VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

RENDEROVÁNÍ VEGETACE

VEGETATION RENDERING

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **MICHAL KLIŠ**
AUTHOR

VEDOUCÍ PRÁCE **Ing. TOMÁŠ STARKA,**
SUPERVISOR

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačové grafiky a multimédií (UPGM) Akademický rok 2020/2021

Zadání bakalářské práce

Student: **Kliš Michal**
Program: Informační technologie
Název: **Renderování vegetace**
Vegetation Rendering
Kategorie: Počítačová grafika

Zadání:

1. Nastudujte techniky renderování vegetace (osvětlovací modely, animace)
2. Navrhněte algoritmy pro vizualizaci vegetace (křoví, stromy, tráva)
3. Vytvořte aplikaci na vizualizaci vegetace na základě navržených algoritmů.
4. Vytvořte krátké video prezentující práci.

Literatura:

- Po dohodě s vedoucím.

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 a 2, a část 3.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Starka Tomáš, Ing.**
Vedoucí ústavu: Černocký Jan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 12. května 2021
Datum schválení: 30. října 2020

Abstrakt

Cílem této práce je návrh a implementace algoritmů pro zjednodušení složitých polygonálních modelů vegetace tak, aby bylo možné zobrazit tyto modely ve velkém množství bez přílišné ztráty jejich vizuálního vzhledu. Práce se konkrétně zabývá jak modelováním již minimalistických modelů rostlin, tak algoritmy pro jejich finální zjednodušení. Mezi tyto algoritmy konkrétně patří metoda převodu polygonálního modelu na objemovou mapu a převod modelu na pláty. Kromě zjednodušení modelů se práce také zabývá jejich zobrazením v reálném čase a jejich osvětlením.

Abstract

The aim of this work is the design and implementation of algorithms to simplify complex polygonal models of vegetation so that it is possible to display these models in large numbers, without excessive loss of their visual appearance. The work specifically deals with both the modeling of already minimalist plant models and algorithms for their final simplification. Specifically, these algorithms include the method of converting a polygonal model to a volume map and converting the model to plates. In addition to simplifying the models, the work also deals with their display in real time and lighting.

Klíčová slova

Počítačová grafika, vegetace, voxelizace, objemová mapa, plátování, vykreslování, aproximace, modelování.

Keywords

Computer graphics, vegetation, voxelization, volume map, plating, rendering, approximation, modeling

Citace

KLIŠ, Michal. *Renderování vegetace*. Brno, 2021. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Tomáš Starka,

Renderování vegetace

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Tomaše Starky. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Michal Kliš
7. května 2021

Poděkování

Děkuji Vám pane Starko za odbornou pomoc, pevné nervy i za to, že jste si na mě vždy našel čas. Také chci poděkovat mým rodičům, bez jejichž materiální i osobní podpory, bych nebyl schopen tuto práci dokončit.

Obsah

Kapitola 1

Úvod

Dnešní grafické karty nabízí velmi rozsáhlé možnosti. Jsou schopné zobrazit miliony polygonů v reálném čase, které mohou tvořit téměř jakýkoliv model objektu z reálného světa. Taktéž dokáží nad těmito polygony vykonat složité matematické operace, díky kterým model vypadá téměř reálně. Nicméně i dnešní grafické karty mají své limity. Modely stromů, keřů a trav byly vždy jak pro programátory, tak pro grafiky problémem. Pokud je potřeba znázornit polygonálním modelem například auto, několikamilionový polygonální strop určitě bude stačit. Pokud je však potřeba znázornit například strom z reálného světa, velmi rychle se dostaví limity i u těch nejvýkonnějších grafických karet. Každý jehličnatý strom se skládá ze stovek až tisíců větví, na kterých roste nespočet jehlic. Tento počet objektů není možné zobrazit na žádné dnešní grafické kartě v reálném čase. Navíc je nutné přihlédnout i na fakt, že každý strom se pohybuje, a uložit informaci o jeho pohybu by bylo velmi paměťově náročné.

Právě z tohoto důvodu existuje řada metod, které se snaží složitý model napodobit něčím jednodušším (aproximovat). S aproximací se počítá již při tvorbě modelu. Jedna z velmi rozsáhlých metod používaných při modelování je tzv. zapékání. Touto metodou se práce bude zabývat podrobněji, protože je důležitá při osvětlování modelu. Přestože je k dispozici takto zjednodušený model a lze jej díky tomu zobrazit v reálném čase, není toto řešení dostatečné. V případě, kdy je místo zobrazení jednoho modelu stromu potřeba těchto modelů zobrazit více (například les, který tvoří desetitisíce stromů), se brzy opět dostaví výpočetní limity grafických karet. Právě z tohoto důvodu je nutné opět aproximovat původně již aproximovaný model něčím, co je ještě jednodušší na výpočet. K tomuto účelu je v této práci probrána a implementována metoda převodu modelu na objem a metoda prolínání statických billboardů. Správným výběrem z daných metod je dosaženo zobrazení několika tisíců až milionů rostlin i s dostatečným zachováním jejich vzhledu.

Kromě příslušných metod k zjednodušení modelu je nutné mít způsob výběru použití těchto metod. Ne každou metodu je totiž vhodné použít ve všech situacích. Právě na správném výběru metod závisí výsledný vzhled i rychlost zobrazování. Práce se tedy také zabývá jak výběrem metod, tak i optimalizacemi vedoucími ke zrychlení aplikace.

Kapitola 2

Teorie

Tato kapitola obsahuje souhrn a hrubý popis některých metod, které se nějakým způsobem využily ve finální aplikaci. Kromě těchto metod také obsahuje některé pojmy, které s problematikou úzce souvisí. Metodami, které byly vybrány a použity ve výsledné aplikaci, se bude podrobněji zabývat kapitola implementace 4.

2.1 Level of detail

Level of detail [8] (česky úroveň detailu, dále v textu pouze LOD) je popis jednotlivých úrovní kvality zobrazovaných modelů za účelem zvýšení výpočetního výkonu. Model je popsán více způsoby a každý způsob má jinou výpočetní náročnost. Nízká výpočetní náročnost úrovní je kompenzována zhoršením vizuálního vzhledu modelu. Za určitých okolností však toto zhoršení vizuálního vzhledu nemusí být zřejmé. Pokud se vybere správná metrika, podle které se bude vybírat úroveň detailu, tak může dojít k drastickému snížení výpočetní náročnosti při zachování stále dobrých vizuálních výsledků. LOD se rozlišuje na statický a dynamický[1].

U statického do procesu zobrazování vstupují již předem vytvořené úrovně detailu. To znamená, že například grafik vytvoří několik modelů stejného objektu a každý z nich je popsán jinou úrovní detailu. Model s nejvyšší úrovní detailu může být popsán desítkami tísíc polygonů, zatímco model s nejnižší úrovní detailu může být popsán pouze stovkami polygonů. Tento způsob tvorby LOD vede k uspokojivým výsledkům jak ze strany rychlosti výpočtu, tak ze strany vizuálního vzhledu aplikace. Nevýhodou je, že pořízení jednotlivých statických úrovní detailu je časově náročné a mnohdy je musí grafik vytvořit ručně.

Alternativou ke statickému LOD je dynamický, kdy jednotlivé úrovně LOD generuje až samotná aplikace. Jako vstup slouží pouze jeden model a za pomoci různých algoritmů či metod aplikace sama vytvoří nižší úrovně LOD. Některé z těchto algoritmů budou nyní probrány a posléze implementovány.

[1] https://cs.wikipedia.org/wiki/Level_of_detail

2.2 Metody zrychlení výpočtu zobrazovaných modelů

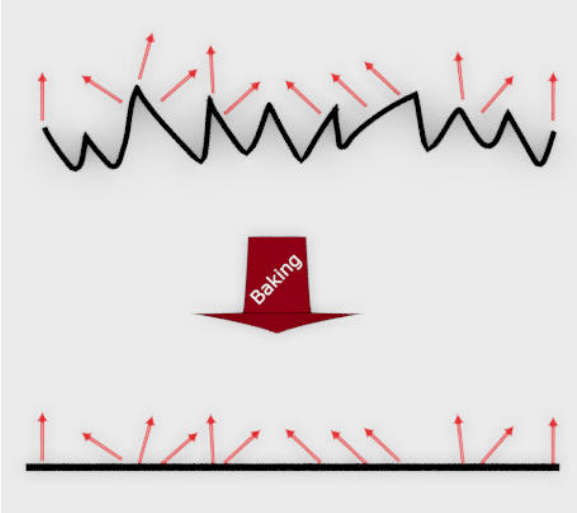
Znázornění rostlin v počítačové grafice zasahuje hned do několika oborů. Kromě programování zde svou roli hraje i grafika. Grafik, který rostlinu modeluje za pomoci k tomu určeného softwaru, se neobejde bez alespoň částečných znalostí fungování renderovací pipeline[2] a metod uzpůsobených k urychlení výpočtu. Několik z těchto základních metod zde bude rozebráno. Jedna ze základních metod používaných při urychlení výpočtu u polygonálního modelu se nazývá Normal Mapping.

2.2.1 Normal Mapping

Normal mapping (nebo také bump mapping)[7] je metoda, při které dochází k redukci počtu polygonů tvořících objekt tak, aby byl co nejvíce zachován původní vzhled objektu. V počítačové grafice jde o jednu z nejvíce rozšířených metod k urychlení výpočtu. Jedná se o osvětlovací metodu, kdy se při výpočtu barvy fragmentu během osvětlování polygonu nepoužívají normálové vektory jednotlivých vertexů a jejich interpolace, ale každému fragmentu se přiřadí normálový vektor z tzv. normálové mapy. Tato normálová mapa je bitmapa, kde každý pixel znázorňuje normálový vektor. Myšlenka normal mappingu byla poprvé zmíněna v práci Jamese F. Blinna, a to již v roce 1978 [3].

Je mnoho způsobů jak získat normálovou mapu. Jeden ze způsobů je tzv. zapékání[3] (anglicky baking), kdy se vezme polygonální model objektu ve vysokém rozlišení a jeho jednotlivé normálové vektory se uloží do bitmapy. Tyto normálové vektory se z bitmapy uloží do textury a ta se pomocí UV koordinátů[4] promítne na jednotlivé polygony. Při osvětlování tak může vzniknout iluze téměř jakéhokoliv povrchu. Výhodou normal mappingu je znatelná redukce polygonů modelu při zachování jeho vysoké věrohodnosti. Její výhody znatelně předčí nevýhody. Hlavní nevýhoda normal mappingu je samotný fakt, že povrch objektu je nahrazen bitmapou, tedy omezeným rozlišením, které je dáno rozlišením bitmapy. Jakmile se pozorovatel přiblíží k objektu, začne docházet k rasterizaci. Navíc je při naklonění a přiblížení objektu zcela patrné, že se jedná pouze o iluzi objemu povrchu.

[2] https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview
[3] http://wiki.polycount.com/wiki/Texture_Baking
[4] https://en.wikipedia.org/wiki/UV_mapping



Obrázek 2.1: Nahoře znázornění polygonálního modelu ve vysokém rozlišení a jeho normálových vektorů, dole znázornění zjednodušeného modelu s jeho původními normálovými vektory.

2.2.2 Billboard Clouds

Další velmi často používanou metodou pro zjednodušování modelů je metoda Billboard clouds. Tato metoda je poměrně stará, avšak stále používaná pro zjednodušování modelů. První pokusy o tuto metodu byly popsány již v roce 1985, kdy autoři Willliam T. Reeves a Ricki Blau nahradili složitý a komplexní strom sadou disků [13].

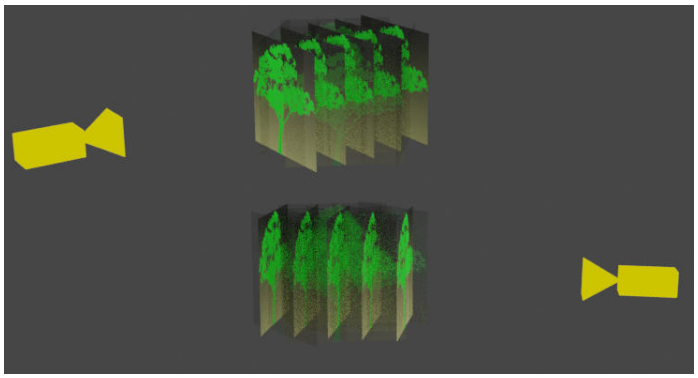
Princip metody spočívá v selekci množiny polygonů zjednodušovaného modelu a v nahrazení této zvolené množiny tzv. billboardem. Billbord (česky plátno) je většinou plocha tvořená jednotkami polygonů. Na tento billboard je poté nanesena textura ortografické projekce[5] všech polygonů z vybrané množiny. Tento postup se jednotlivými iteracemi opakuje. Převod rostliny na soustavu billbordů byl podrobně popsán v práci *Realistic real-time rendering of landscapes using billboard clouds* [2]. V práci je metoda výběru množiny polygonů (clustering)[6] určených ke zjednodušení popsána hned několika způsoby, díky čemuž je docíleno celkem věrohodného napodobení původního modelu rostliny.

Výhody této metody jsou stejně jako u normal mappingu v redukci polygonů daného modelu. Další výhodou je, že se dá kontrolovat LOD polygonálních modelů. Pokud je zapotřebí vyšší LOD, model se reprezentuje větším počtem polygonů. Pokud je zapotřebí nižší úroveň LOD, množství polygonů se sníží.

Nevýhoda této metody je, že úspěch, tedy i kvalita zobrazovaného modelu, záleží na clusterizační metodě. Jedna a ta samá metoda může mít různé výsledky pro různé modely objektů. Pro jeden model může zvolená metoda docílit vizuálně uspokojivých výsledků, nicméně pro jiný model může být zvolená metoda naprosto nedostačující.

[5] https://www.merriam-webster.com/dictionary/orthographic%20projection
[6] https://en.wikipedia.org/wiki/Cluster_analysis

Obrázek 2.2: Reprezentace rostliny pomocí metody billboard clouds. Obrázek je převzat z Realistic real-time rendering of landscapes using billboard clouds [2].

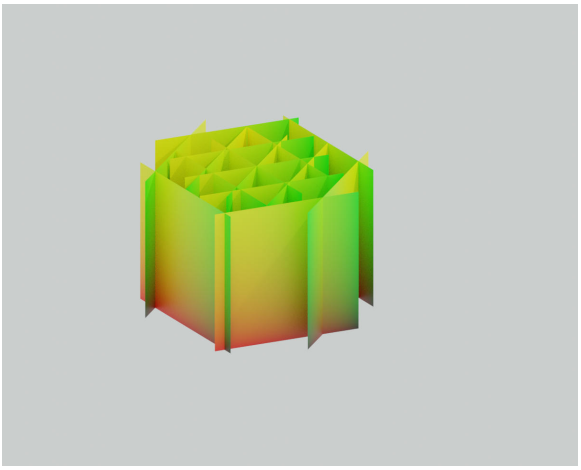


Obrázek 2.3: Znázornění jednotlivých plátů stromu. Kamera je znázorněna žlutou barvou. Nejvíce viditelné billboardy jsou právě ty nakloněné kolmo ke kameře.

