároveň je potřeba pro každou větev vytvořit (pokud možno) unikátní vektor  $\vec{m}_v$  posunu v šumové textuře. Informace doplňují záznamy o délkách rodičovských větví  $l_0 - 3$ . Jelikož je třeba uložit do textury normalizované vektory, je použita textura typu `GL_RGBA32F`.

Minimální sada dat v datové textuře tedy vypadá následovně:

	obsah textury									
i+1	:									
i	$m\vec{v}_0.x$	$m\vec{v}_2.x$	$s_b,0.x$	$s_b,1.x$	$s_b,2.x$	$s_b,3.x$	$r_b,0.x$	$r_b,1.x$	$r_b,2.x$	$r_b,3.x$
	$m\vec{v}_0.y$	$m\vec{v}_2.y$	$s_b,0.y$	$s_b,1.y$	$s_b,2.y$	$s_b,3.y$	$r_b,0.y$	$r_b,1.y$	$r_b,2.y$	$r_b,3.y$
	$m\vec{v}_1.x$	$m\vec{v}_3.x$	$s_b,0.z$	$s_b,1.z$	$s_b,2.z$	$s_b,3.z$	$r_b,0.z$	$r_b,1.z$	$r_b,2.z$	$r_b,3.z$
	$m\vec{v}_1.y$	$m\vec{v}_3.y$	$l_0$	$l_1$	$l_2$	$l_3$				
i-1	:									

Tabulka 4.1: Minimální sada dat uložená v datové textuře pro LOD0.



Obrázek 4.7: Ukázka z datové textury pro LOD0. Číslo řádku  $i$  odpovídá identifikátoru větve.

Velice důležitou je textura sezónních barev (viz obr. 4.8). Jde o jednořádkovou texturu. Z ní je interpolována sezónní barva ve smyslu člena  $c_s(s)$  ze vztahů (3.34). Parametr `sezona` lze použít přímo jako souřadnici do této textury.



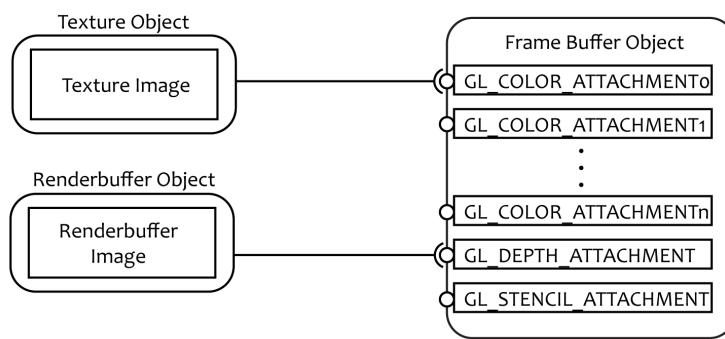
Obrázek 4.8: Ukázka textury sezónních barev. Obrázek je zvětšený, otočený o  $90^\circ$  a šedá šachovnice vizualizuje průhledné pixely.

Zobrazování LOD1 a LOD2 vyžaduje rovněž vytvoření několika speciálních textur. Jejich vytvořením se zabývá další sekce (4.3.3).

### 4.3.3 Předzpracování pro LOD

Pro zobrazení a animaci LOD1 a LOD2 je třeba vytvořit během předzpracování sadu textur. Jednak je to množina textur, které jsou nezbytné i pro zobrazení statického stromu, druhak se pak přidávají i textury, jež jsou nezbytné pro provedení animace. Následující popis uvádí postup generování těchto textur pro LOD1. Pro LOD2 je postup obdobný.

Ke zobrazení statického stromu pomocí trsu řezů je třeba vytvořit textury obsahující pohled na strom z daného úhlu v několika vrstvách. Tyto textury lze relativně jednoduše generovat použitím techniky *offscreen-rendering* neboli vykreslování do textur. V OpenGL se toho dosáhne připojením speciálního tzv. *Frame Buffer Object* (FBO) s napojenými texturami (viz obr. 4.9).



Obrázek 4.9: Mechanismus Frame Buffer Object v OpenGL.

Následující kód ukazuje, jak takovýto FBO vytvořit:

```

// vytvoreni FBO
GLuint fboID = 0;
GLenum buffers[3] = { GL_COLOR_ATTACHMENT0,
                      GL_COLOR_ATTACHMENT1,
                      GL_COLOR_ATTACHMENT2 };
glGenFramebuffers(1, &fboID);
 glBindFramebuffer(GL_FRAMEBUFFER, fboID);
 glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D,
                       colorTextureID, 0);
 glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT1, GL_TEXTURE_2D,
                       normalTextureID, 0);
 glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT2, GL_TEXTURE_2D,
                       branchTextureID, 0);
 glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D,
                       depthTextureID, 0);
 glBindFramebuffer(GL_FRAMEBUFFER, NULL);

```

Použití - tedy přesměrování vykreslování do vytvořeného FBO probíhá následovně:

```
// pouziti FBO
setupCamera();           // nastaveni kamery podle smeru a hloubky rezu
glBindFramebuffer(GL_FRAMEBUFFER, fboID);
glDrawBuffers(3, buffers);
    draw();             // vykresleni do prijmenych textur
    glBindBuffer(GL_BACK);
glBindFramebuffer(GL_FRAMEBUFFER, NULL);
```

Pro generování řezů je kamera nastavena na orthogonální projekci a vykreslení pouze určitého řezu je zajištěno správným nastavením přední a zadní ořezové roviny kamery (*near*, *far*). Jelikož se v této fázi vykresluje celý strom v normované jednotkové velikosti, je určení hodnot *near* a *far* triviální pro *i*-tý řez z daného směru :

$$\begin{aligned} thickness &= \frac{diameter_{tree}}{count_{slice}} \\ near_i &= distance_{camera} - (0.5 \cdot diameter_{tree}) + i \cdot thickness \\ far_i &= near_i + thickness \end{aligned} \tag{4.1}$$

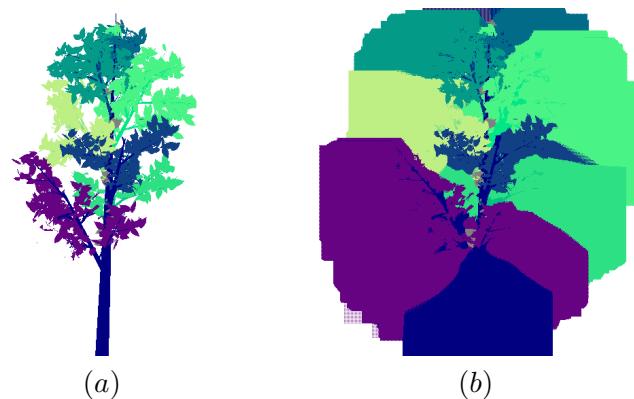
Nastavení projekční matice kamery zajišťuje funkce:

```
glOrtho(float l, float r, float b, float t, float near, float far);
```

Pro každý řez je vytvořena následující sada textur:

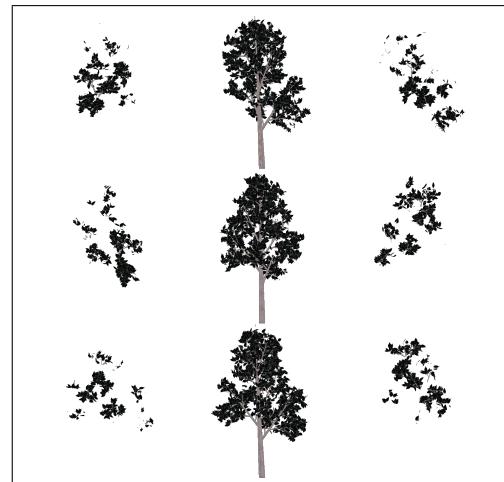
- barevná textura
- hloubková textura
- normálová textura
- textura větvových bodů

Do textur je vykreslen geometrický model v základní pozici (neovlivněný působením větru). Do barevné textury se zapíše nestínovaná barva bez sezónní složky, do textury normál se zapíše pro každý fragment normála (pouze 3 souřadnice) v souřadném systému řezu (budoucí *tangent space*). Čtvrtá složka zaznamenává případně specifické číslo listu. Význam hloubkové mapy je zřejmý. Do textury větvových bodů se zaznamenávají identifikátory rodičovských větví 1. úrovně (napojené přímo na kmen). Následně je na této textuře provedena expanze dat (viz obr. 4.10).



Obrázek 4.10: Ukázka expanze dat (změněna barevnost pro lepší zřetelnost). (a) před expanzí, (b) po expanzi.

Textury daného typu (barevné, normálové, ...) jsou následně sloučeny dohromady (viz obr. 4.11). Děje se tak z důvodu ušetření texturovacích jednotek, jejichž počet je hardwarově omezen. Pro LOD1 s 3 směry řezů po 3 vrstvách a 4 sadami textur by tedy bylo nutné připojit  $(3 \times 3 \times 4) = 36$  textur. Místo toho stačí připojit pouze 4 sloučené textury, což představuje značnou úsporu.



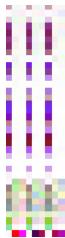
Obrázek 4.11: Slučování textur. Ukázka sloučení barevných textur pro LOD1 do jedné textury.

Zásadní pro provádění animace v LOD operujícími nad trsy řezů je rovněž datová textura s informacemi o jednotlivých větvích. Tato textura obsahuje záznam pro každou větev, který obsahuje infomace o počátku větve v souřadném systému řezu, bázové vektory větve převedené rovněž do souřadného systému řezu, projektovanou délku a pohybové vektory směru posunu v šumové textuře. Tyto informace se různí pro různé směry řezů a musí být tudíž zapsány i do textury pro každý směr zvlášť.

Data v datové textuře jsou tedy následující:

	obsah textury			
	data řezu $j$		data řezu $j + 1$	
	:	:	:	:
$i + 1$	$\vec{o}_s.x$	$\vec{r}_s.x$	$\vec{s}_s.x$	
	$\vec{o}_s.y$	$\vec{r}_s.y$	$\vec{s}_s.y$	
	$\vec{m}\vec{v}.x$	$\vec{r}_s.z$	$\vec{s}_s.z$	...
	$\vec{m}\vec{v}.y$	$l_s$		
$i - 1$	:			:

Tabulka 4.2: Minimální sada dat uložená v datové textuře pro LOD1 a LOD2.



Obrázek 4.12: Ukázka části datové textury pro LOD1. Číslo řádku  $i$  odpovídá identifikátoru větve.

#### 4.3.4 Dynamické parametry

Aplikace nabízí možnost měnit dynamicky řadu parametrů souvisejících s animace stromů i s jejich zobrazováním. Nejpodstatnější jsou zejména tyto:

- směr větru
- síla větru
- amplituda větví (pro každou úroveň zvlášť)
- frekvence větví (pro každou úroveň zvlášť)
- amplituda listů
- frekvence listů
- prahy přechodů LOD
- sezóna
- směr světla

## 4.4 Zobrazení geometrického modelu

Vykreslování stromu v nejvyšší úrovni detailu probíhá v souladu s teorií popsanou v kapitole 3. Vykreslování probíhá ve 2 fázích, nejprve se vykreslí kmen a větve, poté jednotlivé listy. V každé z těchto fází je aktivní jiný shader (program řídící částečně činnost GPU).

Jednotlivé větve jsou zobrazovány jako kužely různých délek a průměrů. Jednotlivé vrcholy tvořící větev jsou zadány v souřadném systému příslušné větve. Dalšími atributy větových vrcholů jsou: normála, tangenta, texturovací souřadnice, index větve v datové textuře a vektor hodnot  $x$  (hierarchické souřadnice viz obr. 3.6). Parametry ohybové funkce  $c_2$  a  $c_4$  (viz 3.8) byly zafixovány na hodnotách  $c_2 = 0.374570$  a  $c_4 = 0.129428$ , což odpovídá hodnotě parametru  $\alpha = 0.2$  (viz 3.7) Nejdůležitější část kódu vertex shaderu je schematicky popsaná níže:

```
// center ... pozice středového řídícího bodu
// t ... podélny vektor
// r ... kolmy vektor, normala
// s ... kolmy vektor, binormala
// x ... hierarchicka souradnice
// amp ... řidici signal animace
// length ... delka veteve
void bend(inout vec3 center, inout vec3 t, inout vec3 r, inout vec3 s,
          in float x, in vec2 amp, in float length){
    float fx = 0.374570*x*x + 0.129428*x*x*x*x;
    float dx = 0.749141*x + 0.517713*x*x*x;
    vec3 corr_r = vec3(0.0); vec3 corr_s = vec3(0.0);
    // vychylkovy prispevek smerove slozky vetru
    amp.x += dot(r, wind_direction) * wind_strength;
    amp.y += dot(s, wind_direction) * wind_strength;
    // ohybova funkce
    fu = fx * amp;
    fu_deriv = dx / length * amp ;
    // zamezeni deleni nulou
    fu_deriv = max(fu_deriv, EPSILON) + min(fu_deriv, EPSILON);
    // korekce delky
    vec2 su = sqrt(vec2(1.0) + fu_deriv*fu_deriv);
    vec2 du = fu / fu_deriv * (su - vec2(1.0));
    corr_r = (t + r*fu_deriv.x)/su.x * du.x;
    corr_s = (t + s*fu_deriv.y)/su.y * du.y;
    // ohnout středovy bod a souradny system veteve
    center = center + x * length * t + fu.x * r + fu.y * s - (corr_s+corr_r);
    t = normalize(t + r*fu_deriv.x + r*fu_deriv.y);
    r = normalize(r - t*fu_deriv.x);
    s = normalize(s - t*fu_deriv.y);
}
```

Následně se postupuje od úrovně 0 - tedy od kmene a provádí se ohyb pomocí popsané funkce **bend**. Pro korektní ohyb je nutné deformovat (natočit) i souřadný systém následující úrovně větví. K tomu se využije skutečnosti, že báze jsou zadané v souřadném systému rodičovské větve. Stačí tedy provést triviální transformaci (br, bs, bt jsou ohnuté bázové vektory rodičovské větve):

```
s = s.x * br + s.y * bs + s.z * bt;
r = r.x * br + r.y * bs + r.z * bt;
t = cross( r , s );
```

Po provedení všech deformací v hierarchii získáme polohu bodu na středovém paprsku příslušné větve. Je tedy nutné ještě odsadit takový bod na povrch větve - využijeme proto vstupního atributu, který udává pozici bodu v souřadném systému větve:

```
position = origin + position.x * br + position.y * bs;
```

Při zobrazování listů se provede stejný postup deformací (list je spojený s větví a pohybuje se s ní), ovšem místo konečného odsazení se provede rotace souřadného systému animující pohyb listu.

```
vec3 bitangent = cross( normal , tangent);
// natocit bazovy system listu
tangent = normalize ( tangent + normal * amplitude.x );
normal = cross( tangent , bitangent );
normal = normalize ( normal + bitangent * amplitude.y );
bitangent = cross ( tangent , normal );

position = origin + ( position.x * bitangent + position.y * tangent );
```

## 4.5 Zobrazení nižších LOD

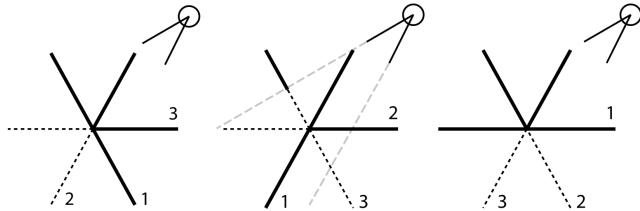
Zásadním problémem, který je nutné v rámci korektního zobrazování nižších LOD vyřešit, je správné zpracování průhlednosti. Průhlednost je podstatná z několika důvodů:

- skrývání řezů skoro rovnoběžných s pohledem
- přechod mezi úrovněmi
- lepší zobrazení samotného řezu

Korektní zpracování průhlednosti při přímé rasterizaci však není v rámci GPU automaticky ošetřeno. Proto je nutné věnovat zobrazování průhledné geometrie zvláštní pozornost. Nejběžnější metodou, která dokáže zobrazovat většinu možných případů prostorových uspořádání průhledných objektů správně, je tzv. *malířův algoritmus*, který vykresluje objekty v pořadí od nejvzdálenějšího k nejbližšímu. Nevýhodou je, že je nutné řadit grafická primitiva (*back-to-front ordering*). V poslední době s rostoucími možnostmi grafického hardware bylo představeno i několik tzv. *order-independent* technik. Uvedeme zejména [2], která je však zatím vázaná na technologii DirectX 11. Další možností je *multisampling* s technikou *alpha-to-coverage*. Využívá vícenásobného vzorkování pixelu obrazu a průhlednost zde řídí, kolik vzorků bude pokryto právě kresleným průhledným objektem. Ačkoliv má tato metoda relativně slibný teoretický základ (výsledek je tím lepší, čím více vzorků použijeme), v praxi často hardware umožňuje vzorkování nanejvýš 16 vzorků. Rozsah běžných průhledností [0...255] se tím sníží na 16 hodnot.

Ošetření problémů spojených s průhledností lze rozdělit do dvou skupin:

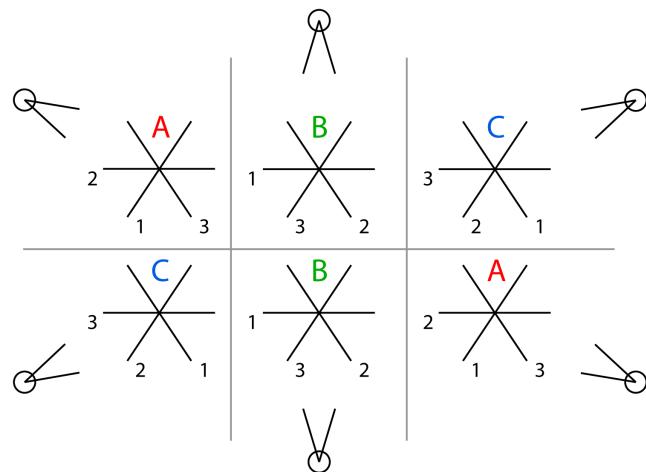
- průhlednost v rámci jedné instance
- průhlednost v rámci instancí mezi sebou



Obrázek 4.13: Schematicky znázorněné problémy vykreslování průhledných, křížících se řezů. Tlustou plnou čarou jsou naznačeny správně vykreslené oblasti, tence čárkované oblasti, které nejsou vykresleny a mohou způsobit problémy.