2.2.3 Prolínání statických billboardů

Z názvu této metody již vyplývá, že stejně jako u billboard clouds i tato metoda pracuje s billboardy. Je mezi nimi však znatelný rozdíl. Zatímco billboard clouds umisťuje billboardy na pozice původních clusterů, metoda prolínání statických billboardů (jak bude dále v práci nazývána) umisťuje billboardy na předem stanovená místa. Tato metoda byla podrobně prozkoumána v práci Alekseje Jakulina z roku 2000 [6].

Princip spočívá v umístění celého objektu do bounding boxu (česky obklopující box), kdy je celý box rozřezán v pravidelných intervalech a v místech řezu objektu je ortograficky promítnut daný objekt do textury. Tato textura je poté promítnuta na příslušný plát, který vznikne při pomyslném rozřezání boxu. Samotným promítnutím textury na billboard by nebylo docíleno žádného vizuálně uspokojivého výsledku. Cíleného výsledku a iluze objemu je docíleno až tzv. prolínáním, kdy jsou zobrazeny pouze ty billboardy, které jsou nakloněny směrem k pozorovateli. Billboardy, které nejsou přímo nakloněny směrem k pozorovateli, postupně mizí. Princip fungování metody je znázorněn na obrázku 2.3.



Obrázek 2.4: Znázornění jednotlivých billboardů bez textury a bez prolínání.

Výhody metody spočívají, stejně jako u předešlých metod, v redukci polygonů. V tomto případě však je redukce polygonů nejdrastičtější. Ve své podstatě metoda nahrazuje celý objekt za sekvenci obrázků. Další výhodou metody je možnost kontrolovat LOD díky redukci počtu billboardů znázorňujících daný objekt. Hlavní nevýhodou je fakt, že metodu lze použít jen pro znázornění rostlin v dostatečné vzdálenosti od pozorovatele. Jakmile by se pozorovatel k objektu přiblížil, bylo by zcela očividné, že se nejedná o rostlinu, ale jen o sekvenci několika obrázků, což je zcela nežádoucí.

2.2.4 Převod na objem

Poslední metodou pro převod modelu, na kterou bude text zaměřen, je metoda převodu původně polygonálního modelu objektu do tzv. objemové mapy (je možné se setkat s pojmem voxel mapy), nebo jinak nazývané 3D textury. Tato 3D textura je poté různými metodami znázorněna ve dvourozměrném prostoru. Myšlenkou zobrazení objemu se zabývali v roce 1988 Robert A. Drebin, Loren Carpenter a Pat Hanrahan v práci *Volume Rendering* [5].

Na rozdíl od polygonálního modelu, kdy je model tvořen pouze povrchem daného objektu a tudíž neexistuje žádná informace o složení či vnitřní struktuře daného modelu, je u voxel mapy zaznamenán jeho objem. Objem je reprezentován voxely[7], což je ve své podstatě analogie pixelu v trojrozměrném prostoru. Voxel může uchovávat téměř jakékoliv informace, ať už se jedná o barvu, nebo například radiointenzitu.

Se znázorněním objemu v počítačové grafice je dnes možné se setkat zcela běžně. Velmi častým oborem, kde se tato metoda používá, je zdravotnictví. Tam najde využití například u CT skenu[8], kde se nejprve příslušná část například lidského těla naskenuje CT skenerem a právě výstup tohoto skeneru je voxel mapa. Ta se pak dále zobrazuje metodami, které budou nyní jen ve stručnosti probrány.

2.2.5 Metody zobrazení voxel mapy

Pokud je k dispozici voxel mapa, kterou je třeba zobrazit na dvourozměrný displej, je třeba ji převést na něco, co bude schopna zobrazit grafická karta. Jelikož renderovací pipeline grafických karet pracuje s polygonálním modelem, je nutné tuto voxel mapu převést taktéž na polygonální model, případně na pole pixelů. Tento proces představuje opravdu rozsáhlý obor v oblasti programování grafiky a sám o sobě citelně přesahuje rozsah bakalářské práce. Právě z tohoto důvodu se touto problematikou nebude následující text příliš zabývat. Jen ve stručnosti bude uvedeno pár základních metod pro zobrazení objemu.

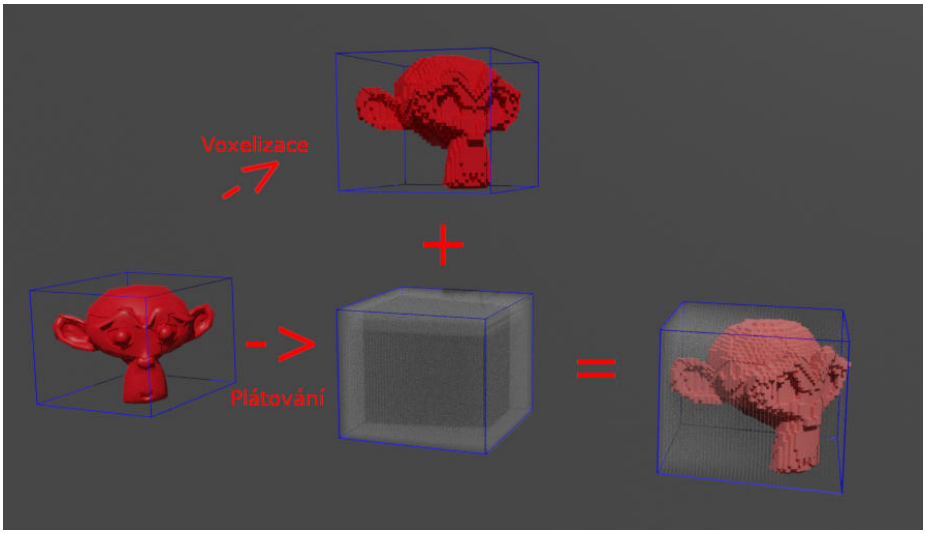
2.2.6 Volume ray casting

Volume ray casting [9], nebo také volume ray tracing, je v podstatě ray tracing, při kterém je vyslán paprsek procházející voxel mapou a končící v pixelu plátna, kde je voxel mapa zobrazována. Výsledná barva pixelu na plátně je dána heuristickou funkcí. Ta bere v potaz všechny voxely, kterými paprsek prošel. Přiřazením složek RGB jednotlivým komponentám voxelu na základě jejich intenzit tak vzniká výsledná barva daného pixelu. Díky raytracingu vzniká velmi hezké znázornění voxel mapy. Nicméně tato metoda je velmi výpočetně náročná a její použití v realtimové grafice je zcela minimální.

2.2.7 Zobrazení pomocí kolmých plátů

Tato metoda používá zcela jiný způsob zobrazení. Je založená na plátování [10], kdy je 3D textura proložena množinou plátů, které jsou permanentně nakloněny směrem k pozorovateli. Jsou uspořádány těsně u sebe a jejich množství je rovno rozlišení 3D textury. Tyto pláty jsou dále otexturovány právě 3D texturou s tím, že barvu jednotlivých pixelů plátu reprezentuje hodnota 3D textury na souřadnicích UVW. Jedná se tedy o stejnou metodu mapování, jako u klasických UV koordinátů textury, s tím rozdílem, že k souřadnicím UV

[7] https://en.wikipedia.org/wiki/Voxel?oldid=268651304
[8] https://cs.wikipedia.org/wiki/V%C3%BDpo%C4%8Detn%C3%AD_tomografie



Obrázek 2.5: Převod polygonálního modelu na objem. Nejprve je nutné vytvořit 3D texturu modelu a poté vytvoři pláty, na které tato textura bude nanesena.

se přidá i souřadnice W pro rozměr v ose Z. Tato metoda bude dále implementována v kapitole 4.4.1.

Výhody metody jsou především v její rychlosti oproti ray castingu a v možné implementaci na hardwaru grafické karty. Hlavní nevýhodou metody je, že vizualizace 3D textury v některých situacích nemusí být zcela přesná.

Zobrazení modelů rostlin za pomocí 3D textury je pravděpodobně ze všech zmíněných metod nejnovější. Je to právě i z toho důvodu, že 3D textura sama o sobě zabírá velké množství paměti a pro praktické využití v realtimových aplikacích se začala používat teprve nedávno, díky průběžně se zvyšujícímu výpočetnímu výkonu hardwaru. Konkrétně zobrazení rostlinstva pomocí 3D textury popsali ve své práci Philippe Decaudin a Fabrice Neyret v roce 2009 [4].

2.3 Animační a osvětlovací metody

Metody pro animaci rostlin se od běžných animací modelů příliš neliší. Jednou z nejrozšířenějších metod je keyframes [14]. Keyframes je metoda, při které se v určitém časovém okamžiku uloží informace o pozici vertexu. Výsledkem je posloupnost dvojicí pozic vertexů a času, ve kterém jednotlivé vertexy dané pozice nabývají. Pro plynulou animaci pohybů se pak interpoluje daná pozice vertexu pro aktuální časový okamžik. Výhodou této metody je její jednoduchost na implementaci a její přímočarost. Mezi nevýhody patří znatelná paměťová náročnost. Další nevýhodou je, že animace je fixní a nelze ji jednoduše dynamicky měnit. Pro dynamickou manipulaci pohybu objektu se více hodí kosterní animace.

2.3.1 Kostení animace

Kosterní animace [11] je způsob animování objektu, při kterém se polygonální model namapuje na tzv. kostru, což je jakási analogie koster v živém organismu. Kostra je stromová struktura kostí, které mají (většinou) fixní neměnnou délku. Geometricky se tyto kosti dají představit jako úsečka. Kostra má v počítačové grafice stejnou úlohu, jako v reálném světě.

Místo tkání však na sebe váže polygony modelu. Každý polygon je přiřazen jedné či více kostem a jeho pozice je vázána na pozice jeho rodičovských kostí. Pokud se začne pohybovat kostra objektu, dochází k pohybu samotných na ni vázaných vertexů. Prvotní myšlenka kosterních animací byla zmíněna v roce 1988[9]. Tato metoda má své výhody především v možnosti realtimové dynamické animace. To znamená, že díky kostem je možné vytvořit dynamicky libovolný pohyb modelu bez nutnosti uchovávat pozice o jednotlivých vertexech pro daný pohyb. Nevýhoda je vyšší náročnost na výpočet.

2.3.2 Skybox

Skybox je metoda pro zobrazení oblohy. Funguje na principu tzv. cube mappingu[10], kdy je obloha popsána množinou šesti obrázku, každý představující jednu stranu krychle. Kromě šesti textur se cube mapping vyznačuje od běžných textur i tím, že pro výběr jednolivých pixelů z textury nepoužívá UV koordináty, ale směrový vektor. To znamená, že se tento směrový vektor umístí do středu krychle a výsledná barva získaná z textury odpovídá té barvě, na kterou ukazuje daný směrový vektor. Pro zobrazení celé oblohy tedy stačí jedna polygonální krychle.

[9] https://en.wikipedia.org/wiki/Skeletal$_a nimation$
[10] https://en.wikipedia.org/wiki/Cube_mapping

Kapitola 3

Návrh

Cílém práce je implementovat aplikaci schopnou zobrazit tisíce až desetitisíce stromů, keřů a trav v reálném čase. Toto je sám o sobě velmi hrubý popis aplikace. Následující kapitola se bude zabývat popisem podrobnější.

3.1 Konkretizace aplikace

Zobrazit statisíce stromů jako pozadí, kde uživatel nemá možnost se k vegetaci přiblížit, nepředstavuje širší problém. Stačí, když se velké množství stromů promítne do jedné textury a poté je tato textura několika stromů promítnuta na billboard či jinak reprezentována. Tento postup se hodí zejména při generování pozadí, kde nehrozí, že se pozorovatel přiblíží k dostatečné vzdálenosti od zobrazovaného objektu.

Cílem implementované aplikace je, aby se pozorovatel mohl kdykoliv k libovolnému kusu vegetace svévolně přiblížit a aby při přiblížení byl zachován původní detail modelu. Tedy aby aplikace obsahovala informaci o jednotlivých kusech vegetace a aby nabízela možnost se k těmto jednotlivým kusům vegetace přiblížit.

Pokud bude existovat možnost se k zobrazovaným objektům vegetace libovolně přibližovat a zobrazit jejich detail, bylo by ideální, kdyby metoda určená k změně detailu nevyžadovala žádné specifické nároky na samotný model rostliny. To znamená, že vstup modelu rostlin by měl být klasický polygonální model v běžně používaných formátech, bez jakýchkoliv jiných nároků na implementaci samotného modelu. Aplikace by měla být schopna sama generovat jednotlivé LOD modelu a měla by je umět i sama použít.

Samotné modely ve scéně, kde jsou pouze modely rostlin, by nepůsobily příliš realistickým a hlavně účelným dojmem. Právě z tohoto důvodu je žádoucí, aby aplikace zobrazovala i terén, na kterém bude množství vegetace zobrazeno. Nároky na tento terén jsou, stejně jako u vegetace, aby byl model terénu vyjádřen klasickým formátem pro ukládání 3D modelu. Z toho vyplývá, že térén nebude generovaný, ale bude předem vymodelovaný. Je žádoucí, aby bylo možné určit pozice v terénu, na kterých budou zobrazovány jednotlivé rostliny. S tím souvísí generovaní vegetace.

Jak již bylo zmíněno, od aplikace se očekává, že bude schopna jednotlivé modely rostlin umisťovat na příslušný model terénu. Ovšem generování těchto rostlin náhodně nepřichází v úvahu. Pokud by byly rostliny umisťovány zcela náhodně, krajina by nepůsobila příliš reálným dojmem. V reálném světě ohromné stromy také většinou nerostou na vodě ani nemůžou zakořenit ve skalách. Většinou na skalách ovšem mohou růst malé rostliny nebo traviska. Je tedy nutné nějakým způsobem určit, kde a jaké rostliny mohou na terénu růst.

Také by bylo vhodné mít možnost určit pozice rostlin ručně, tedy aby existovala možnost umístit libovolnou rostlinu na příslušné předem vybrané místo v terénu.

Aby bylo možné určit, zda daná rostlina na příslušném místě roste, je nutné rozlišit i typy těchto rostlin. Model stromu určitě nebude moci růst na stejném místě, jako například model vysokohorského křoví. Dále se musí brát v potaz i fakt, že množství stromů bude pravděpodobně nižší, než například množství trav. Proto je nutné určit množství zvlášť pro každou rostlinu.

Jako v každé aplikaci pracující s grafikou, je vždy účelné aby zobrazovaná grafika vypadala vizuálně co možná nejlépe. Z tohoto důvodu je žádoucí, aby byl implementován alespoň základní osvětlovací model pro jednotlivé rostliny. Z toho vyplývá možnost manipulace se samotným světlem a s ruční změnou osvětlení dané scény. Aplikace by také měla nabídnout možnost se světlem a dalšími parametry aplikace manipulovat.