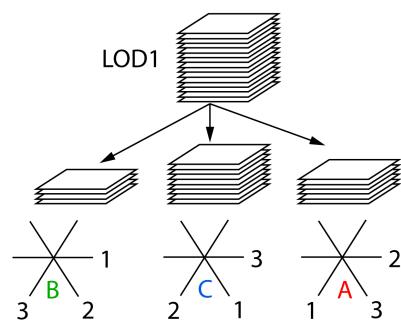
Průhlednost mezi instancemi je ošetřena řazením v rámci zpracování instancí. Geometrická primitiva v rámci jedné instance je ovšem také třeba řadit. Jak je patrné z obrázku 4.13, nelze pouhým řazením geometrických primitiv zajistit zcela korektní vykreslení. Ovšem důležité je si uvědomit skutečnost, že problémy způsobují zejména řezy skoro rovnoběžné s

pohledem (dále jen *rovnoběžný řez*), které jsou stejně skrývány (proto jsou vysoce průhledné). Zejména problematická je pak jejich část blíž k pozorovateli. Část dále od pozorovatele zakrývají řezy méně průhledné natočené víceméně kolmo ke směru pohledu. Stačí tedy zajistit, aby se takový rovnoběžný řez vykresloval až po ostatních. Jak je patrné z obr. 4.14, je třeba



Obrázek 4.14: Různé případy poloh pozorovatele vůči trsu řezů. Všechny možnosti se redukují na 3 případy pořadí vykreslování řezů (A,B,C).

rozlišit, jaký případ nastává a podle toho určit pořadí vykreslení jednotlivých řezů. Fronty instancí se tedy rozpadají ještě podle tohoto kritéria (viz obr. 4.15), neboť různé pořadí vykreslení jednotlivých řezů je zajištěno různým souborem indexů v EBO.



Obrázek 4.15: Rozpad fronty LOD1 do několika menších front podle pořadí vykreslení (A, B, C).

Samotné vykreslování a animace jednotlivých instancí LOD1 a LOD2 probíhá rovněž pomocí speciálního *shaderu* (programu na GPU). Existuje stejný shader jak pro vykreslování instancované, tak i neinstancované geometrie. Odlišnosti chování se projevují pouze ve *vertex shaderu* (prefixy *ia* označují instanční atributy):

```
// color_variance... uniformni parametr, barevna odchylka instance
// time_offset... uniformni parametr, casova odchylka instance
// wind_direction... uniformni parametr, smer vетru
// ia_transform_matrix... instancni atribut, transformacni matice instance
// ia_wind_matrix... instancni atribut, transformacni matice otoceni smeru vетru

if (isInstancingActive){
    // kresli se v modu instancovane geometrie
    color_variance = ia_color_variance.rgb;
    time_offset     = ia_time_offset;

    position = gl_ModelViewMatrix * (ia_transform_matrix * position);
    mat3 T    = gl_NormalMatrix * mat3(ia_transform_matrix);

    tangent   = ( T * tangent );
    normal    = ( T * normal );
    bitangent = ( T * bitangent );

    wind_direction = mat3(ia_transform_matrix) * ia_wind_matrix * wind_direction;
} else {
    // kresli jednotlive instance
    position   = gl_ModelViewMatrix * position;
    tangent    = ( gl_NormalMatrix * tangent );
    normal     = ( gl_NormalMatrix * normal );
    bitangent  = ( gl_NormalMatrix * bitangent );
}
```

Jak již bylo naznačeno v sekci 4.3.3, jsou textury využívané ve *fragment shaderu* slučeny po skupinách. Je tedy nutné v rámci fragment shaderu rozlišit, jaký řez je aktuálně kreslen a podle toho transformovat souřadnice dotazů do takových sloučených textur (mapovat původní rozsah  $\langle 0 - 1 \rangle$  do příslušné oblasti sloučené textury). Informaci o právě kresleném řezu lze předat do fragment shaderu pomocí atributů vrcholů. Stačí použít dvousložkový vektor, kde první složka definuje, která sada řezů je aktivní (z jakého směru) a druhá složka určuje vrstvu. To zajišťuje funkce *sliceCoordsToTextureCoords*:

```
// va_sliceDescription... vertexovy atribut, popisuje právě kreslený řez
// u_sliceCnt... uniformni parametr, pocet rezu z jednoho smera
// u_sliceSetCnt... uniformni parametr, pocet sad rezu (smeru)
vec2 sliceCoordsToTextureCoords(in vec2 sliceCoords){
    return ( clamp ( sliceCoords, 0.0, 1.0 ) + va_sliceDescription)
        / vec2( u_sliceCnt , u_sliceSetsCnt );
}
```

Zatímco animace geometrického modelu probíhá ve vertex shaderu, ohyb větví v LOD1 a LOD2 musí probíhat ve fragment shaderu. Klíčová je funkce `animateBranch`:

```

// position... pozice fragmentu
// time... cas
// offset... identifikator sady rezu
// texCol... prepocetni koeficient do textury
// wood_a... amplituda pohybu
// wood_f... frekvence pohybu
// wind_d... smer vetru
// wind_s... sila vetru
// time... uniformni parametr, pocet sad rezu (smernu)
// time... uniformni parametr, pocet sad rezu (smernu)
// u_sliceSetCnt... uniformni parametr, pocet sad rezu (smernu)

void animateBranch(inout vec2 position, in float branchID, in float time,
                    in float offset, in float texCol, in float wood_a,
                    in float wood_f, in vec3 wind_d, in float wind_s){
    float x, lengthP;
    vec2 mv, corr_s, corr_r, amp, origin, t, amp, fu, fu_deriv;
    vec3 r, s;
    vec4 b_data0, b_data1, b_data2, b_data3;
    // nacteni dat o vetvi z textur
    b_data0 = texture2D(lod_data_tex, vec2((0.5+offset)*texCol, branchID));
    b_data1 = texture2D(lod_data_tex, vec2((1.5+offset)*texCol, branchID));
    b_data2 = texture2D(lod_data_tex, vec2((2.5+offset)*texCol, branchID));
    origin = b_data0.xy;
    mv = b_data0.zw;
    r = b_data1.xyz;
    s = b_data2.xyz;
    t = cross(s,r).xy;
    lengthP = b_data1.w;
    // hodnota x
    x = min(1.0, length(abs(position-origin))/lengthP);
    // vychylka
    amp = vec2(dot(r, wind_d) * wind_s, dot(s, wind_d) * wind_s);
    amp += wood_a * ( texture2D( branch_noise_tex , mv * time * wood_f).rg * 2.0 - vec2(1.0) );
    // ohybova funkce
    float x2 = x * x;
    float fx = 0.374570*x2 + 0.129428*x2*x2;
    float dx = 0.749141*x + 0.517713*x2*x;
    fu = vec2(fx) * amp;
    fu_deriv = vec2( dx / lengthP ) * amp;
    // zamezeni deleni nulou
    fu_deriv = max(fu_deriv, EPSILON) + min(fu_deriv, EPSILON);
    ...
}

```

```

// korekce delky
    vec2 us = sqrt( vec2(1.0) + fu_deriv * fu_deriv );
    vec2 ud = fu / fu_deriv * ( us - vec2(1.0) );
    corr_r = ( t + r.xy * fu_deriv.x ) / us.x * ud.x;
    corr_s = ( t + s.xy * fu_deriv.y ) / us.y * ud.y;
// ohyb - POZOR: zpetna transformace musi pusobit proti puvodni
    position = position - ( fu.x * r.xy + fu.y * s.xy - ( corr_r + corr_s ) );
}

```

Tato funkce je aplikována dvakrát, nejprve na 1. úroveň větví, poté na kmen. Tedy opačně než v případě deformace geometrického modelu. S tím souvisí problém aplikace směrové složky větru. Zatímco správně je deformován nejprve kmen a s ním i souřadné systémy napojených větví a tím pádem se průměr směru větru na větve 1. úrovně mění s deformací kmene. V LOD1 a LOD2 však je aplikována směrová složka větru na větve na neohnutém kmene a teprve poté je ohnut kmen. Tato chyba však nezpůsobuje příliš velký problém a je skoro nepostřehnutelná.

## 4.6 Sezónní barvy a barevné variace

Vegetace (stromy nevyjímaje) podléhá změnám během ročních období. Tato sekce popisuje, jak zajistit změnu barevnosti listů, která je daná roční dobou (sezónou), a jak zajistit, aby se mírně lišila barevnost jednak listů mezi sebou, ale i celková barevnost jednotlivých stromů mezi sebou.

Jak již bylo naznačeno v předchozím textu, sezónní barva je definovaná texturou (viz obr. 4.8) a souřadnicí, ze které se příslušná barva získává. Sezónu definuje parametr `season`. Pro zajištění barevných variací mezi jednotlivými listy je třeba přiřadit listu specifické číslo `leafSpecificNumber`. Toto číslo je třeba distribuovat i do textur používaných pro zobrazení řezů. Využije se k tomu s výhodou 4. složka textury normál. Odchylku barevnosti celých stromů mezi sebou zajistíme podobným mechanismem. Ten je společný i pro zajištění odchylky ve fázi animace. Instanci tak přísluší `instanceSpecificNumber`. Pomocí těchto proměnných lze zajistit odchylku od přímé sezónní barvy:

```

float seasonCoord = season + 0.2*leafSpecificNumber
                    - 0.0001*instanceSpecificNumber;
vec4 seasonColor = texture2D(seasonMap, vec2(0.5, seasonCoord));

```

Koefficienty váhy jednotlivých odchylek byly určeny pouze na základě výsledného vizuálního dojmu a pro různé textury sezónních barev mohou být rozdílné.

Tento metodou lze velmi jednoduše zajistit i přibývání počtu listů na jaře a naopak snižování počtu listů na podzim. Stačí, když je příslušná část textury sezónní barvy průhledná. Pokud je tedy výsledná sezónní barva průhledná, je zpracovávaný fragment zahozen:

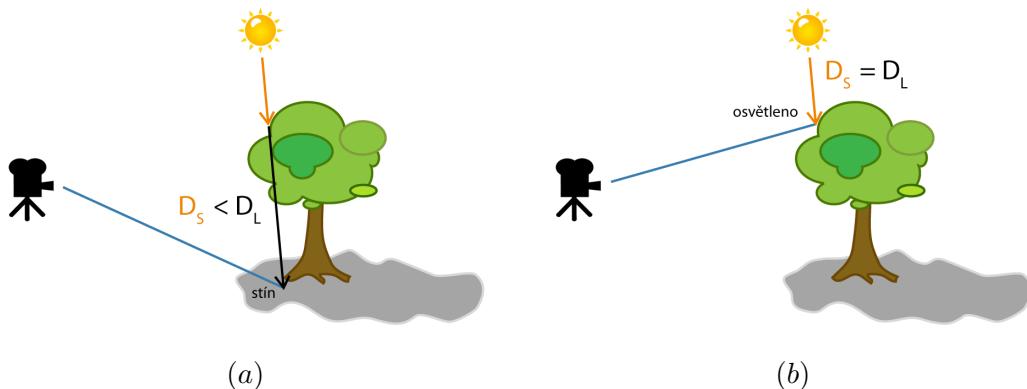
```

if ( seasonColor.a < 0.5 ){
    discard;
}

```

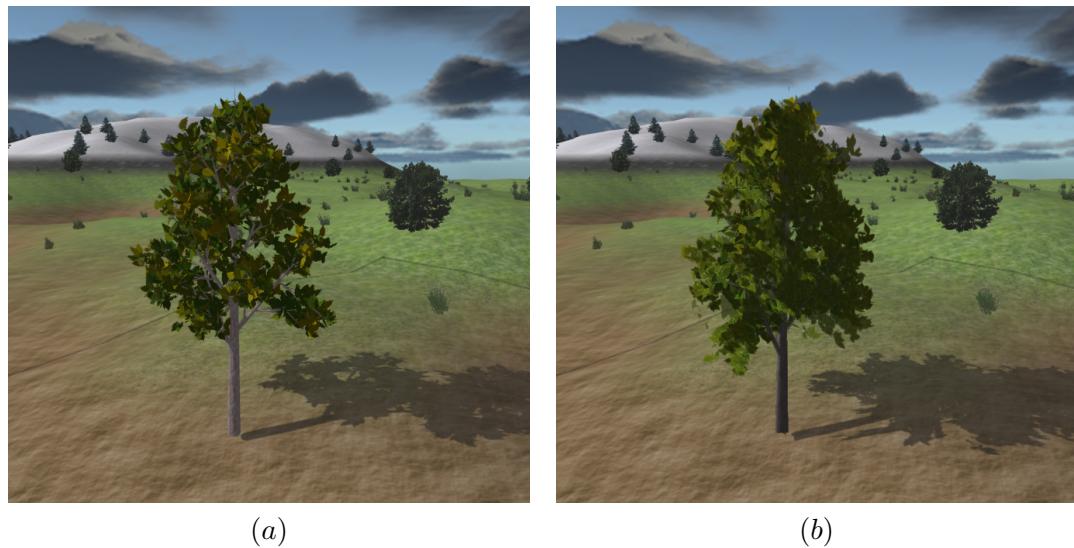
## 4.7 Stíny

Stíny mají velký vliv na vizuální kvalitu generovaného obrazu a jejich věrnost může použitím trsů řezů velmi utrpět. Standardní technikou generující dynamicky stíny je tzv. *shadow-mapping*. Je generována stínová mapa jako záznam vzdálenosti fragmentů od zdroje světla. Následně je pro daný zobrazovaný bod porovnávána vzdálenost od světla  $D_L$  se záznamem na odpovídajícím místě stínové mapy  $D_S$  (viz obr. 4.17).



Obrázek 4.16: Princip shadow mappingu. (a) vzdálenost zobrazovaného bodu ke světlu je větší než záznam ve stínové mapě - bod leží ve stínu, (b) vzdálenosti jsou shodné - bod je osvětlen.

Pokud uvažujeme pouze světla, která nesvítí na trs řezů přímo zvrchu, generuje i tato primitivní konstrukce relativně dobře vypadající stíny. Problém ovšem nastává ve stínech na stromu samotném. Na geometrickém modelu je jasné patrné, že některé části koruny stromu jsou ve stínu. S určitými kompromisy je možné dosáhnout podobného efektu i pro ploché řezy. Klíčové je zapsat do stínové mapy nikoliv přímo hloubku plochy řezu, ale posunout hloubku ještě o příslušnou hodnotu získanou z hloubkové mapy řezu. Stejný princip se uplatňuje i při vyhodnocování, kde je rovněž dobré hodnotu posunout. Hloubku lze do stínové mapy zapsat jednoduše přiřazením do proměnné `gl_FragDepth = hodnota`. Tato operace však může zapříčinit zpomalení celého procesu zobrazení, neboť GPU provádí sérii optimalizací, mezi kterými je i tzv. *early z-test*. Provedení této optimalizační techniky, která probíhá před spuštěním fragment shaderu a případně ho vůbec nespouští, je vázáno na podmínu, že hloubka fragmentu není ve fragment shaderu měněna.



Obrázek 4.17: Porovnání stínů LOD0 (a) a LOD1 (b).

Stíny mohou vytvářet problém i při plynulém přechodu mezi úrovněmi LOD. Jde o to, že zatímco barevný obraz se plynule prolíná pomocí průhlednosti, ve stínové hloubkové mapě žádná průhlednost nefunguje. Děje se tak skoková změna v zápisu do stínové mapy a tím je ovlivněn i barevný obraz jež je skokově jinak stínován. Tento nepříjemný efekt lze částečně potlačit využitím tónování (tzv. *dithering*). Tímto způsobem lze simulovat průhlednost i bez kanálu průhlednosti. Z plochy, která má být průhledná, je odebrán náhodně určitý počet pixelů (jsou plně průhledné), podle hodnoty požadované průhlednosti. Čím má být plocha průhlednější, tím víc pixelů je takto odstaněno. Pokud je touto řídící veličinou proměnná *transition\_control*, pak lze tónování stínové mapy docílit takto:

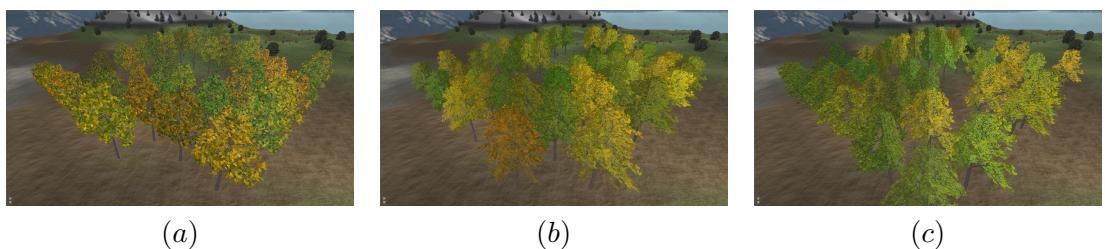
```
// noise... náhodné číslo stejného rozsahu jako transition_control
if ( transition_control < noise ) {
    discard;
}
```



## Kapitola 5

### Testování

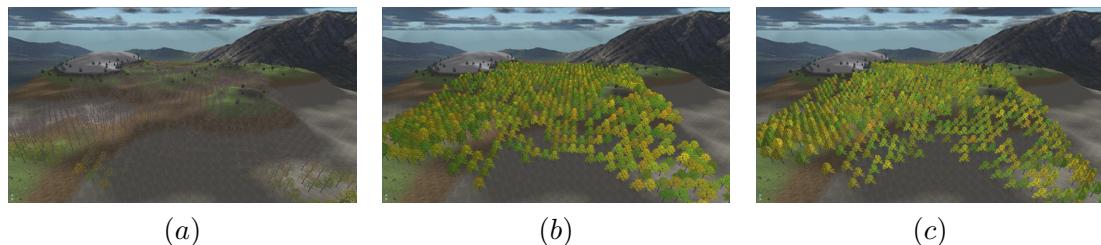
Testování výsledné aplikace probíhalo na sestavě Intel® Core™ i7 CPU 950, frekvence 3.07GHz, 8 logických jader, 12288 MB RAM, GeForce GTX 470/PCI/SSE2. Rozlišení obrazu bylo pro všechny testy fixováno na  $1280 \times 720$  px. Veškeré naměřené hodnoty fps (resp. zobrazovacích časů), jsou brány jako minimum dané veličiny za určitou dobu (typicky 10-20 vteřin). Měření těchto hodnot bylo prováděno pomocí tzv. TIMER \_ QUERY. Jde o mechanismus, který měří čas potřebný pro vykreslení přímo na GPU pomocí Query Object asynchronního dotazu. Hodnota je určována s přesností na nanosekundy, ale uváděné hodnoty (v ms) jsou pro přehlednost zaokrouhleny na menší počet desetinných míst.



Obrázek 5.1: Náhledy z testování, scéna SMALL FOREST s 50 instancemi, (a) LOD0, (b) LOD1, (c) LOD2, 0

Nejprve byla testována kvalita a rychlosť zobrazování různých úrovní LOD. Pro tyto účely byly fixovány dvě scény s různým počtem instancí stromů. Scéna **SMALL FOREST** představuje pohled na menší les (max. 50 instancí). Všechny instance by se měly nacházet v pohledovém prostoru kamery. Podobu scény při zobrazení všech stromů pouze v určitém LOD zachycují obrázky 5.1

Pro testování rozsáhlějších porostů (až 1000 instancí) byla používána scéna **FOREST** (viz obr. 5.2).