Nyní byla aplikace a její funkcionalita konkretizována. V další části textu bude konkrétněji probráno, jakými způsoby a metodami bude v této práci požadovaných kritérií docíleno.

3.2 Tvorba terénu

V první řadě je nutné zajistit terén, na kterém se příslušné rostliny budou moci generovat. Jelikož z popisu aplikace vyplývá, že se jedná pouze o polygonální model, což je pouze popis vzhledu daného objektu, je nutné nějakým způsobem doplnit informaci o jeho výšce v daném bodě. Způsobů, jak tuto informaci získat, je více. Jedním z řešení může být výpočet kolizí s povrchem terénu. Pro účely této aplikace však byla zvolena metoda výškové mapy[1]. Díky ní může mít grafik plnou kontrolu při tvorbě terénu nad jeho výškou. Tedy i tou výškou, na které se mohou generovat rostliny.

3.3 Generování vegetace

Jakmile bude k dispozici terén, přichází na řadu generování samotných pozic rostlin. Terén je povrch, který je na osách X a Z definován vždy. Jediný rozměr, kde není definován, je právě osa Y. Jelikož na osách X a Z je definován povrch vždy v intervalech rozměru dáného terénu, můžeme generovat X a Z souřadnice náhodně. K tomu poslouží generátor pseudonáhodných čísel v rozsahu rozměrů geometrie daného modelu. Jakmile je vygenerovaný pár pozic X a Z, zjistí se příslušná výška daného terénu z height mapy.

Jakmile je vygenerovaná pozice XYZ nové rostliny, je nutné ověřit, jestli na této pozici terénu daná rostlina může existovat. Tato informace je uchována pro každý terén v takzvané veggie mapě.

Pod pojmem veggie mapa bude dále označována bitmapa, která se stejně jako výšková mapa přiřadí k terénu. Na rozdíl od height mapy nebude obsahovat informaci o výšce terénu, ale o zastoupení vegetace na daném terénu. Veggie mapa je dále konkrétněji popsána v kapitole implementace 4.3.4.

3.4 Zobrazení vegetace

Jakmile budou k dispozici pole pozic jednotlivých rostlin, přichází na řadu jejich zobrazení. Není možné zvolit pouze jeden způsob zobrazení rostliny. Pokud se rostlina nachází blízko

[1] https://en.wikipedia.org/wiki/Heightmap

pozorovateli, je logické, že bude žádoucí zobrazit model ve vysokém rozlišení i s jeho detaily. Tento způsob je pochopitelně výpočetně většinou nejnáročnější. Zatímco když je rostlina od pozorovatele daleko, zobrazení původního detailního modelu je neefektivní z zlediska výpočetního výkonu. Z tohoto důvodu je potřeba rozlišit úrovně zobrazovaného modelu a tedy rozdělení LOD.

Použitá metrika pro určování LOD je vzdálenost od kamery. Prvotní návrh výběru zobrazovací metody je takový, že ty objekty, které jsou ke kameře nejblíže, se zobrazí původním polygonálním modelem. Ty objekty, které jsou ve střední vzdálenosti od kamery, se zobrazí metodou převodu na objem. Nakonec ty objekty, které jsou nejdále od kamery, se zobrazí pomocí metody prolínání statických billboardů.

Konkrétní užití metod se může však ve výsledné implementaci lišit. Cílem je tyto metody prozkoumat a zjistit, pomocí kterých metod a s jakými parametry lze docílit nejlepších výsledků.

Kapitola 4

Implementace

V této kapitole budou podrobněji probrány implementační detaily práce. Bude zaměřena konkrétněji na:

- Použité API, knihovny a formáty zpracovávaných dat
- Tvorbu a zapékání zjednodušených modelů
- Tvorbu scény, její rozdělení na sektory a generování vegetace
- Zobrazení modelů a jejich jednotlivých úrovní detailu
- Metody na urychlení vykreslování modelů

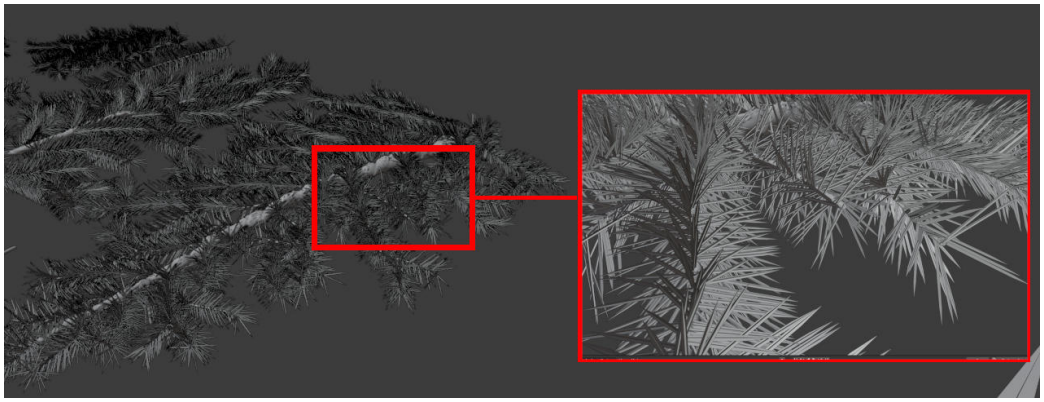
4.1 Použité API a knihovny

Výsledná aplikace (dále také popisována jako Veggie) je implementována pomocí grafického API OpenGL [12]. Je napsána v programovacím jazyce C++ verze 14 a byla vyvíjena pod operačním systémem Linux (konkrétněji Ubuntu). Nicméně výsledná aplikace je také spustitelná pod operačním systémem Windows. Kromě samotného API OpenGL bylo pro implementaci některých části aplikace užito knihovny GPUEngine[1]. Konkrétně je knihovna využita pro zapouzdření scény (GeSg) a import obrázků i objektů za pomocí jejího rozšíření GeAd. Pro správu okna v systému je využit framework GLFW[2]. Pro import objektů používá GeAd externí knihovny Assimp a pro import obrázků knihovnu FreeImage.

Assimp[3] je open source knihovna pro načítání polygonálních modelů. Podporovaný soubor modelu objektu reprezentuje do jednotné stromové struktury, kterou poté převádí knihovna GeAd do vnitřní reprezentace scény GPUEnginu za pomocí GeSg. Z toho vyplývá, že Veggie podporuje všechny vstupní formáty objektů, které dokáže přečíst Assimp. Mezi tyto soubory patří například .obj, .dae, .mdl, .md5 a jiné.

FreeImage[4] je open source knihovna pro načítání obrázku. Všechny načtené a podporované typy obrázků jsou převedeny do jednotné podoby, bez ohledu na jejich formát a typ uložení dat. Knihovna a tedy i Veggie podporují obrázky formátu např. png., jpg., bmp. a jiné.

[1] https://github.com/Rendering-FIT/GPUEngine
[2] https://www.glfw.org/
[3] https://www.assimp.org/
[4] https://freeimage.sourceforge.io/



Obrázek 4.1: Na obrázku je vidět detail modelu větve. Je zde vidět každá jehlička, která je reprezentována několika polygony. Z obrázku je patrné, že tímto způsobem nelze vymodelovat celý strom. I samotná větev není pro dnešní grafické karty z hlediska výkonu zanedbatelná. Právě proto přichází na řadu zapečení tohoto modelu do textury.

Pro GUI a ovládání různých parametrů aplikace (jako třeba pozice světla) je využito knihovny AntTweakBar[5]. Tato knihovna je stejně jako předešlé knihovny OpenSource. Pro úplnost je třeba zmínit i OpenGL wrapper GLEW[6].

4.2 Tvorba modelů

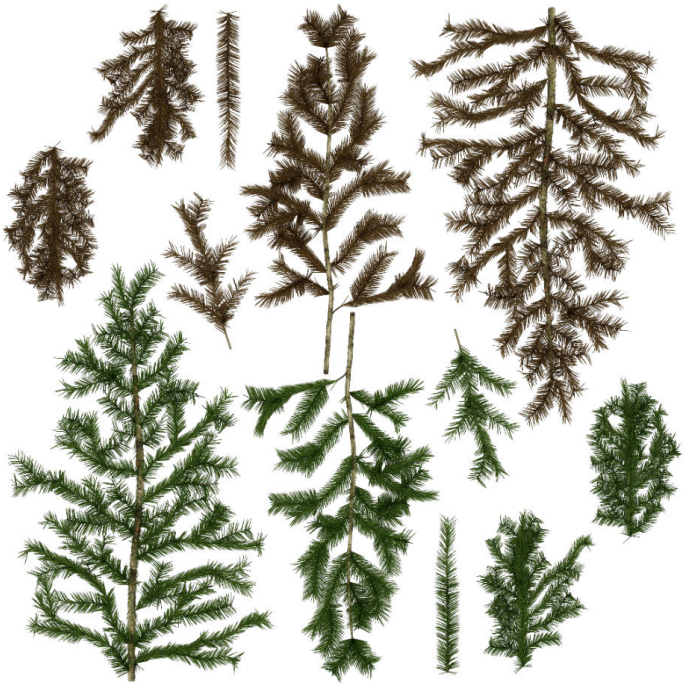
Pro účely této práce bylo vytvořeno několik polygonálních modelů stromů, křoví, trav i terénu za pomocí 3d grafického softwaru Blender[7]. Proces se skládal z několika částí. V první řadě bylo zapotřebí vytvořit polygonální model v nejvyšším možném rozlišení. Pokud by se vzal jako příklad model jehličnatého stromu, nejprve bylo nutné vymodelovat jednu větev se vším jejím jehličím, tu takzvaně zapéct a z této zapečené větve poté vymodelovat následující větší větve. Tento iterativní proces se opakoval až k finálnímu celému stromu.

4.2.1 Zapečení modelu do textury

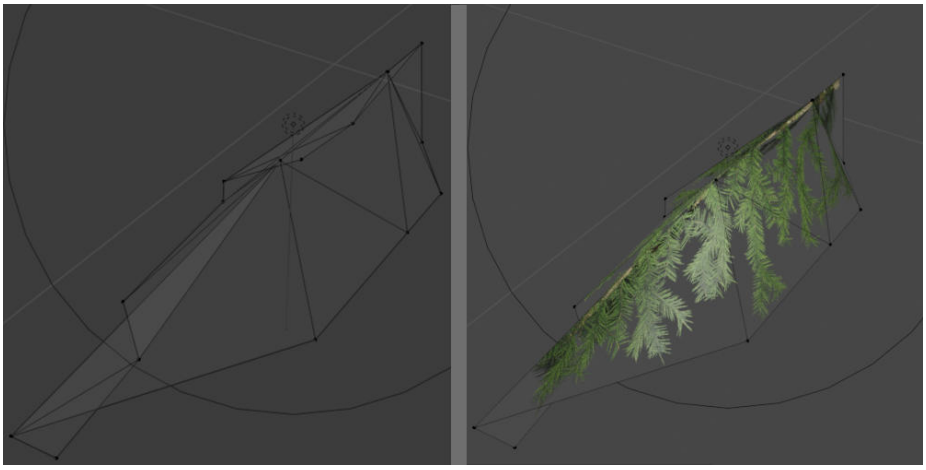
Zapečení modelu je provedeno jednoduchým způsobem, kdy je celý model vykreslen za pomoci ortografické projekce do textury. Jakmile je model zapečený v textuře, je možné tento model aproximovat malým množstvím polygonů. Na obrázku 4.2 lze vidět výsledek zapečeného polygonálního modelu.

Jakmile je k dispozici textura daného objektu, je možné ji umístit na primitivnější model. Tento postup je znázorněn na obrázku 4.3. Nejdříve je vytvořen model, poté je UV namapován a následně je na něj umístěna textura z původního modelu. Původní větev byla tvořena statisíci polygony, zatímco zjednodušenou větev tvoří 12 polygonů. Nicméně u použití této metody vzniká problém v tom, že se ztratila informace o umístění jednotlivých jehliček, tudíž je není možné korektně osvětlit. Z tohoto důvodu je nutné vygenerovat z původního modelu i normálovou mapu, která obsahuje normálový vektor naklonění pro každý pixel textury.

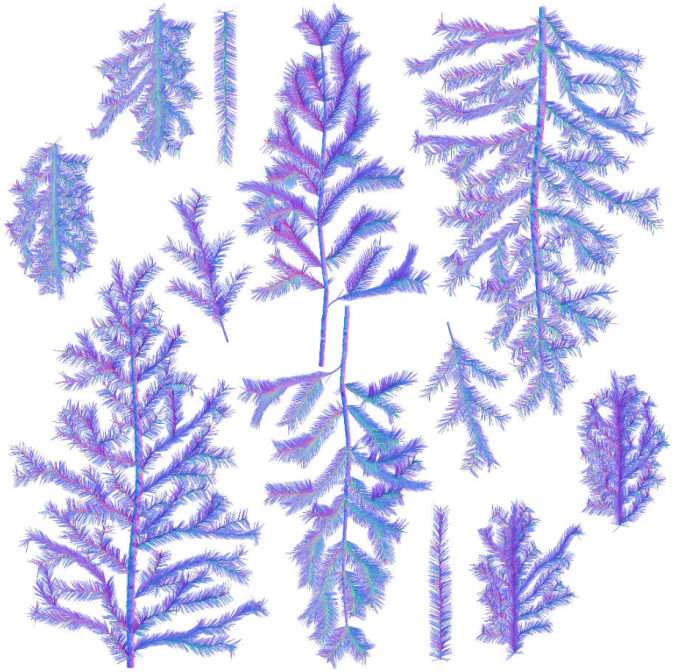
[5] http://anttweakbar.sourceforge.net/
[6] http://glew.sourceforge.net/
[7] https://www.blender.org/



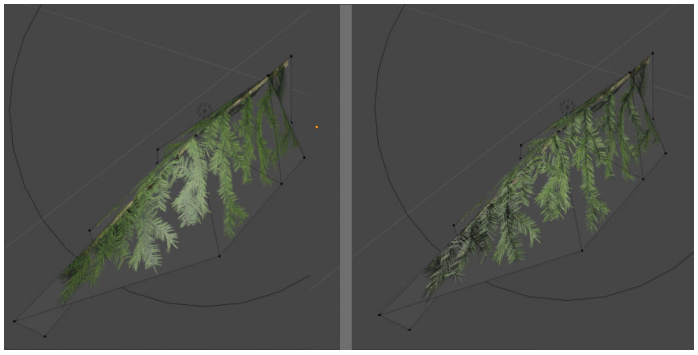
Obrázek 4.2: Výsledek zapečení modelu z obrázku 4.1. Obrázek je ortografickou projekcí modelu směrem ze shora.



Obrázek 4.3: Umístění textury na zjednodušený model větve. Nalevo je zobrazen polygonální model větve, napravo ten samý model po aplikaci textury.