Obrázek 5.2: Náhledy z testování, scéna FOREST s 1000 instancemi , (a) LOD0, (b) LOD1, (c) LOD2, 0

Následující tabuľka uvádza hodnoty zobrazovacích časov namēřené pro rôzny počet instancí pri zobrazovaní bez multisamplingu:

#instancí	LOD0 (ms)	LOD1 (ms)	
10	2,98	3,71	scéna SMALL FOREST
25	7,37	8,96	
50	13,71	16,89	
100	20,25	6,54	scéna FOREST
250	48,15	15,59	
500	94,73	30,36	
1000	188,96	57,16	

Tabuľka 5.1: Porovnanie LOD0 a LOD1, bez multisamplingu

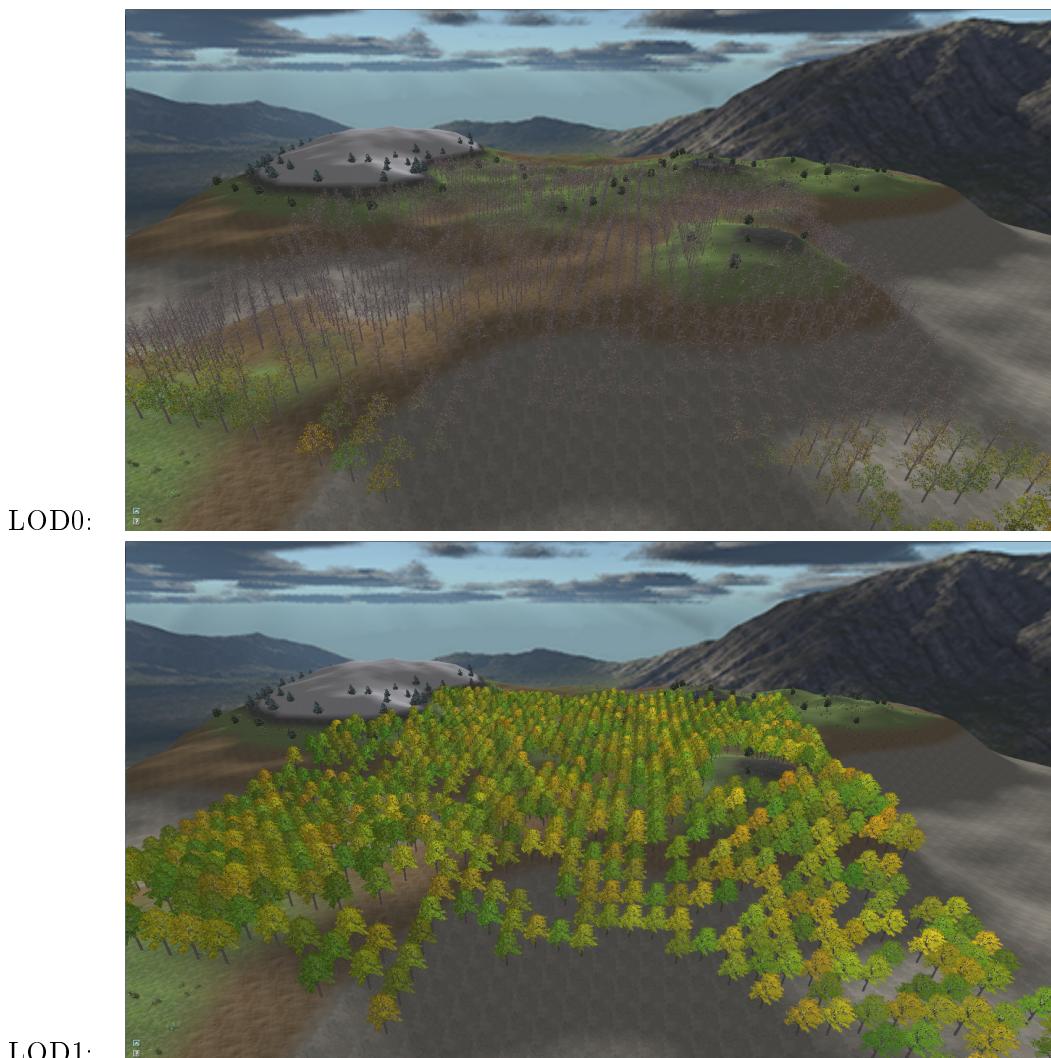
A stejný test pri vykreslování s multisamplingom (4 vzorky na pixel)

#instancí	LOD0 (ms)	LOD1 (ms)	
10	3,72	4,26	scéna SMALL FOREST
25	8,32	9,48	
50	15,94	16,5	
100	19,34	6,04	scéna FOREST
250	47,5	14,36	
500	94,2	28,58	
1000	188,3	55,87	

Tabuľka 5.2: Porovnanie LOD0 a LOD1, 4x multisampling

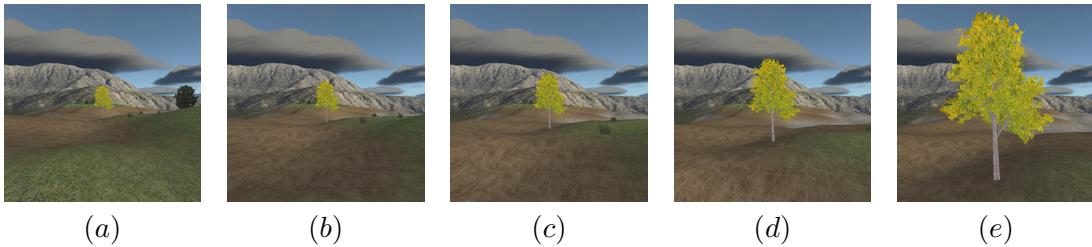
Jak sa ukázalo, zpomalenie zpôsobené zpracováváním více vzorkov je takmer neznatelné. Potvrdil sa i predpoklad, že rýchlosť zobrazovania LOD0 závisí predevším na počte zpracovávaných vrcholov (a tedy na počtu stromov), zatímczo doba potrebná k vykresleniu LOD1 závisí predevším na počte fragmentov, do ktorých je kresleno. Zatímczo zobrazovací čas LOD0 s príbývajúcimi instancemi roste a je víceméně jedno, jak je scéna usporiadána, hraje pro

LOD1 uspořádání scény důležitou roli, neboť **zobrazení 50 instancí ve scéně SMALL FOREST trvá déle než zobrazení pětinásobného počtu (250) instancí ve scéně FOREST**. Záleží na výsledném pokrytí obrazu pixely instancí LOD1. Podstatné je ovšem zjištění, že pro malé scény je efektivnější využít LOD0, zatímco pro pohledy na rozsáhlé scény se již projevují přednosti a efektivita LOD1, které je pro 1000 instancí v testovaných scénách zhruba **3×efektivnější**. Jak se navíc ukázalo, je velice zásadní rozdíl při pohledu na instance LOD0 a LOD1 z větší vzdálenosti. Instance zobrazované v LOD0 ztratí listy a stanou se tak těžko použitelnými (viz porovnání na obr. 5.3). Tento jev souvisí s použitím částečně průhledných textur pro zobrazení jednotlivých listů. Při pohledu z dálky je taková textura vzorkována tak nešťastně, že se stává kompletně průhlednou. Naproti tomu, instance LOD1 si zachovávají svůj vzhled a co se týče kvality tak převyšují pro pohledy z dálky instance LOD0.



Obrázek 5.3: Náhledy z testování, scéna FOREST s 1000 instancemi, porovnání výsledné kvality

Silnou závislostí zobrazovacího času na počtu fragmentů se zabývá následující test. Je vytvořena scéna s jedinou instancí v počátku. Následně je měřen zobrazovací čas pro pohledy z různých vzdáleností. Využívá se vlastností perspektivní projekce, kdy vzdálenější objekty se jeví menší - a tedy jsou vykreslovány do menšího počtu fragmentů. Měření probíhalo bez multisamplingu a počet vykreslovaných fragmentů byl zaznamenáván pomocí Query Object čítače zpracovávaných fragmentů `GL_SAMPLES_PASSED`. Jednotlivé pohledy na strom zachycuje sada obrázků 5.5.

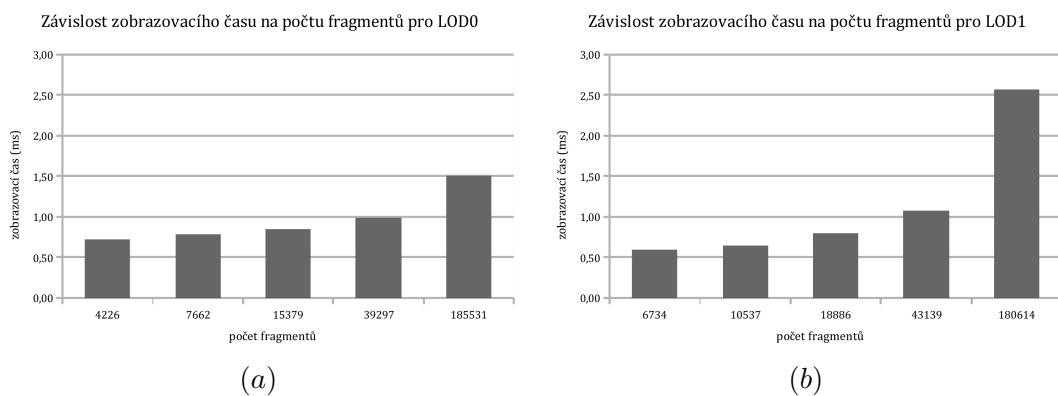


Obrázek 5.4: Náhledy z testování, scéna FRAGMENTS, (a) vzdálenost 50, (b) vzdálenost 40, (c) vzdálenost 30, (d) vzdálenost 20, (e) vzdálenost 10

Následující tabulka a grafy zprostředkovávají naměřené hodnoty pro vzdálenosti 10, 20, 30, 40 a 50 jednotek (velikost stromu je 10 jednotek).

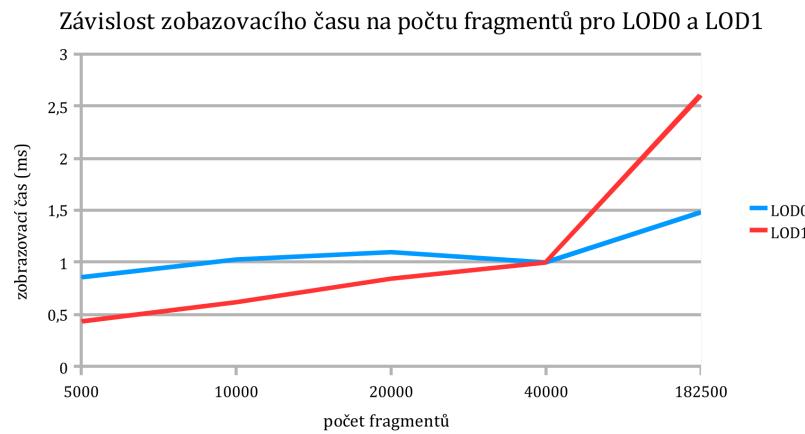
vzdálenost	LOD0			LOD1		
	# fragmentů	fps	čas (ms)	# fragmentů	fps	čas (ms)
50	4226	1391,3	0,71875	6734	1693,29	0,59
40	7662	1268,64	0,78825	10537	1538,84	0,65
30	15379	1183,22	0,84515	18886	1263,75	0,79
20	39297	1018,22	0,98211	43139	928,04	1,08
10	185531	665,14	1,50345	180614	389,14	2,57

Tabulka 5.3: Závislost zobrazovacího času na počtu fragmentů.



Obrázek 5.5: Grafy závislosti zobrazovacího času na počtu fragmentů

Vzájemná souvislost je pak dobře patrná z grafu na obr. 5.6



Obrázek 5.6: Graf závislosti zobrazovacího času na počtu fragmentů pro LOD0 a LOD1.

Je dobře patrné, že zatímco pro LOD0 je závislost na počtu fragmentů relativně nízká, pro LOD1 představuje silnou závislost. Pokud počet zpracovávaných fragmentů přesáhne hranici 40000, pak je zobrazování LOD1 časově náročnější než zobrazování LOD0.

Podíl jednotlivých LOD na výsledném zobrazovacím čase se snaží odhalit následující test. Scéna je zafixována (50 instancí stromů), jsou zafixované i vzdálenosti, ve kterých dochází k přechodu mezi LOD (viz obr. 5.7). Poté je scéna zobrazena nejprve bez stromů. Poté jsou zobrazeny pouze stromy v LOD0, následně jsou zobrazeny stromy v LOD1 a nakonec i v LOD2.

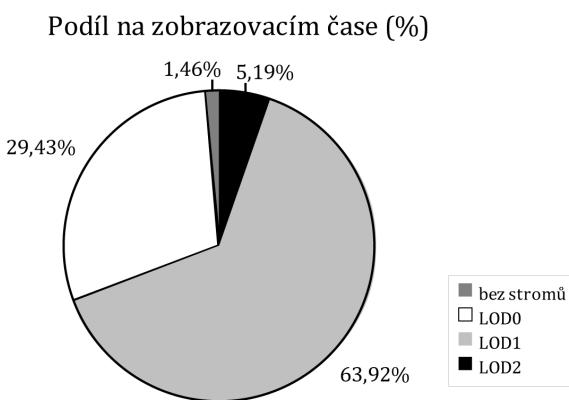


Obrázek 5.7: Náhled testovací scény podílu LOD na zobrazení čase.

Výsledky měřená zaznamenává tabulka 5.4 a graf 5.8:

	fps	čas (ms)	rozdíl časů	%
bez stromů	2384,76	0,42	0,42	1,46
LOD0	112,64	8,88	8,46	29,43
+LOD1	36,7	27,25	18,37	63,92
+LOD2	34,79	28,74	1,49	5,19

Tabulka 5.4: Podíly jednotlivých LOD na výsledném čase zobrazení



Obrázek 5.8: Graf podílů jednotlivých LOD na zobrazovacím čase

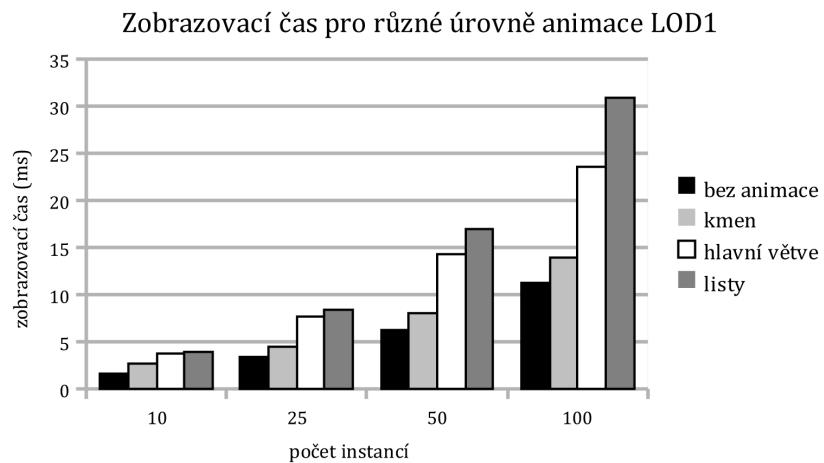
Je jasné patrné, že pro tuto scénu je časově nejnáročnější vykreslování instancí v LOD2, které z celkové doby 28,74 ms zabere více než polovinu (18,37 ms).

Další měření se týká toho, jak podrobnost animace LOD1 a LOD2 ovlivňuje časové nároky. Pro fixní scénu s různým počtem instancí stromů byl měřen zobrazovací čas pro různě podrobné animace. Byla změřena statická verze bez animace a následně byly přidávány úrovně animací. Nejprve pouze pohyb celého stromu (kmene), poté se přidaly i větve 1. úrovně. Nakonec, aby byla animace kompletní, byla měřena i plná verze LOD1 (resp. LOD2), včetně pohybu listů. Výsledky měření obsahuje tabulka 5.5.

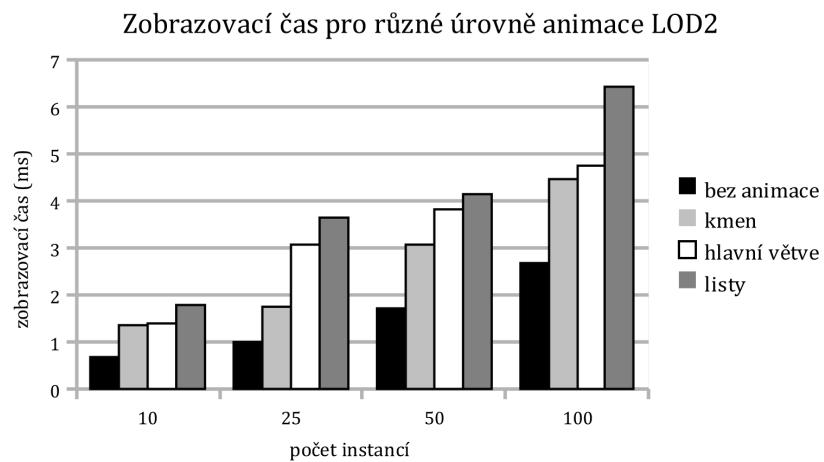
	LOD1				LOD2			
	# instancí 10	# instancí 25	# instancí 50	# instancí 100	# instancí 10	# instancí 25	# instancí 50	# instancí 100
bez animace	1,55	3,39	6,35	11,31	0,68	1	1,73	2,68
+ kmene	2,64	4,51	8,14	13,96	1,35	1,74	3,07	4,47
+ hlavní větve	3,84	7,73	14,22	23,53	1,39	3,06	3,83	4,75
+ listy	4	8,45	16,91	30,93	1,79	3,66	4,14	6,44

Tabulka 5.5: Vliv podrobnosti animace na zobrazovací čas.

Následující grafy (5.9 a 5.10) dávají tyto hodnoty do vzájemných souvislostí.



Obrázek 5.9: Graf závislosti zobrazovacího času na podrobnosti animace LOD1.



Obrázek 5.10: Graf závislosti zobrazovacího času na podrobnosti animace LOD2.

Poslední test se zabývá tím, jaký vliv má rozlišení stínové mapy na zobrazovací čas pro různé instance. Měření probíhalo pouze s jedinou instancí, a fixovaným světelným zdrojem a scénou. Byly měřeny zobrazovací časy LOD0 a LOD1, přičemž LOD1 pro dva případy - s přepisem hloubky při zápisu do hloubkové textury a bez něj. Scéna byla vykreslována bez multisamplingu jak na obrazovku, tak do stínové mapy. Výsledky měření lze vyčíst z tabulky 5:

rozlišení stínové mapy	zobrazovací čas (ms)		
	LOD0	LOD1 (přepis hloubky)	LOD1 (bez přepisu)
512 × 512	2,26	3,21	3,28
1024 × 1024	2,44	3,08	3,02
2048 × 2048	2,75	5,98	5,74

Tabulka 5.6: Vliv rozlišení stínové mapy na zobrazovací čas

V zásadě se potvrdilo, že s rostoucím rozlišením se snižuje i rychlosť vykreslování. Pro LOD1 je však zpomalení znatelnější. Rozdíl mezi postupem, kdy se přepisuje hloubka zapisovaná do stínové mapy, se pro danou scénu skoro neprojevil. Zajímavé ovšem je, že LOD1 se vykresluje rychleji při použití stínové mapy s rozlišením 1024×1024 px než při rozlišení 512×512 px. Toto chování lze zřejmě příčist na vrub rozdílné optimalizaci ze stany GPU při použití různých rozlišení. Odhalení konkrétní příčiny by vyžadovalo další studium tohoto jevu.