Obrázek 4.4: Vygenerovaná normálová mapa vysoce polygonální větve.

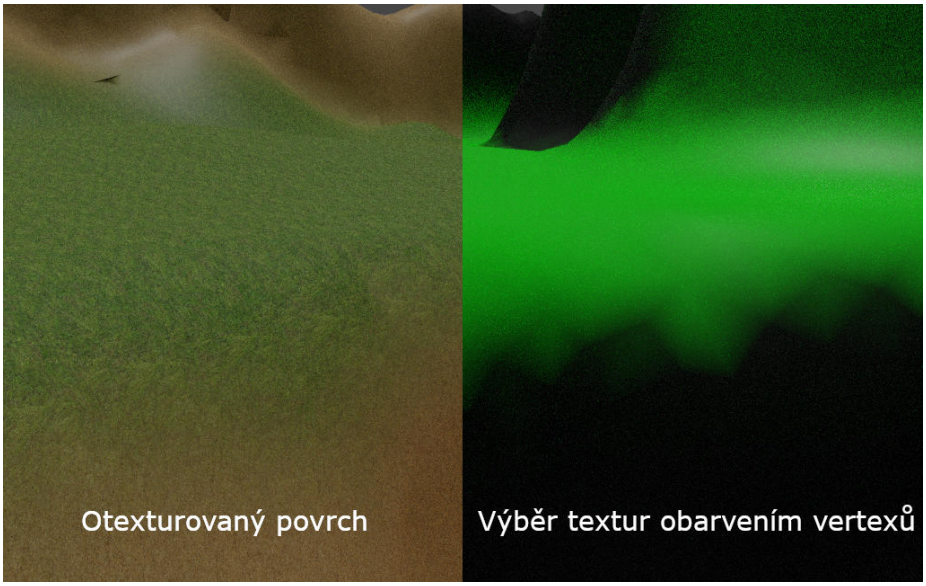


Obrázek 4.5: Porovnání modelu bez normálové mapy a s normálovou mapou.

Aplikace blender nemá pravděpodobně žádnou možnost jednoduše vygenerovat normálovou mapu přesně pro tento požadovaný případ. Nicméně má možnost tvořit své individuální shadery a aplikovat je na vykreslovaný objekt. Díky tomu lze vytvořit jednoduchý shader, který obarví požadovaný povrch objektu na základě jeho normálového vektoru. Stačí když se jednotlivým složkám barvy pixelu RGB přiřadí hodnoty složek normálového vektoru příslušného fragmentu XYZ a aplikuje se na ně vztah:

$$C = N * 0.5 + 0.5$$

Výsledkem je normálová mapa, kterou lze použít pro přesné osvětlení dané větve. Výsledná normálová mapa je znázorněna na obrázku číslo 4.4. Dále pak na obrázku číslo 4.5 je porovnání osvětlení objektu bez normálové mapy a osvětlení objektu s normálovou mapou. Postup vytváření objektu se dále opakuje. Výsledná větev tvoří větší větve a ty pak dále tvoří celý strom.



Obrázek 4.6: Vlevo je znázornění výsledného obarveného terénu. Vpravo je znázornění obarvených vertexů, díky kterým se vybírá výsledná aplikovaná textura na terén.

4.3 Tvorba scény

Scéna je tvořena terénem popsaným polygonálním modelem stejně jako vegetace. To znamená, že není nijak procedurálně vygenerován, ale je vytvořen ručně taktéž za pomoci aplikace blender. Nicméně je nutné tento model vytvořit s metodami, které jsou odlišné od původního postupu.

4.3.1 Terén

U běžných polygonálních modelů přichází model s UV koordináty pro každý vertex, které určují pozici vertexu na dané textuře. Díky tomu získá výsledný polygon barvu příslušné textury. Barva celého modelu je poté popsána pouze jednou texturou. Nicméně toto se nedá příliš použít při tvorbě terénu z toho důvodu, že terén je velký několik kilometrů. Pokud by bylo potřeba na terénu zobrazit například detail trávy (není myšleno nyní model trávy), musela by být textura a tedy její rozlišení přímo enormní. Řešením tohoto problému je tzv. roztáhnutí UV koordinátu, kdy se na objekt textura nanese několikrát a tudíž se zdánlivě zvýší rozlišení výsledné textury. Toto sice pomůže vyřešit jeden problém, ale druhý problém vzejde z faktu, že celý objekt je popsán pouze jednou opakující se texturou, což je pro terén nepřijatené. Tento problém se podaří odstranit v případě, že se terén obarví nejprve různými barvami. Tentokrát se však nepoužije textura pro obarvení, ale obarví se příslušné vertexy. Vertex může mít několik atributů, tudíž lze přidat informaci i o barvě. Každá barva vertexu poté odkazuje na příslušnou použitou texturu. UV koordináty tedy zůstávají roztáhlé na všechny textury stejně, mění se však pouze daná textura, která se na polygon umisťuje. Výsledkem této metody je, že grafik může terén obarvit libovolným způsobem a barva terénu si zachová vysoké rozlišění. Metoda je popsána podrobněji na obrázku 4.6.

Nyní je potřeba tento model načíst do paměti grafické karty. Model terénu může být tvořen libovolným počtem textur. Textury jsou v aplikaci veggie uloženy za pomocí pole

Obrázek 4.7: Height mapa terénu. Nejtmavěji vybarvená místa jsou ty nejhlouběji položená. Nejsvětleji vybarvená místa jsou naopak ty nejvýše položená.

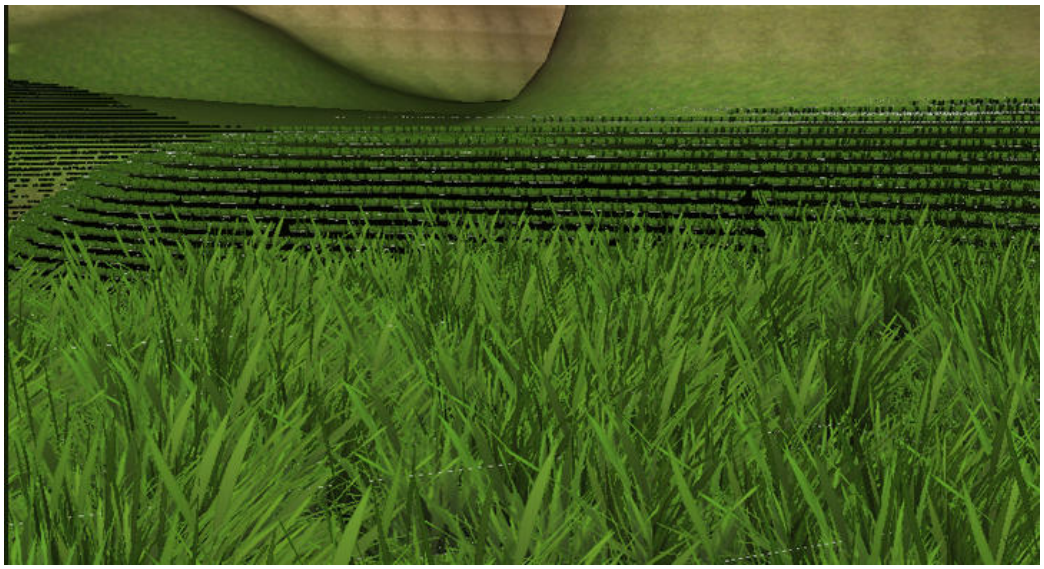
2D texur. K tomuto účelu slouží v OpenGL textura typu GL_TEXTURE_2D_ARRAY, kde se jedná v podstatě o 3D texturu, nicméně nedochází k interpolaci pixelů mezi jednotlivými texturami. Pole se indexuje stejně jako u běžné texutry, kde U a V značí souřadnici na konkrétní textuře a souřadnice W značí index textury v poli. Stejně tak slouží i GL_TEXTURE_2D_ARRAY pro textury jednotlivých normálových map terénu.

4.3.2 Height mapa

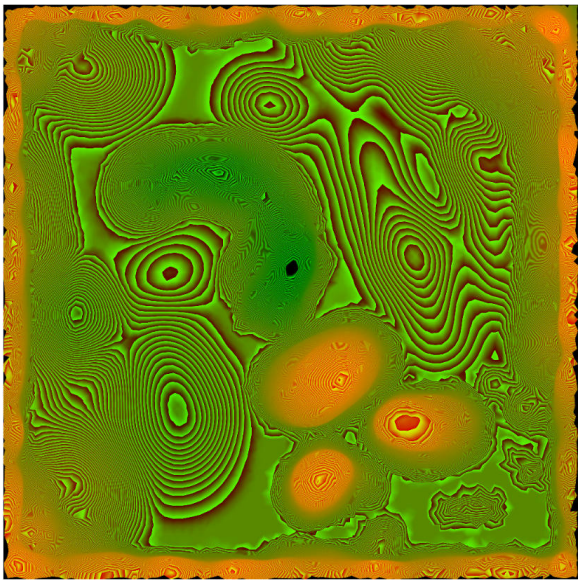
K zjištění výšky terénu v aplikaci je použita velmi častá metoda height mapy, kdy je povrch terénu reprezentován rastrovou sítí, kde hodnota pixelu představuje v této síti výšku v ose Y. Pokud je potřeba zjistit výšku terénu v bodě X = 500 a Y = 500, vypočte se pomocí jednoduchého vztahu $V_k = F(500, 500) * (V/B_m) + V_m$, kde V_k představuje výslednou výšku terénu, funkce $F(x, z)$ navrací hodnotu bitmapy v bodě $[X, Z]$, V je celková výška modelu terénu, B_m je maximální hodnota bitové hlouby bitmapy a kde V_m představuje nejníže položený bod terénu.

Z dané metody však vyplývá problém znázorněný na obrázku 4.8 . Z důvodu maximálně 256 hodnot u černého obrázku na pixel, vzniká pouze 256 různých úrovní výšky terénu. Z toho důvodu vznikne dojem, že rostliny jsou seřazeny ve schodovitých úrovních. Snaha k vyřešení tohoto problému vedla k vytvoření výškové mapy, která je reprezentována 16 bity pro každý pixel. Původních 256 úrovní terénu tedy vzroste na 65 536. Taková mapa je zobrazena na obrázku 4.10.

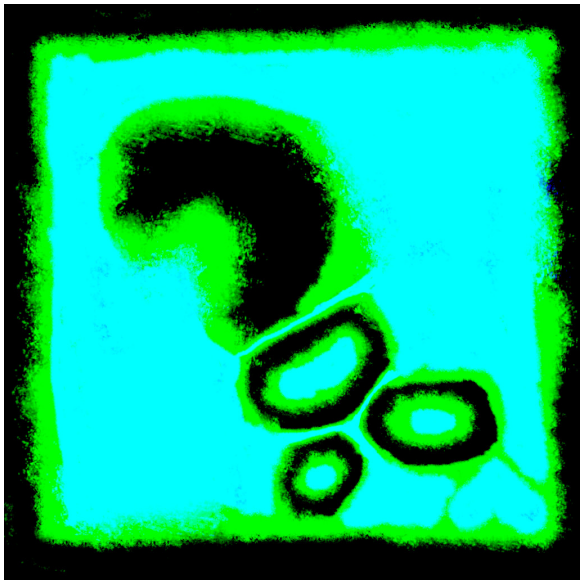
I když se tímto problém nízkého rozlišení bitmapy odstraní, vzniká další problém. A to je problém pořízení bitmapy. Aplikace blender při generování bitmapy u barevných obrázků automaticky interpoluje mezi jednotlivými složkami barev bitmapy. To způsobuje občasné nepřesnosti hlavně u přechodů mezi úrovněmi výšky u červené barvy. Tyto nepřesnosti se v implementaci podařilo odstranit zprůměrováním hodnot okolních pixelů. K určení hodnoty jednoho pixelu je tedy zapotřebí znát hodnoty všech jeho okolních sousedů a tyto hodnoty zprůměrovat. Teprve tímto způsobem bylo docíleno rovnoměrného, neschodovitého



Obrázek 4.8: Schodovitý efekt. Kvůli maximální hodnotě jednoho bytu která je 255, lze rozdělit celý terén jen do 256 úrovní výšky. Z toho důvodu dochází ke schodovitému efektu.



Obrázek 4.9: Výšková bitmapa s 16 bitovou hloubkou. Nejprve je terén rozdělen na 256 úrovní, zapsaných červenou barvou. Každá z těchto 256 úrovní je poté rozdělena na další 256 podúrovní. Díky tomu vzniká pruhovité vyplnění textury.



Obrázek 4.10: Znázornění veggie mapy. Jednotlivé složky RGB znázorňují výskyt jednotlivých typů rostlin.

zobrazení vegetace na povrchu. Tímto se dostáváme k tématu implementace generování rostlin.

4.3.3 Generování vegetace

Každá rostlina je ve veggie reprezentována třemi atributy. První je odkaz na objekt, kterým je rostlina reprezentována, druhým atributem je pole pozic, kde se rostlina vyskytuje a třetím atributem je její typ. Typ určuje, jestli se jedná o malou rostlinu (tráva, květina aj.), střední rostlinu (keře a malé stromky) a velkou rostlinu (stromy).

Veggie pro umisťování rostlin (dále jen sázení) po terénu nabízí možnost, kdy lze rostlinu zasadit na konkrétní pozici, případně je možné vygenerovat libovolný počet rostlin na náhodných pozicích za pomocí generátoru pseudonáhodných čísel. Tento pseudonáhodný generátor má nastavenou výchozí hodnotu seedu, díky čemuž dochází k pravidelným výsledkům rozmístění rostlin. Je však nutné nějakým způsobem zajistit informaci o tom, na kterých pozicích lze rostliny generovat a na kterých je nelze generovat. K tomuto účelu slouží veggie mapa.

4.3.4 Veggie Mapa

Veggie mapa je pojem kterým je označena metoda uchování informace o existenci různých typů rostlin na příslušném povrchu. Stejně jako height mapa se váže na příslušný povrch objektu. Jednolivými barvami je popsána možnost výskytu rostliny. Každá rostlina se řadí do jedné ze tří kategorií (málá, střední, velká). Každé kategorii odpovídá jedna komponenta z barev RGB. Touto veggie mapou se pokryje povrch objektu. Příslušná vygenerovaná 3D pozice rostliny se pak kontroluje, zdali se může na povrchu nacházet.Pokud je na veggie mapě na pozici vygenerované rostliny barva její kategorie, do které rostlina spadá, výsledná pozice se uloží do pole pozic. Pokud se tam barva příslušné rostlinné kategorie nenachází, pozice je zahozena. Algoritmus generování je popsán níže.