V rámci testování byla zachycena z okna aplikace videa, která dokazují, že systém pracuje na testovací sestavě v reálném čase. Tato videa lze nalézt na přiloženém datovém nosiči.

# Kapitola 6

## Závěr

V rámci této práce byly v sekci 2.2( Stávající řešení) zmíněny různé metody zobrazování vegetace - zejména stromů. Následně byl problém zobrazování pohybujících se stromů analyzován na teoretické úrovni. Byl odvozen model ohybu větví a listů působením větru, který lze uplatnit při zobrazování na GPU. Rovněž byla rozebrána problematika zobrazování listů včetně pokročilého modelu, který bere v úvahu i průsvitnost. Na základě prostudovaných metod zobrazování stromů byl navržen diskrétní systém LOD se 3 úrovněmi. Zatímco nejpodrobnější úroveň (LOD0) je vykreslována pomocí detailní geometrie, další dvě úrovně jsou zobrazovány jako trs plochých textur (řezů). **Hlavní přínos této práce je však představení nové metody zpětné transformace, díky níž je možné simulovat pohyb vegetace na úrovni jednotlivých větví v rámci jednotlivých řezů.** Tato metoda je koherentní s animací trojrozměrné geometrie a pracuje zcela na GPU a umožňuje tak vykreslovat rozsáhlejší vegetační celky v reálném čase. Rovněž je navržen postup, kterým lze jednoduše reflektovat sezonné změny vegetace.

Navržené metody byly implementovány a byla vytvořena interaktivní aplikace. Následně byla tato aplikace otestována z hlediska efektivity zobrazování jednotlivých úrovní LOD i z hlediska kvality zobrazení a různých úrovní realističnosti simulace. Na základě výsledků testování lze konstatovat, že se povedlo splnit cíle vytyčené na počátku této práce.

Následující část navrhne možné směry a způsoby vylepšení.

### 6.1 Možnosti vylepšení a dalšího rozvoje

Problematika realistického zobrazování vegetace v real-time je velice obsáhlé téma a jeho kompletní pokrytí mnohonásobně přesahuje rozsah této práce. Se vší skromností je třeba přiznat, že tato práce se do zmíněné problematiky vydává pouze na jakousi rychlou a letmou exkurzi. Stejně tak je třeba konstatovat, že je v této sféře stále velké množství neprozoumaných a velmi zajímavých oblastí. Namátkou lze zmínit věrný a zároveň obecný deformační a destrukční model stromů pracující v reálném čase.

#### Uložení předzpracovaných dat

V rámci zrychlení fáze inicializace aplikace, kdy jsou vstupní soubory zpracovávány, bylo možné tato předzpracovaná uložit a poté načítat už rovnou pouze potřebný a zpracovaný soubor informací. Zároveň by bylo možné případně do některých dat zasáhnout a tvůrčím

způsobem je ovlivnit (nabízí se to u textury expandovaných dat pro jednotlivé řezy). Mohlo by se tím dosáhnout kvalitativně lepších výsledků.

### Preciznější frustum culling

Současná aplikace řeší otázku frustum cullingu velmi primitivním způsobem. Vyřazuje všechny instance LOD1 a LOD2, které jsou za rovinou pohledu. Všechny další operace využívající frustum culling probíhají pouze v rámci standardního zpracování na GPU. Bylo by jistě možné aplikovat pokročilou metodu, která by ušetřila zpracování instancí, které ve výsledku stejně nejsou vidět. Lze předpokládat, že své využití by nalezly i Occlusion Queries.

### Efektivnější řízení LOD

Zatímco popsaná metoda řídí LOD jednotlivě pro každou instanci, zajímavé by mohly být metody řízení LOD po skupinách. Lze předpokládat použití akceleračních struktur jako je kD-tree, či BVH. Otevřenou otázkou je v takovém případě řazení kvůli průhlednosti.

### Optimalizace počtu vykreslovaných fragmentů LOD

Nyní se díky back-to-front řazení neuplatňuje žádná optimalizace díky z-testům. Všechny fragmenty jsou vykreslovány, ačkoliv velká část není stejně vidět. Tento postup jde totiž bohužel přímo proti efektivnímu využití z-bufferu. Ideální by byl postup v přesně opačném pořadí s tím, že pokud již na daném místě nemůže fragment přispět svou barvou kvůli neprůhlednosti fragmentů bližších, tak by bylo zpracování ukončeno. Jelikož je efektivita LOD1 a LOD2 kriticky závislá na počtu vykreslovaných fragmentů, byla by tato optimalizace velmi vítanou a projevila by se ve výrazném zvýšení efektivity vykreslování.

**Jiná forma LOD** Není zcela bez perspektivy zabývat se i myšlenkou využití jiné formy LOD než je trs řezů popsaný v této práci. Zajímavé výsledky by mohlo přinést aplikování metody zpětné transformace na billboard-clouds (billboardové mraky) resp. na low-poly modely, případně na jiný způsob zobrazení ve snížené úrovni detailu.

**Stíny** Zejména přechod mezi stíny dvou úrovní by bylo možné ošetřit sofistikovanějším způsobem. Zatímco představený přístup řeší tento problém pomocí ditheringu, lze si představit i řešení na základě multisamplingu stínové mapy. Další alternativou jsou stochastické přístupy.

**Zvýšení detailnosti LOD0** Výsledná aplikace abstrahuje z modelu vlastně pouze topologii a větve jsou nahrazeny relativně primitivní geometrií. V zásadě nic nebrání tomu upravit aplikaci tak, aby pracovala s obecnými (i velmi detailními) modely se známou topologií. Ohybová metoda však předpokládá větve, které jsou v základní poloze přímé. Problémy by však mohly tvořit větve zahnuté, či jinak pokroucené, neboť by mohlo docházet k viditelnému natahování či naopak zkracování. Tento problém si proto vyžaduje další studium. Podobně by bylo možné využít technik bump-mappingu či parallax-mappingu pro přidání detailnosti kmene a větví. Další možností je tzv. *Silhouette Clipping*, který obohacuje siluetu kmene a větví o případné detaily.

**Order-independent průhlednost** Průhlednost nyní způsobuje mnoho komplikací s řazením instancí i primitiv, zejména by bylo výhodné zpracování průhlednosti optimalizovat vzhledem k z-testům. Řešení průhlednosti ovlivňuje v aplikaci celou architekturu LOD systému. Toto zpracování by mohlo být ušetřeno.

**Ambient occlusion** Jak se ukazuje, ambient occlusion (zastínění světelných příspěvků prostředí) může podstatným způsobem zlepšit vizuální dojem. Přesné a korektní ambient occlusion je v reálném čase velmi obtížné řešit. Zejména jde o efekt, kdy listy uvnitř koruny

stromu se jeví tmavší, než listy poblíž okraje koruny. Řešením by mohlo být předzpracování a určení příslušného koeficientu pro každý list.

**Barevné stínové mapy** Reálné stíny přebírají barvu listů, jimiž světlo prošlo. V aplikaci jsou brány v úvahu sice průsvitné listy, ovšem stíny jsou vlastně monochromatické a tuto průsvitnost vůbec nereflektují. Na základě článku [18] by bylo možné implementovat techniku, která by brala v potaz právě průsvitnost listů a ovlivňovala by tak barvu stínů.



# Literatura

- [1] OpenGL SDK, Documentation, December 2011. Dostupné z: <<http://www.opengl.org/sdk/docs/>>.
- [2] BARTA, P. – KOVÁCS, B. Order Independent Transparency with Per-Pixel Linked Lists. CESCG. Dostupné z: <<http://www.cg.tuwien.ac.at/hostings/cescg/CESCG-2011/>>.
- [3] BOUSQUET, L. et al. Leaf BRDF measurements and model for specular and diffuse components differentiation. *Remote Sensing of Environment*. 2005, 98, 2-3, s. 201 – 211. ISSN 0034-4257. doi: 10.1016/j.rse.2005.07.005. Dostupné z: <<http://www.sciencedirect.com/science/article/pii/S0034425705002324>>.
- [4] COOK, R. L. – TORRANCE, K. E. A reflectance model for computer graphics. *SIGGRAPH Comput. Graph.* August 1981, 15, s. 307–316. ISSN 0097-8930. doi: <http://doi.acm.org/10.1145/965161.806819>. Dostupné z: <<http://doi.acm.org/10.1145/965161.806819>>.
- [5] DECAUDIN, P. AntTweakBar, December 2011. Dostupné z: <<http://www.antisphere.com/Wiki/tools:anttweakbar>>.
- [6] DECAUDIN, P. – NEYRET, F. Rendering Forest Scenes in Real-Time. In KELLER, A. – JENSEN, H. W. (Ed.) *15th Eurographics Workshop on Rendering, Rendering Techniques'04, June, 2004*, s. 93–102, Norköping, Suède, 2004. Eurographics Association.
- [7] FUHRMANN, A. L. – UMLAUF, E. – MANTLER, S. Extreme Model Simplification for Forest Rendering . s. 57–66, 2005. doi: 10.2312/NPH/NPH05/057-066. Dostupné z: <<http://www.eg.org/EG/DL/WS/NPH/NPH05/057-066.pdf>>.
- [8] GARCÍA, I. et al. Multi-layered indirect texturing for tree rendering. In D. EBERT, S. M. (Ed.) *Eurographics Workshop on Natural Phenomena*, 2007. Dostupné z: <<http://iiia.udg.es/GGG/publicacions>>.
- [9] GEELNARD, M. – BERGLUND, C. GLFW - An OpenGL library, December 2011. Dostupné z: <<http://www.glfw.org>>.
- [10] GIEGL, M. – WIMMER, M. Unpopping: Solving the Image-Space Blend Problem for Smooth Discrete LOD Transitions, March 2007. ISSN 0167-7055. Dostupné z: <<http://www.cg.tuwien.ac.at/research/publications/2007/GIEGL-2007-UNP/>>.