Algorithm 1: Algoritmus generování pozic

```
nastav RostlinaTyp na typ generované rostliny;
ulož informaci o délce terénu do délkaTerénuX a délkaTerénuZ;
nastav délkaTerénuX na délka terénu ulož informaci o počátku terénu do
 počátekTerénuX a počátekTerénuZ;
while PolePozic je menší než počet generovaných rostlin do
    nastav pseudoX na náhodné číslo od 0.0 do 1.0;
    nastav pseudoZ na náhodné číslo od 0.0 do 1.0;
    nastav poziceX na pseudoX * délkaTerénuX + počátekTerénuX;
    nastav poziceZ na pseudoZ * délkaTerénuZ + počátekTerénuZ;
    if na pozici [poziceX,poziceZ] ve veggie mapě existuje RostlinaTyp then
        nastav poziceY na výška heightMapy v bodě [poziceX,poziceZ];
        přidej do PolePozic bod [poziceX,poziceY,poziceZ];
    else
        zahoď vygenerované pozice a pokračuj;
    end
end
```

4.4 Zobrazení modelů

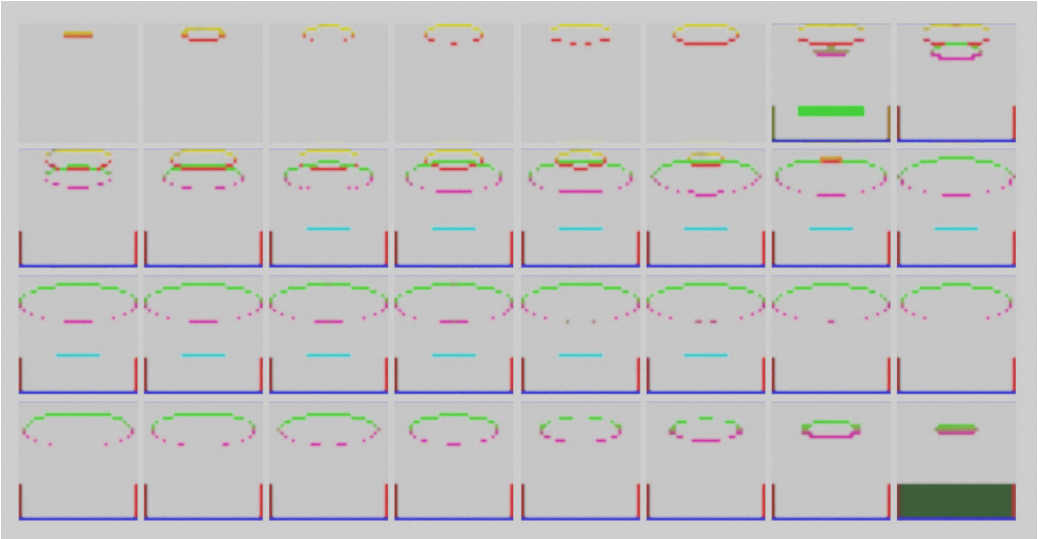
Jakmile máme k dispozici model i jeho pozici, je možné jej vykreslit. Aplikace veggie implementuje tři různé metody pro zobrazení modelu, jak bylo popsáno v návrhu. První z nich je metoda převodu modelu na objem.

4.4.1 Převod na objem

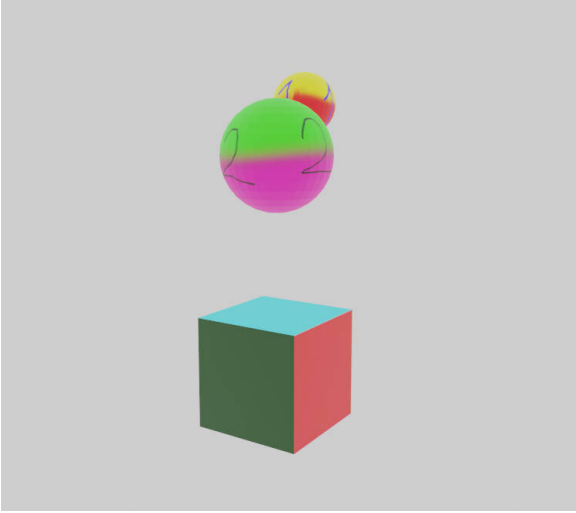
První fází této metody je převod modelu na 3D texturu. Jednotlivé voxely této 3D textury jsou tvořeny prvky stejnými, jako klasický pixel, tedy prvky RGBA. Texturu získáme skenováním objektu.

Skenováním je myšleno promítání tohoto objektu v pravidelných řezech ortografickou projekcí. V implementaci je objekt promítán do framebufferu, odkud je poté ukládán to textury. Ortografická projekce je docílena užitím ortografické projekční matice (glm::ortho), která má své parametry Znear a Zfar určeny aktuálním řezem objektu. Šířka i výška projekčního okna matice se rovná rozdílům maximální a minimální pozice všech vertexů objektu, které jsou viditelné z pohledu projekce. Objekt se ve své podstatě umístí do pomyslné nesymetrické krychle a každá strana krychle je poté naskenována. Tento proces se tedy opakuje celkově šest krát, pro každou stranu krychle. 3D textura je znázorněna na obrázku 4.11 a původní vzhled objektu je znázorněn na obrázku 4.12.

Počet řezů objektu závisí na rozlišení textury. Pokud má textura například rozlišení 32x32x32 pixelů, je třeba provést skenování 32x pro každou stěnu krychle. Právě toto je největší problém u 3D textur. S rozlišením jejich paměťová náročnost roste exponenciálně. Pokud budeme mít texturu o rozlišení 512x512x512 a pro ideální zobrazoní 3D objektu potřebuje 6 těchto textur pro každou stranu krychle, narazíme na paměťový problém. Pokud každý voxel zabírá 4 Byty paměti, výsledná textura zabere 3 221 225 472 Bytů (přesně 3GB) paměti. 3GB paměti představují průměrnou maximální kapacitu paměti dnešních grafických karet. Tedy tato metoda by byla pro dnešní použití takřka nepoužitelná. Pokud však zvolíme



Obrázek 4.11: 3D textura. Jednotlivé obrázky značí řezy texturou z pohledu osy Z. Rozlišení textury je 32x32x32 pixelů pro názornost. Pro úplnou 3D texturu je zapotřebí dalších 5 skenů objektu ze zbývajících stran krychle.



Obrázek 4.12: Původní objekt převedený do 3D textury na obrázku 4.11.

texturu o rozměrech 256x256x256 voxelů, její paměťová náročnost klesne na 384 MB, což sice není zanedbatelné, ale pro dnešní grafické karty je tento požadavek zvládnutelný.

Jakmile máme 3D texturu naskenovaného objektu připravenou, je nutné ji uložit do paměti grafické karty. Pro tyto účely slouží u API OpenGL formát textury GL_TEXURE_3D. Ta tuto texturu interpretuje stejně jako jinou texturu a po předání dat textury je v shaderech přístupná přes samplery, stejně jako například u 2D textury. Stejně tak se na 3D texturu aplikuje MipMapping[8]. K jednotlivým voxelům a k jejich interpolacím se přistupuje přes příslušnou *texture* funkci, která bere jako parametry mimo jiné i UVW koordináty.

Nyní je 3D textura naskenovaného objektu připravena a načtena v paměti grafické karty. Api OpenGL a jeho pipeline pracuje pouze s trojúhelníkovými polygony. Proto je nutné 3D texturu těmito polygony nějak reprezentovat.

4.4.2 Metoda kolmých plátů

Jak již bylo zmíněno v úvodní sekci práce, tato metoda rozseká texturu na N plátů, kde N je rozlišení textury. Tyto pláty obarví barvou textury a vznikne iluze objemu objektu. Tento proces bude probrán nyní krok po kroku.

Ze všeho nejdřív je nutné vytvořit bounding box kolem původního polygonálního modelu. Tento bounding box je v práci reprezentován úsečkou představující tělesovou diagonálu vzniklého bounding boxu. K popisu bounding boxu totiž stačí pouze dva body. Začátek a konec již zmíněné diagonály. Počátek diagonály se nachází v nejzápornějších hodnotách os souřadného systému a konec v nejkladnějších hodnotách os souřadného systému. Jakmile je k dispozici tato úsečka, může se dále jako atribut poslat vertex shaderu, který ji dále předává geometry shaderu. Právě v geometry shaderu vzniká příslušný kolmý plát, který je permanentně nakloněn směrem k pozorovateli.

Nejprve si geometry shader převezme úsečku od vertex shaderu. Z úsečky vypočte tzv. midpoint, což je bod přesně ve středu dané krychle. V tomto bodě zkonstruuje geometry shader rovnici roviny v protoru, která je kolmá k rovnici přímky, směřující od pozorovatele k midpointu. Tento proces je znázorněn na obrázku 4.13.

Jakmile je k dispozici rovnice roviny, sestrojí geometry shader rovnice přímek všech hran bounding boxu. Poté vypočte průsečíky mezi zmíněnou rovinou a jednotlivými přímkami ohraničující bounding box. Z těchto průsečíků poté vybere právě ty body, které leží pouze v boundingu boxu. Výsledkem je množina bodů, které tvoří požadovaný plát. Z logiky věci pak vyplývá, že plát může být trojúhelník, čtverec, kosočtverec atd., až po hexagon.

V tomto případě pro výpočet roviny, přímek a bodů průniku postačí jen 3 základní informace. A to jsou vektory $\vec{U_{min}}$, $\vec{U_{max}}$ značící pozici záčátku a konce úhlopříčky tvořící krychli, a vektor $\vec{C}$, který značí pozici kamery. Z těchto tří informací lze vypočítat vše potřebné.

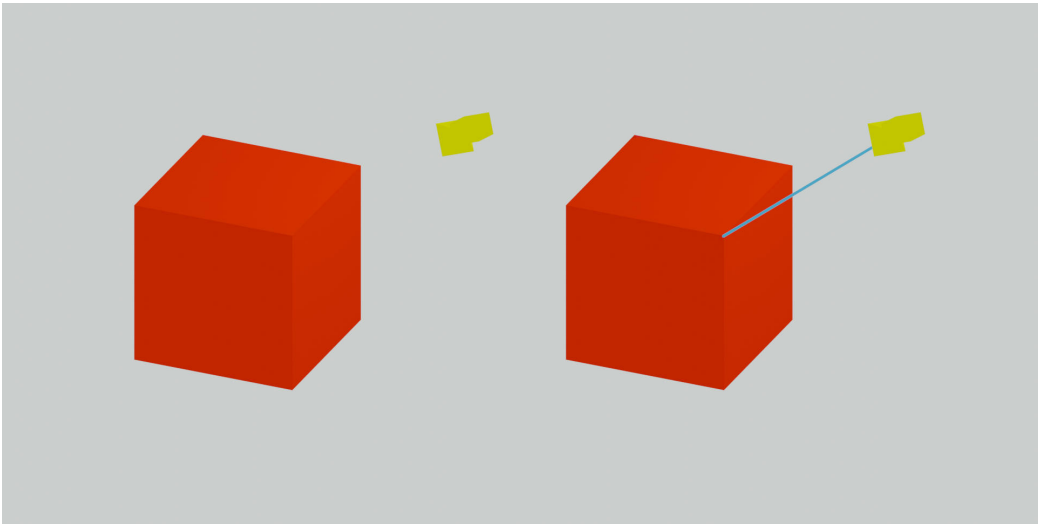
Ze všeho nejdřív je nutné vypočítat midpoint $\vec{M}$, což je vektor obsahující souřadnici středu krychle v prostoru. Midpoint lze vypočítat jednoduchým vztahem

$$\vec{M} = \vec{U_{min}} + (\vec{U_{max}} - \vec{U_{min}}) * 0.5$$

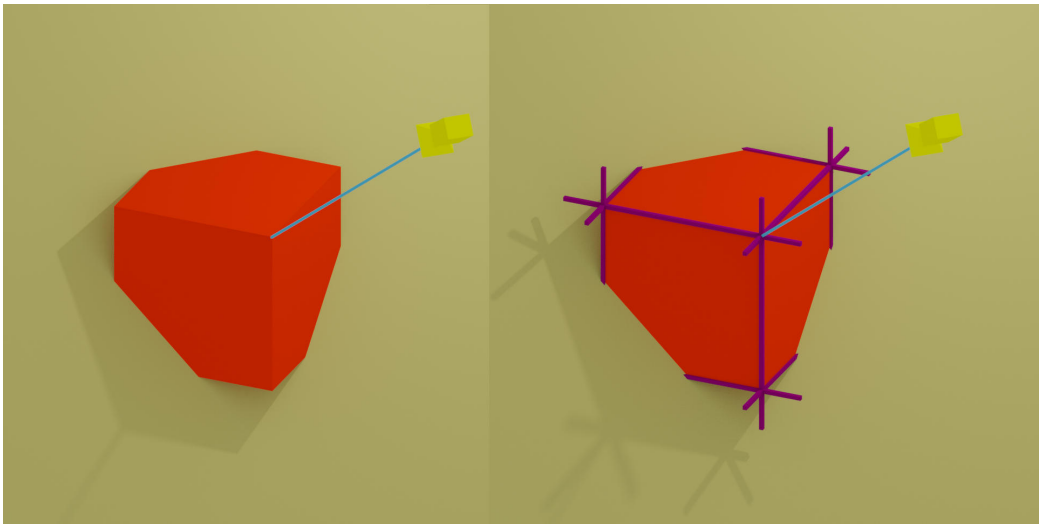
Jakmile je známá pozice středu krychle, lze vypočítat vektor mířící směrem od midpointu ke kameře. Tento vektor je vždy žádoucí normalizovat. Normalizace je zapsána funkcí $normalize()$. Vektor $\overrightarrow{MC}$ je znázorněný na obrázku 4.13 modrou barvou a vypočítá se vztahem

$$\overrightarrow{MC} = normalize(\vec{C} - \vec{M})$$

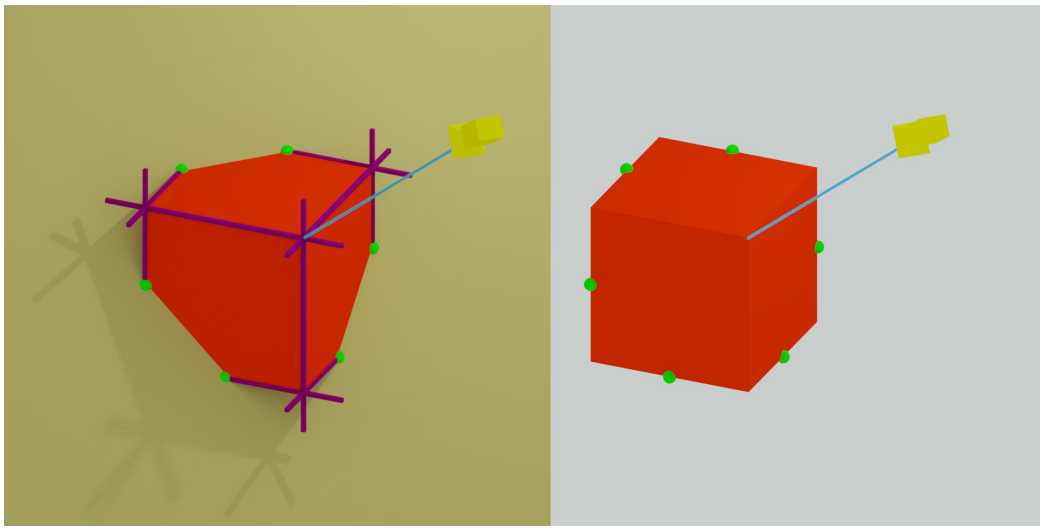
[8] https://en.wikipedia.org/wiki/Mipmap



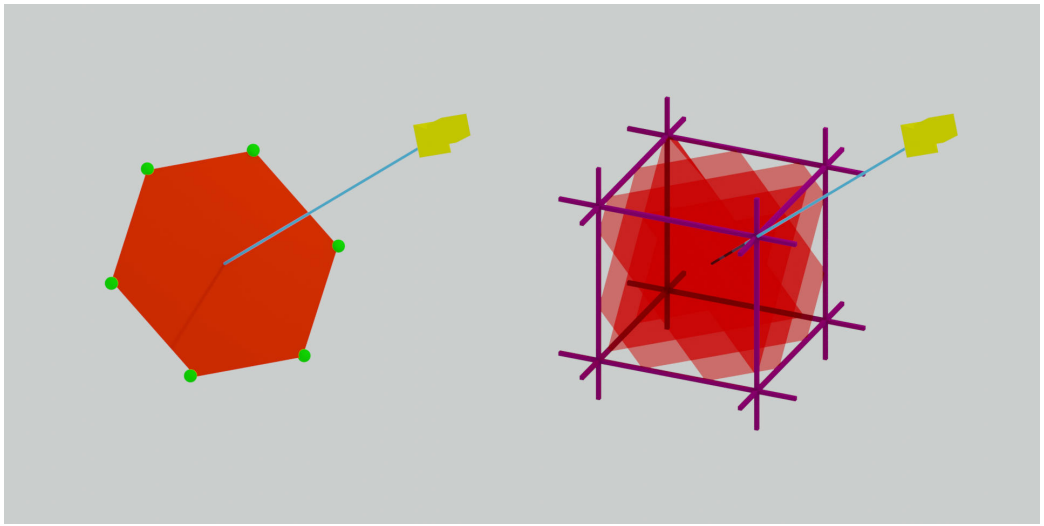
Obrázek 4.13: Nalevo je znázornění pomyslné krychle, obklopující polygonální model. Tato 3D krychle je zkonstruována za pomocí její úhlopříčky. Napravo je modrou barvou znázorněn vektor směřující od kamery do středu krychle. Kamera je vybarvena žlutou barvou.



Obrázek 4.14: Nalevo je znázornění původní krychle z obrázku 4.13 a roviny (vybarvena béžovou barvou), která je kolmá k vektoru směřujícímu od kamery ke středu krychle. Rovina protíná střed krychle. Napravo jsou zobrazeny pomyslné přímky ležící na hranách krychle (vybarveny fialovou barvou).



Obrázek 4.15: Nalevo obrázek doplněný o body průniku znázorněné roviny a krychle. Body průniku jsou znázorněny zelenou barvou. Tyto body průniku jsou vypočteny dosazení rovnic fialových přímek do rovnice roviny. Tyto body průniku krychle a roviny mohou tvořit polygon, od heaxagonu až po jeden jediný bod (případně rovina nemusí krychli protnout vůbec). Napravo je znázornění krychle a bodů průniku již bez roviny.



Obrázek 4.16: Nalevo je zobrazení hexagonu, který vznikne spojením všech šesti bodů průniku krychle a roviny. Tento hexagon musí být následně převeden na trojúhelníky. Znázornění převodu je na obrázku. Napravo je poté zobrazení výsledného prokládání krychle polygony, které vzniknou postupným posouváním kolmé roviny po diagonále krychle. Výsledkem je znázornění objemu voxel mapy na dvourozměrném plátně.

Nyní je k dispozici vše potřebné k sestrojení roviny znázorněné na obrázku 4.15 béžovou barvou. Obecná rovnice roviny v prostoru je daná vztahem

$$a * x + b * y + c * z + d = 0$$

kde a, b, c jsou souřadnice normálového vektoru roviny a x,y,z jsou souřadnice bodu ležícího na rovině. V tomto případě a, b, c jsou souřadnice vektoru $\overrightarrow{MC}$ a x, y, z jsou souřadnice středu krychle $\vec{M}$. Jedinou neznámou je d, kterou lze vypočítat vztahem

$$d = -(a * x + b * y + c * z) \Rightarrow$$
$$d = -(\overrightarrow{MC} \cdot \vec{M})$$

Dále je nutné vypočítat rovnice přímek ležících na hranách pomyslné krychle. Tyto přímky jsou znázorněny na obrázku 4.13 fialovou barvou. Parametrické vyjádření rovnice přímek je

$$x = a_1 + t * \vec{u_1}$$
$$y = a_2 + t * \vec{u_2}$$
$$z = a_3 + t * \vec{u_3}$$

kde x, y, z jsou souřadnice bodů ležících na přímce, a je výchozí bod ležící na přímce a vektor $\vec{u}$ je jejím směrovým vektorem. t nakonec značí posun po přímce. Tyto rovnice se dají rovněž přepsat do maticového tvaru:

$$\vec{P} = \vec{a} + t * \vec{u}$$

Jelikož má krychle 12 hran, je nutné vytvořit 12 rovnic přímek popisujících tyto hrany. Pro názornost bude vypsáno jen pár z nich. Jelikož se jedná o krychli, která není nijak nakloněná, či zkosená, všechny její rovnice hran jsou popsány některým z bodů ležících na přímce a vekterů popisujících směr některé ze souřadných os systémů. Pro určení například přední horní hrany, která je viditelná při kolmém pohledu na krychli, postačí znát bod v levém horním rohu krychle a směrový vektor souřadné osy x. Situace je znázorněna na obrázku 4.17.

V tomto případě se vypočítá bod znázorněný na obrázku 4.17 (a tedy vektor $\vec{a}$) za pomocí následujícího vztahu

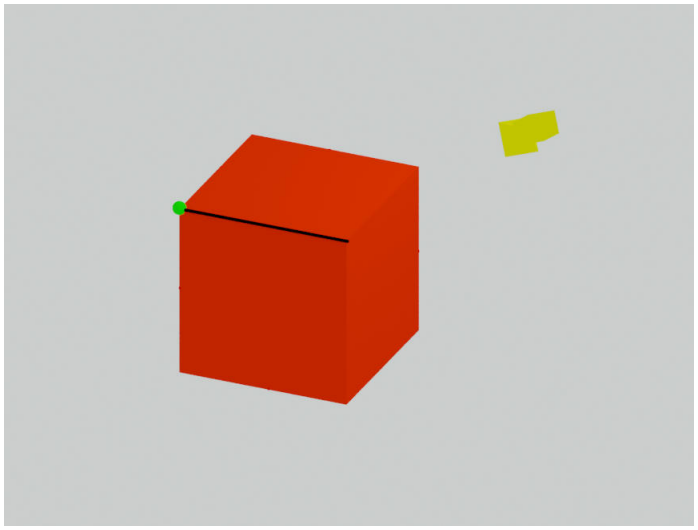
$$a_x = U_{minX}$$
$$a_y = U_{maxY}$$
$$a_z = U_{maxZ}$$

Hodnota směrového vektoru $\vec{u}$ je rovna směrovému vektoru osy X:

$$\vec{u} = (1.0, 0.0, 0.0)$$

Výsledná parametrická rovnice roviny P_1 je tedy

$$P_{1x} = U_{minX} + t * 1.0$$
$$P_{1y} = U_{maxY} + t * 0.0$$
$$P_{1z} = U_{maxZ} + t * 0.0$$



Obrázek 4.17: Znázornění konstrukce rovnice přímky. Pro sestrojení poslouží bod v levém horním rohu znázorněný zelenou barvou a směrnicový vektor osy x, znázorněný tmavě modrou barvou.

Stejným způsobem se vypočte všech 12 rovnic hran krychle. Jakmile jsou známy rovnice přímek a rovnice roviny, je možné vypočítat jejich průsečíky. Průsečíky se vypočítají dosazením rovnic P_{1x},P_{1y} a P_{1z} za hodnoty proměných x, y, z v rovnici roviny.

$$a * P_{1x} + b * P_{1y} + c * P_{1z} + d = 0$$

Po dosazení všech známých proměnných je získán vztah:

$$MC_x * (U_{minX} + t * 1.0) + MC_y * (U_{maxY} + t * 0.0) + MC_z * (U_{maxZ} + t * 0.0) - (\overrightarrow{MC} \cdot \vec{M}) = 0$$

Jak je z rovnice vidět, jedinou neznámou kterou zbývá dopočítat je proměnná t, která představuje právě ten posun po směrovém vektoru rovnice přímky P_1, ve kterém dojde k protnutí roviny. Úpravami předchozí rovnice je získán vztah pro výpočet proměnné t:

$$t = -\frac{(MC_x * U_{minX} + MC_y * U_{maxY} + MC_z * U_{maxZ} - (\overrightarrow{MC} \cdot \vec{M}))}{(MC_x * 1.0 + MC_y * 0.0 + MC_z * 0.0)}$$

Jak je možné vidět, jedná se většinou o součty součinů jednotlivých složek vektorů. Z toho vyplývá, že je lze nahradit skalárním součinem příslušných vektorů, jak tomu bylo učiněno například u proměnné d. V implementaci tak učiněno je hlavně z toho důvodu, že grafické karty mají hardwarové obvody pro skalární součin, které skalární součin vypočtou mnohem rychleji. Nicméně pro názornost a přehlednost je rovnice zachována v původní podobě.

Pro určení bodu průniku nyní zbývá dosadit proměnnou t do parametrických rovnic přímky P_1. Výsledkem je souřadnice bodu průniku. Tento postup je nutné celý opakovat dohromady 12 krát pro každou hranu krychle.

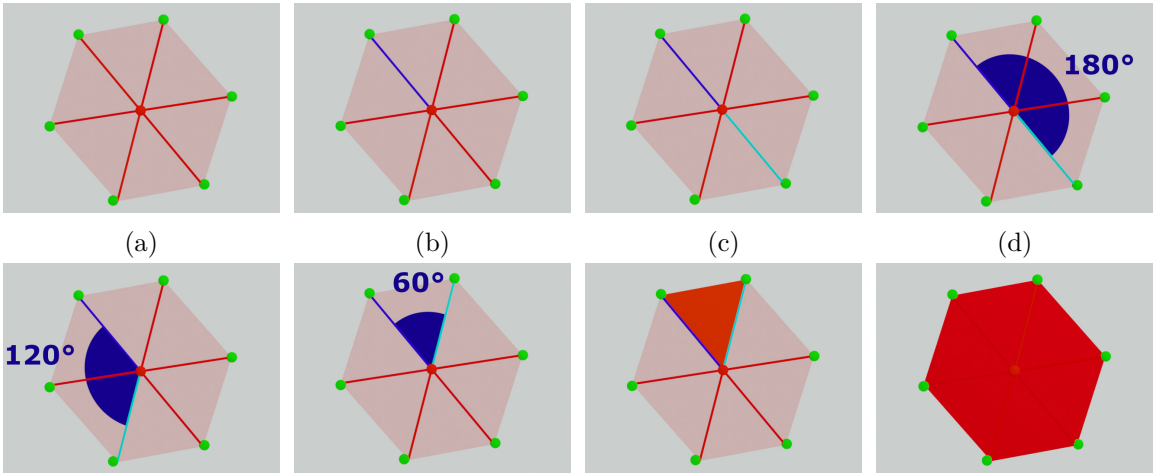
Nicméně ne všechny body průniku jsou ty hledané. Hledají se ty průsečíky, které zároveň leží na některé z hran krychle. Navíc je nutné zmínit, že ne vždy existuje průsečík dané roviny a přímky. Někdy mohou být vzájemně rovnoběžné a v takovém případě se nikdy neprotnou. Z množiny vygenerovaných jsou tak ponechány jen ty body, jejichž souřadnice se nachází jen v rozměrech krychle. S ostatními se dále nepočítá.

Jak již bylo zmíněno, API openGL pracuje pouze s trojúhelníky. Proto je nutné tento neznámý polygon převést na trojúhelníky. Toto také představuje problém sám o sobě, protože není jedno v jakém pořadí budou příslušné vertexy spojeny. Pokud by byly spojeny špatně, vzniknou nežádoucí díry v obrazci. Zde je algoritmus generování trojúhelníkových polygonů:

Algorithm 2: Algoritmus generování trojúhelníků

```
nastav midpoint na střed bodu rotace roviny;
ulož všechny vygenerované vertexy do pole pointsOut;
nastav pointsNumber na velikost pole pointsOut ;
definuj pole taken o velikosti pointsNumber;
for každý prvek v poli taken do
    nastav prvek na false;
end
nastav nextToTake na 0;
nastav emitedCounter na 0;
nastav closestAngle na nejvyšší možná hodnota;
nastav closestId na 0;
while emitedCounter je menší než pointsNumber do
    nastav taken[nextToTake] na true;
    if emitedCounter je rovno pointsNumber-1 then
        emituj vertex na pozici midpoint;
        emituj vertex na pozici pointsOut[nextToTake];
        emituj vertex na pozici pointsOut[0];
        ukonči primitivu;
    else
        for i =0; i < pointsNumber; i++ do
            if taken[i] je rovno false then
                sestroj vektor v1 v bodu počátku midpoint a konci
                  pointsOut[nextToTake];
                sestroj vektor v2 v bodu počátku midpoint a konci pointsOut[i];
                if úhel mezi v1 a v2 je menší, než closestAngle then
                    nastav closestAngle na aktuálně vypočtenou hodnotu úhlu;
                    nastav closestId na hodnotu i;
                end
            end
        end
        emituj vertex na pozici midpoint;
        emituj vertex na pozici pointsOut[nextToTake];
        emituj vertex na pozici pointsOut[closestId];
        ukonči primitivu;
    end
    nastav closestAngle na nejvyšší možnou hodnotu;
    nastav nextToTake na closestId;
    inkrementuj emitedCounter ;
end
```

Algoritmus je také znázorněn na obrázku 4.16 a dále.