- [11] GILET, G. – MEYER, A. – NEYRET, F. Point-based rendering of trees. In GALIN, E. – POULIN, P. (Ed.) *Eurographics Workshop on Natural Phenomena 2000, August, 2005*, s. 67–72, Dublin, Irlande, 2005. ACM/SIGGRAPH, A.K. Peters.
- [12] HABEL, R. – KUSTERNIG, A. – WIMMER, M. Physically Based Real-Time Transparency for Leaves, June 2007. Dostupné z: <[http://www.cg.tuwien.ac.at/research/publications/2007/Habel\\_2007\\_RTT/](http://www.cg.tuwien.ac.at/research/publications/2007/Habel_2007_RTT/)>.
- [13] HABEL, R. – KUSTERNIG, A. – WIMMER, M. Physically Guided Animation of Trees, March 2009. ISSN 0167-7055. Dostupné z: <[http://www.cg.tuwien.ac.at/research/publications/2009/Habel\\_09\\_PGT/](http://www.cg.tuwien.ac.at/research/publications/2009/Habel_09_PGT/)>.
- [14] IKITS, M. – MAGALLON, M. GLEW: The OpenGL Extension Wrangler Library, December 2011. Dostupné z: <<http://glew.sourceforge.net>>.
- [15] JAKULIN, A. Interactive Vegetation Rendering with Slicing and Blending. In SOUSA, J. T. (Ed.) *Eurographics 2000: Short presentations*, 2000. Dostupné z: <<http://www.stat.columbia.edu/~jakulin/slicing-and-blending/trees-electronic.pdf>>.
- [16] Lode Vandevenne. LodePNG, December 2011. Dostupné z: <<http://lodev.org/lodepng>>.
- [17] MANTLER, S. – TOBLER, R. F. – FUHRMANN, A. L. The State of the Art in Realtime Rendering of Vegetation, 2003.
- [18] MCGUIRE, M. – ENDERTON, E. Colored Stochastic Shadow Maps. February 2011. Dostupné z: <<http://research.nvidia.com/publication/colored-stochastic-shadow-maps>>.
- [19] Přispěvatelé desktopwallpaper-s.com. [http://desktopwallpaper-s.com/19-Computers/~/Green\\_leaves\\_HD\\_wallpaper/](http://desktopwallpaper-s.com/19-Computers/~/Green_leaves_HD_wallpaper/), stav z 14. 12. 2011.
- [20] REBOLLO, C. et al. Fast rendering of leaves. In *Proceedings of the Ninth IASTED International Conference on Computer Graphics and Imaging*, CGIM '07, s. 46–53, Anaheim, CA, USA, 2007. ACTA Press. Dostupné z: <<http://dl.acm.org/citation.cfm?id=1710707.1710717>>. ISBN 978-0-88986-645-4.
- [21] RENé TRUELSEN, M. B. Visualizing procedurally generated trees in real time using multiple levels of detail, 2008. Dostupné z: <<http://image.diku.dk/projects/media/truelsen.bonding.08.pdf>>.
- [22] The Khronos Group Inc. GLUT- The OpenGL Utility Toolkit, December 2011. Dostupné z: <<http://www.opengl.org/resources/libraries/glut/>>.
- [23] UMLAUF, E. J. Image-Based Rendering of Forests. Master's thesis, Vienna University of Technology, 2004.

# Seznam obrázků

2.1	Znázornění příkladu úrovní detailu.	4
2.2	Možnosti realistického zobrazování vegetace	5
2.3	Obrazy různých stupňů detailu téhož modelu	6
2.4	Znázornění problémů měkkého přechodu mezi diskrétními úrovněmi detailu	6
2.5	Různé úrovně detailu tvořené billboardy a jejich porovnání	7
2.6	Reprezentace korun stromů pomocí řezů	7
2.7	Různé přístupy k billboardům	8
2.8	Volumetrické zobrazování využívající rovnoběžných textur	9
2.9	Reprezentace listů jednotlivými body	9
3.1	Grafická pipeline	12
3.2	Fyzikální model pružného prutu (větve).	12
3.3	Korekce délky ohybové funkce	14
3.4	Původní ohybová funkce	15
3.5	Souřadný systém větve	15
3.6	Vyjádření hierarchie	16
3.7	Souřadný systém větve a jeho transformace při ohybu nadřazené větve	17
3.8	Transformace souřadných systémů při ohybu	17
3.9	Souřadný systém listu	18
3.10	Dvě generované šumové funkce	20
3.11	Model 3D turbulentního pole	20
3.12	Fotografie průsvitných listů	21
3.13	Situace na povrchu listu	22
3.14	Fotografie skutečných listů, barevné variace v ploše listu	24
3.15	Zdrojové textury pro výpočet osvětlení listů	25
3.16	Konstrukce jednoduchého trsu billboardů	26
3.17	Tvorba řezů pro vícevrstvé billboardy	27
3.18	Schematicky znázorněné oblasti, kam se mohou větve deformovat	28
3.19	Myšlenkový model ohybu pomocí zpětné transformace	29
3.20	Zjištování nejbližšího bodu na ohybové křivce	29

3.21 Mnohoznačnost při zpětné transformaci . . . . .	30
3.22 Postup zpětné transformace . . . . .	30
3.23 Dvourozměrná deformace obrazu . . . . .	32
3.24 Znázornění zón, kde se instance stromů vykreslují danou metodou . . . . .	33
3.25 Různé přechody mezi LOD . . . . .	34
4.1 Ukázka výsledného zobrazení . . . . .	37
4.2 AntTweakBar, jednoduché GUI . . . . .	38
4.3 Různé druhy pamětí . . . . .	39
4.4 Fronty instancí . . . . .	40
4.5 Zdrojová textura barvy kmene a větví . . . . .	43
4.6 Šumové textury animace větví a listů . . . . .	43
4.7 Ukázka z datové textury pro LOD0 . . . . .	44
4.8 Ukázka textury sezónních barev . . . . .	44
4.9 Mechanismus FBO v OpenGL . . . . .	45
4.10 Ukázka expanze dat . . . . .	47
4.11 Slučování textur . . . . .	47
4.12 Ukázka části datové textury pro LOD1 . . . . .	48
4.13 Průhlednost trsu řezů . . . . .	51
4.14 Variace pořadí vykreslování trsu řezů . . . . .	52
4.15 Pořadí vykreslování LOD1 . . . . .	52
4.16 Princip shadow mappingu . . . . .	56
4.17 Stíny v LOD0 a LOD1 . . . . .	57
5.1 Náhledy testovací scény . . . . .	59
5.2 Náhledy testovací scény . . . . .	60
5.3 Náhledy testovací scény FOREST . . . . .	61
5.4 Náhledy testovací scény . . . . .	62
5.5 Grafy závislosti zobrazovacího času na počtu fragmentů . . . . .	62
5.6 Graf závislosti zobrazovacího času na počtu fragmentů pro LOD0 a LOD1 . .	63
5.7 Náhled testovací scény podílu LOD na zobr. čase . . . . .	63
5.8 Graf podílu jednotlivých LOD na zobrazovacím čase . . . . .	64
5.9 Graf závislosti zobrazovacího času na podrobnosti animace LOD1 . . . . .	65
5.10 Graf závislosti zobrazovacího času na podrobnosti animace LOD2 . . . . .	65

# Seznam tabulek

3.1	Data potřebná k výpočtu hierarchické deformace větve . . . . .	18
4.1	Minimální sada dat uložená v datové textuře pro LOD0. . . . .	44
4.2	Minimální sada dat uložená v datové textuře pro LOD1 a LOD2. . . . .	48
5.1	Porovnání LOD0 a LOD1, bez multisamplingu . . . . .	60
5.2	Porovnání LOD0 a LOD1, 4x multisampling . . . . .	60
5.3	Závislost zobrazovacího času na počtu fragmentů. . . . .	62
5.4	Podíly jednotlivých LOD na výsledném čase zobrazení . . . . .	64
5.5	Vliv podrobnosti animace na zobrazovací čas. . . . .	64
5.6	Vliv rozlišení stínové mapy na zobrazovací čas . . . . .	66



## Příloha A

### Seznam použitých zkratek

**2D** Two-Dimensional

**3D** Three-Dimensional

**BRDF** Bidirectional Reflectance Distribution Function

**BSSRDF** Bidirectional Surface Scattering Reflectance Distribution Function

**BVH** Bounding Volume Hierarchy

**CPU** Central Processing Unit

**FBO** Frame Buffer Object

**fps** Frames Per Second

**GPU** Graphics Processing Unit

**kD-tree** k-Dimensional tree

**LOD** Level Of Detail

**OpenGL** Open Graphic Library



## Příloha B

# Instalační a uživatelská příručka

Tato příloha velmi žádoucí zejména u softwarových implementačních prací.



## Příloha C

### Obsah přiloženého DVD

DVD

- └─ BIN ..... binární soubor pro platformu Windows, příslušná data a knihovny
- └─ IMAGES ..... screenshoty z aplikace
- └─ SRC ..... zdrojové kódy aplikace, příslušná data a knihovny
- └─ TEXT ..... co vlastne ten adresar obsahuje
  - └─ PDF .... text diplomové práce ve formátu PDF, optimalizovaný pro různé rozlišení
  - └─ TEX ..... zdrojové soubory a data pro sazbu textu systémem L<sup>A</sup>T<sub>E</sub>X
- └─ TOOLS ..... nástroje spojené s aplikací
- └─ VIDEO ..... videa
  - └─ CAPTURED ..... zaznamenaný generovaný obraz aplikace
  - └─ REFERENCE ..... referenční videa reálných stromů