(a) (b) (c) (d)

(e) (f) (g) (h)

Obrázek 4.18: Znázornění postupu převodu konkávního polygonu na množinu trojúhelníků, které dále může vykreslit grafická karta. Na obrázku (a) je červeným bodem znázorněn střed bodů průniku krychle a roviny. Zelenou barvou jsou znázorněny body průniku původních hran krychle a roviny. Úsečky vycházející z červeného bodu k zeleným bodům průniku poté tvoří hrany výsledných trojúhelníků sestavujících Ngon. Pro začátek algoritmu konstrukce trojúhelníků je nutné si zvolit výchozí úsečku. Na obrázku (b) je úsečka znázorněna modrou barvou. Po zvolení výchozí úsečky je třeba vybrat prvotní bod k měření úhlů. Jelikož nejsou body nijak seřazeny, je nutné vybrat body náhodně. Mezi náhodně vybraným a středovým bodem se sestrojí úsečka. Na obrázku (c) je tato úsečka znázorněna světle modrou barvou. Poté se mezi modrou a světle modrou úsečkou změří úhel, který mezi sebou svírají. Na obrázku (d) se jedná o úhel 180°. Tento proces se opakuje pro všechny zbývající body (znázornění některých dalších výběrů na obrázku (e) a (f)). Nakonec je vybrána úsečka znázorněna na obrázku (f). Úsečky mezi sebou svírají úhel 60°. Jelikož je úhel nejmenší, bude vybrán k utvoření trojúhelníku tento bod. (mohl by být vybrán i bod nacházející se po levici modré úsečky. V takovém případě záleží na zvoleném algoritmu, který rozhodne). Jakmile je vybrána trojice bodů, je vytvořen trojúhelník. Ze světle modré úsečky se stane tmavě modrá a z množiny zelených bodů se poté odstraní již vybraný bod. Vybírá se pouze z těch zbývajících. Tento postup se opakuje, dokud nejsou odstraněny všechny body z množiny. Výsledkem je 6 polygonů tvořících hexagon. Těchto 6 trojúhelníků dále může zpracovat grafická karta.

Tímto způsobem je možné vygenerovat jeden plát. Vertexům vygenerovaných trojúhelníků pak ještě zbývá přiřadit UVW kordináty. UVW kordináty jsou rovny pozici daného vertexu na krychli a normalizované vzhledem k rozměrům krychle. To znamená, že bod $\vec{U_{min}}$ má hodnotu UVW koordinátů (0,0,0) a $\vec{U_{max}}$ má hodnotu UVW koordinátů (1,1,1). Tyto vygenerované vertexy a jejich atributy se nakonec dostanou až ke fragment shaderu a ten přiřadí příslušnou barvu na základě UVW koordinátů z 3D textury.

Tento proces se celý musí opakovat několikrát. Počet opakování je roven rozlišení dané 3D textury. Při každé iteraci vykreslování plátu je nutné změnit bod, ve kterém se rovina nachází. Tyto body se pohybují po prvotní diagonále a vždy se posouvají na základě uniformní proměnné. Vždy je uložen index a maximální počet vykreslovaných plátů. Do geometry shaderu také přichází na vstup uniformní proměnné, které značí počet plátů a aktuální vykreslovaný plát. Na základě tohoto indexu se vypočte pozice, na které má být aktuální plocha zobrazena a to vztahem:

$$\vec{midpoint} = \vec{U_{min}} + (\vec{U_{max}} - \vec{U_{min}}) * (pocetPlatu/aktualniPlat)$$

Je pak jasné, že proces zobrazení touto metodou není úplně bez postihu. Použití geometry shaderu samo o sobě snižuje rychlost vykreslování jednotlivých polygonů.

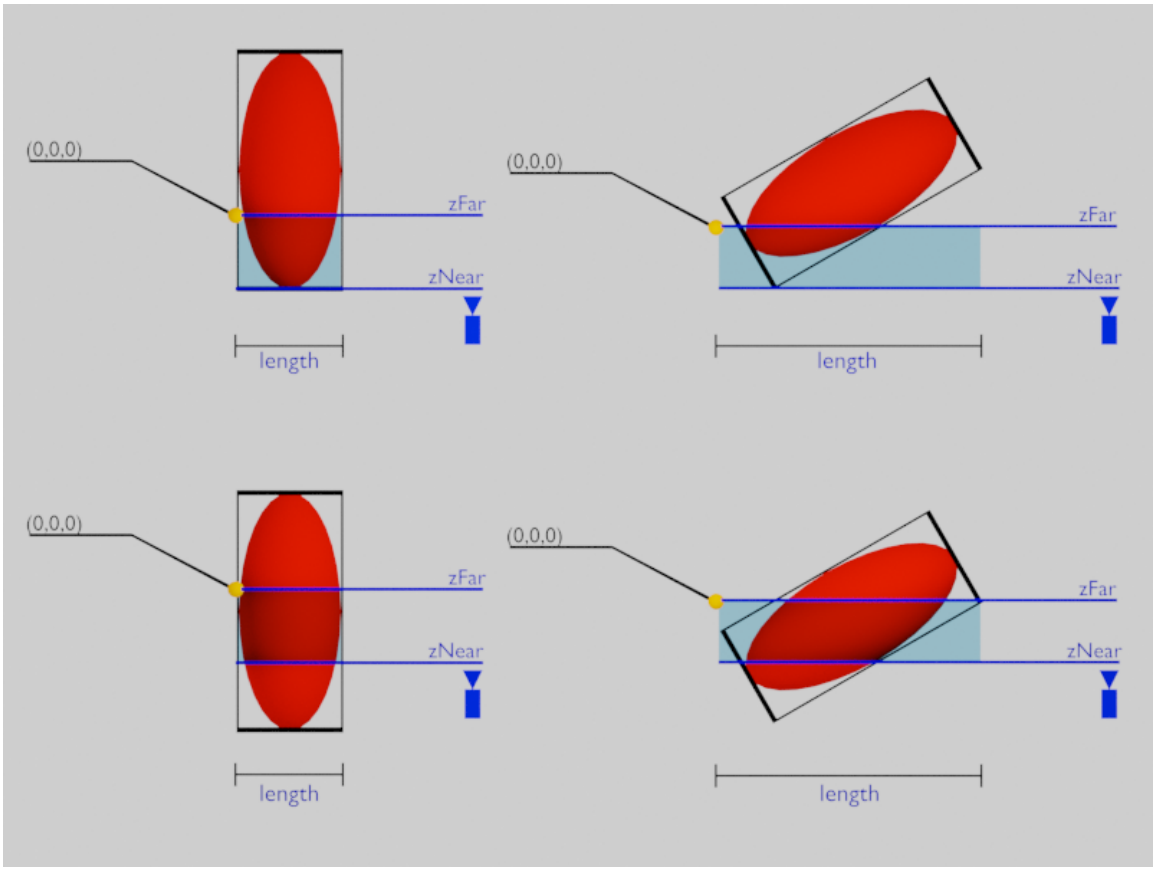
Tímto byla podrobněji probrána metoda převodu na objem. Nyní bude probrána metoda převodu na statické pláty, neboli metoda prolínání statických billboardů.

4.4.3 Metoda prolínání statických plátů

Nejprve je potřeba vytvořit textury jednotlivých plátů. Veggie obsahuje třídu pro uložení pole textur. Třída *scanner3d* naskenuje příslušný objekt do 3D textury, nebo do pole 2D textur. Poté se tato textura uloží do databáze, kterou zprostředkovává třída *TextureBank*. Třída přiřadí ukládané textuře jmený identifikátor a pomocí něj uloží vygenerovanou texturu do hešovací tabulky. Odtut může být vždy vyhledána a to jen za pomoci jejího jmenného identifikátoru. Kromě dat o textuře ukládá i hodnotu GLuint identifikátoru dané textury, což je odkaz na nahranou texturu v grafické kartě. Stejně jako u textur obarvující terén, i zde je pole textur reprezentováno pomocí GL_TEXTURE_2D_ARRAY. Aby bylo však možné uložit texturu, musí ji *scanner3d* nejprve vygenerovat. Nejprve se definuje, kolika pláty se má daný objekt reprezentovat. Množství plátů závisí na aktuálním generovaném LOD modelu. Objekt se vloží do bounding boxu, pomocí translace se posune přesně tak, aby zadní levá hrana bounding boxu ležela v počatku systému na pozici (0,0,0). Objekt se pomocí translace posune o převrácenou hodnotu počtu plátů (1/početPlátů) směrem do osy Z. Poté se nastaví projekční ortografická matice za pomoci *glm::ortho(left, right, bottom, top, zNear, zFar)* s následujícími parametry:

$$\begin{aligned} left &= 0.0 \\ right &= delkaObjektu \\ bottom &= 0.0 \\ top &= vykaObjektu \\ zNear &= 1/pocetPlatu * hloubkaObjektu \\ zFar &= 0.0 \end{aligned}$$

nyní se může objekt vykreslit. Po vykreslení je výsledná bitmapa uchována ve framebufferu, odkud jej lze získat pomocí funkce *glReadPixels*. Jednotlivé pixely se uloží do paměti

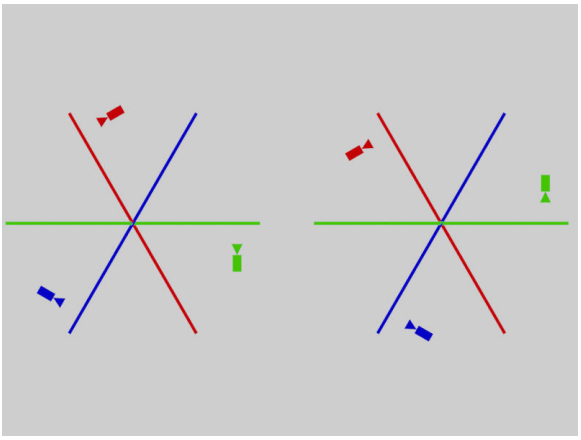


Obrázek 4.19: Znázornění procesu skenování. Nejprve se objekt umístí na pozici znázorněnou v levém horním rohu. Poté se naskenuje a posune se o vzdálenost danou počtem skenovaných plátů. Tento posun je znázorněný na levém spodním obrázku. Jakmile se se tento proces zopakuje Nkrát, kde N je počet plátu, přichází na řadu otočení objektu o -120°. Toto otočení je znázorněno na pravém horním obrázku. S otočením se mění rozměry bounding boxu objektu. Délka bounding boxu je znázorněna úsečkou *length*. Po naskenování opět dochází k posunu orotovaného objektu, jak je tomu znázorněno na pravém dolním obrázku.

a proces se s menšími změnami opakuje. Objekt se opět posune o další $1/poetPlt$ celkové hloubky objektu a opět se vykreslí. Mezi jednotlivými etapami vykreslování se samozřejmě promazává jak color buffer, tak depth buffer. Proces je znázorněn na obrázku 4.19.

Je důležité, aby se objekt posouval přesně tímto popsaným způsobem. Pokud například dojde k tomu, že se do projekční ortografické matice dostane záporná hodnota v ose Z, tak dojde k tomu, že bude špatně interpretována hloubka za pomocí depth bufferu a všechno, co je dále od pozorovatele, se může zobrazit blíže a překrýt přední fragmenty textury. Což je samozřejmě chyba.

Tento proces skenování a posunu objektu se opakuje tolikrát, kolikati pláty je reprezentován objekt. Poté se objekt orotuje o 120°, kdy dojde ke změně rozměrů bounding boxu. Nové rozměry se vypočítají následujícími vztahy:



Obrázek 4.20: Nalevo je znárornění pozice kamery vzhledem k objektu u rotace 0°, 120°, 240°. Napravo je znázornění jejich protilehlých projekcí.

$$Xlen_{new} = Xlen_{orig} * cos(\theta) + Zlen_{orig} * sin(\theta)$$
$$Zlen_{new} = Zlen_{orig} * cos(\theta) + Xlen_{orig} * sin(\theta)$$
$$\theta \in < 0, 90 > \cup < -180, -90 >$$

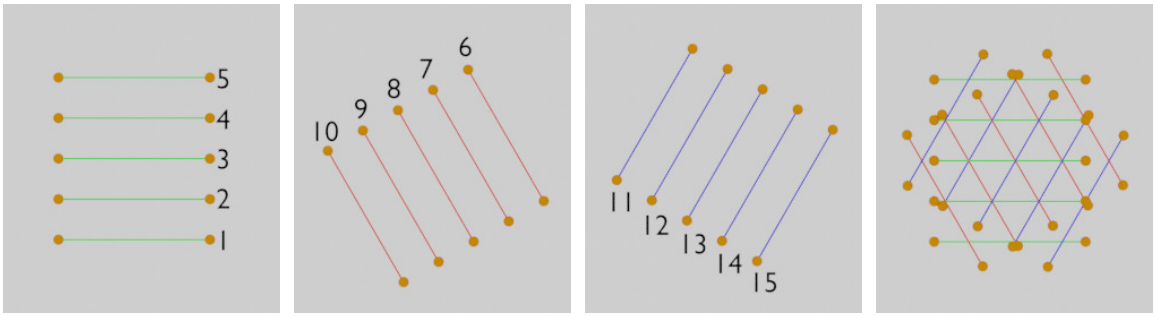
$$Xlen_{new} = Zlen_{orig} * cos(\theta) + Xlen_{orig} * sin(\theta)$$
$$Zlen_{new} = Xlen_{orig} * cos(\theta) + Zlen_{orig} * sin(\theta)$$
$$\theta \in (90, 180) \cup (-90, 0)$$

Pro LOD o pěti plátech dojde u orotovaného objetu opět k vytvoření pěti textur, které se přidají do pole a zvýší tak velikost celkového pole textur na 10. Nakonec se objekt orotuje o dalších 120° a opět se aplikuje stejný proces. Výsledkem je 15 textur objektu, reprezentujících projekci ze tří úhlů pohledu. Nyní se k těmto úhlům pohledu vytvoří jejich protilehlé náhledy na objekt. Na obrázku je znázornění jejich orotování 4.20.

Tyto protilehlé projekce vytvoří dalších 15 textur. Celkově tedy 30 textur pro pěti plátovou reprezentaci. V paměti jsou úmyslně seřazeny tímto způsobem pro snazší aplikaci ve vertex shaderu.

Nyní je pole textur připraveno a nahráno do paměti. Veggie vezme bounding box, rozřeže jej a vygeneruje vertexy jednotlivých plátu a indicie tvořící samotné polygony plátů. Nejprve vytvoří vertexy pro neorotovanou krychli, tyto vertexy zkopíruje a orotuje je pomocí rotační matice o 120 a 240 stupňů. Každý plát tedy tvoří 4 vertexy. Dohromady tedy vygeneruje 60 vertexů. Těmto 60 vertexům přiřadí UV koordináty. Navíc atribut UV koordinátu na textuře rozšíří o W koordinát, což je ve skutečnosti index do pole vygenerovaných textur. Na obrázku 4.21 je zobrazeno, jaké hodnoty W přiřazuje jednotlivým plátům. Jde vidět, že hodnoty nabývají rozmezí 1-15. Těchto 15 textur představuje textury pro úhly rotací 0°, 120° a 240°. Všem vertexům je také přiřazen normálový vektor. Textury 16-30 se nikam nepřiřazují z toho důvodu, že se jedná jen o opačné pohledy k texturám 1-15.

Ve vertex shaderu se změří úhel mezi normálovým vektorem a vektorem směřujícím od vertexu ke kameře. Pokud je skalární součin vektorů kladný, bude pro indexaci ve fragment



(a) (b) (c) (d)

Obrázek 4.21: Vytvoření plátů při pohledu ze shora. Oranžovou barvou jsou znázorněny viditelné vertexy. Popisky u čar značí index, tedy hodnutu W atributu. Na obrázku (a) jsou znázorněny vertexy orotované o 0°, na obrázku (b) o 120°, na obrázku (c) o 240° a na obrázku (d) je znázorněn útvar, který všechny vertexy finálně utváří.

shaderu používato rozmezí textur 1-15. Pokud je skalární součin záporný, k indexu textury se přičtě číslo 15, díky čemuž se změní rozsah na 16-30 a bude se vybírat z protilehlých textur. Je nutné zmínit, že pokud je skalární součin záporný, je třeba zrcadlově otočit UV koordináty textury. Poslední, co zbývá udělat v shaderech, je prolínání, kdy se průběžně mění průhlednost plátů podle úhlu mezi normálovým vektorem a vektorem směřujícím ke kameře.

4.4.4 Osvětlení modelů

Tímto byly probrány dvě použité metody pro převod původního polygonálního modelu na jinou reprezentaci. V této části se bude text stručně zabývat zobrazením původního modelu, který tvoří nejvyšší LOD. Pro osvětlení modelů je kromě samozřejmostí, jako je phongův osvětlovací model[9], použit jak již zmíněný normal mapping, tak environment mapping.

Normal mapping je považován za velmi známou metodu, proto bude probrán jen ve stručnosti. Jakmile je k dispozici normálová mapa, která reprezentuje množinu normálových vektorů, je nutné tyto vektory upravit. Normálová mapa je reprezentována v tzv. tangentním prostoru. Tangentní prostor tvoří osy TBN, kde N představuje osu směřující k ortograficky opačnému pohledu směrem ke kameře. Vertikální osa pak představuje B a horizontální osa T. To znamená, že rovina, která má být reprezentována normálovou mapou a podle které jsou zapečeny normálové vektory, leží na osách TB a N je osa směřující ke kameře. Toto je však jiný prostor, než ve kterém je reprezentován polygonální model. Jeho vertexy, kamera i pozice světla jsou reprezentovány v prostoru XYZ. Proto je nutné jeden, nebo druhý prostor převést do požadovaného prostoru. Z hlediska výpočetního výkonu je lepší převádět prostor XYZ do TBN, jelikož lze ty nejnáročnější výpočetní operace provádět ve vertex shaderu a náročné maticové operace přenechat na něm. Pokud by byl vybrán opačný případ, nebylo by možné použít interpolaci u fragment shaderu a maticové operace by se musely nechat na něm. Tímto byla stručně popsána implementace normal mappingu. Nyní se text bude zabývat environment mappingem.

Jelikož pro reprezentaci oblohy je použita ve veggie technologie cube mapy, lze tuto stejnou cube mapu využít při environment mappingu. Téměř každý objekt v reálném světě má schopnost odrážet světlo. I když většina objektů odráží světlo svého okolí jen v malé

[9] https://en.wikipedia.org/wiki/Phong_reflection_model

(a) (b)

Obrázek 4.22: Obrázek osvětlení kmenu stromu bez použití environment mappingu a s použitím environment mappingu. Pokud dojde ke změně textury prostředí (cube mapy), dojde i ke změně barvy objektu.

míře, i tento nepatrný odraz má značný dopad na celkový vzhled objektu ve scéně. Ve veggie se tento efekt odrazu světla získá za pomoci již zmíněné cube mapy. Nejprve se sestrojí vektor směřující od kamery směrem ke fragmentu. Tento vektor se poté tzv. odrazí od normálového vektoru fragmentu. V GLSL se tento odražený vektor vypočte pomocí funkce *reflect*. Tento vypočtený vektor poté poslouží jako index pro výběr barvy z dané cube mapy. Pokud by byla takto získaná barva fragmentu aplikována na celý objekt, vypadal by jako zrcadlo. To není efekt, který by se běžně očekával u rostlin. Lze ho však využít jiným způsobem. Pokud se objektu ponechá jeho původní barva například ze 70 % a zbylých 30 % barvy objektu se vynásobí právě barvou odraženého fragmentu, získá se velmi zajímavý a vizuálně uspokojivý efekt. Tento efekt je znázorněn na obrázku 4.22

4.5 Optimalizace

Byly popsány implementace metod ke zjednodušení polygonálních modelů a jejich osvětlení. Samy o sobě tyto metody vedou ke zvýšení výkonu při vykreslování na straně GPU. To však nemusí být dostatečně efektivní. Je žádoucí, aby CPU určilo, co je nutné předávat grafické kartě k vykreslení. K tomuto účelu slouží sada implementovaných metod, které budou v následujícím textu probrány. Konkrétně se jedná o metodu rozdělení na sektory (tiling), metodu pro řazení sektorů a metodu pro urychlení vykreslování velkého množství objektů na straně GPU nazývanou geometry instancing [1].

4.5.1 Rozdělení na sektory

Veggie celou scénu rozdělí na čtvercové sektory. Počet sektorů je určen makrem SECTOR_NUMBER v modelu aplikace. Defaultně je hodnota 10. To znamená, že se celý terén ve scéně rozdělí na 10 sektorů podélně i do hloubky. Dohromady terén tedy rozdělí na 100 sektorů. S touto hodnotou sektorů lze experimentovat a je docíleno různých výsledků

optimalizace. Toto bude podrobněji prozkoumáno v kapitole měření 5. Jak již bylo zmíněno v sekci návrhu, každá rostlina je reprezentovaná mimo jiné i polem pozic, na kterých se nachází. Při rozdělení na sektory dochází navíc k tomu, že se každá pozice v tomto poli, přiřadí do příslušného sektoru na terénu. Každá rostlina má dvourozměrné pole třídy std::vector o velikosti právě SECTOR_NUMBER. Během inicializace se projde celé pole pozic rostliny a roztřídí se její pozice do příslušných odpovídajících sektorů na mapě. Výsledkem třídění je tedy dvourozměrné pole vektorů pozic rostliny, které lze indexovat jednotlivými sektory v rozmezí od 0 do SECTOR_NUMBER-1. Pro získání pozic rostliny, např. v sektoru (10,10), tedy postačí znát jen jeho index. Díky tomu lze vytvořit příslušné vykreslovací funkce, které budou vykreslovat pouze daný sektor a ne celou scénu.

4.5.2 Geometry Instancing

Nyní se funkce pro vykreslování rozeberou podrobněji. Veggie má funkce pro vykreslení objektů klasickým způsobem. Tedy tak, že se vytvoří program pro vykreslení modelu, tomu se přiřadí modelová matice (obsahující translaci, rotaci a scaling) pomocí uniformní proměnné a ve vertex shaderu se vypočte finální pozice objektu (tedy jeho vertexů). Díky tomuto lze umístit květinu na její danou pozici. Informace o její pozici se uloží do modelové matice a ta se pošle vertex shaderu. Pokud je potřeba vykreslit více rostlin, je nutné aktivovat program, přiřadit mu upravenou modelovou matici a poté vykreslit model pomocí příslušné funkce draw.

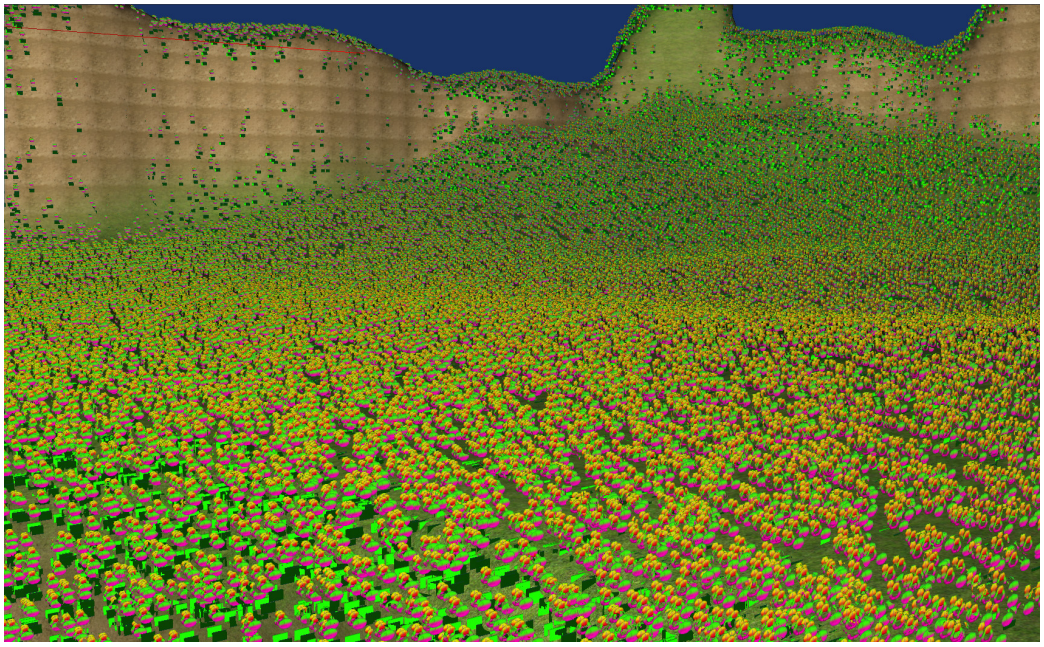
Tento způsob je velice pomalý a ve většině případů se nedá použít pro vykreslování velkého množství objektů. Je to z toho důvodu, že každé volání funkce draw je zdržení pro grafickou kartu, která musí přiřadit buffery, zvolit program a přiřadit příslušný vertex buffer objekt. To znamená velkou režii pro grafickou kartu. Výpočetní čas, který by se ušetřil redukcí polygonů modelu, by byl ztracen v režii.

Právě z toho důvodu existuje tzv. geometry instancing, u kterého se zavolá příslušná vykreslovací funkce jen jednou, přiřadí se jí počet opakování vykreslování a příslušné opereace, které jsou pro totožný model vždy stejné, se vykonají pouze jednou. Díky tomu dojde k znatelnému zvýšení výpočetního výkonu. Nicméně toto zajistí pouze rychlé renderování stejného objektu na stejné pozici. Je nutné nějakým způsobem odlišit pozice jednotlivých rostlin. Řešení by bylo, kdyby se do paměti grafické karty uložily informace o objektu a také informace o pozici jednotlivých instancí objektu. Přesně k tomu slouží atribut divisor[10].

Informace (jako pozice, UV koordináty, normálový vektor atd.) o vertexu jsou ve veggie uloženy v jednom bufferu a poskládány za sebou. Informace o pozicích jednotlivých objektů jsou uloženy v jiném samostatném bufferu. Buffer obsahující informace, které se obnovují jen pro každou instanci, nikoliv pro každý verex, se nazývá instance buffer object. Jelikož je terén rozdělen na několik sektorů, na kterých mohou růst stejné rostliny, ale na každém sektoru mají pouze jinou lokaci, je možné vytvořit pouze jeden VAO (Vertex Array Object)[11] pro každou rostlinu a měnit pouze instance buffer object v tomto VAO. Přesněji lze měnit odkaz atributu popisují informaci o pozici dané instance ve VAO. Tedy každá rostlina má pro všechny sektory stejné VAO, ale zároveň má pro každý sektor svůj vlastní instance buffer object. Při vykreslování jednotlivých sektorů tak dochází jen k jednomu zvolení vao pro rostlinu a pro každý sektor se mění pouze jeho attribut pointer. Toto vede k tomu, že

[10] https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glVertexAttribDivisor.xhtml

[11] https://www.khronos.org/opengl/wiki/Tutorial2:_VAOs,_VBOs,_Vertex_and_Fragment_Shaders_(C_/_SDL)



Obrázek 4.23: Použití geometry instancingu. Na scéně se nachází 100 000 objektů. Díky redukci režie při vykreslování je možné vykreslit velké množství objektů v reálném čase.

nevznikají v paměti duplicitní informace a zároveň se šetří režijní čas při změnách kontextu jednotlivých vao objektů.

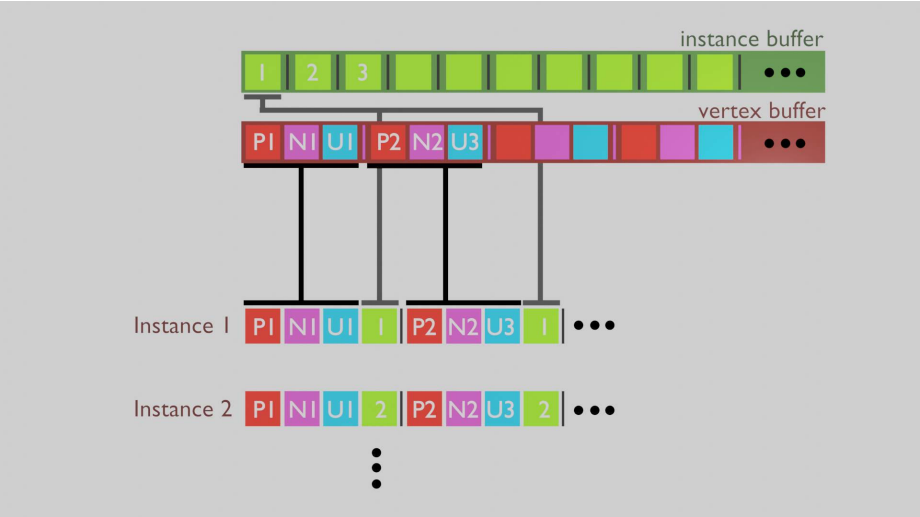
Celý tento postup je zahrnut v příslušné vykreslovací fukci v aplikaci veggie, kdy se zadá název rostliny a index sektoru, který se má vykreslit.

4.5.3 Ořezávání

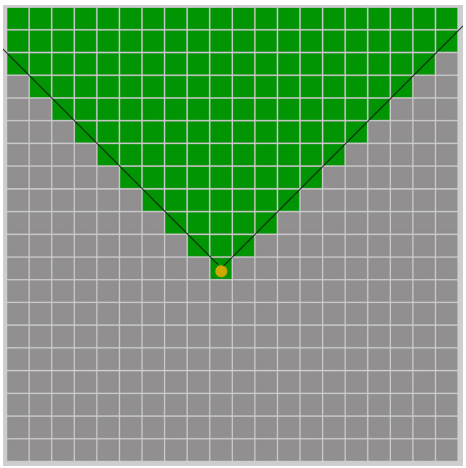
Nyní je implementace v takové fázi, kdy lze zjednodušit složitý polygonální model a vykreslit jej rychlým způsobem ve velkém množství. Měření ukázalo, že tyto metody vedou samy o sobě k zajímavým a již celkem dobrým výsledkům. Nicméně stále dochází k velkému plýtvání výpočetním výkonem na straně GPU. Stále totiž nebyl popsán způsob, jak zamezit zobrazování těch objektů ve scéně, které nejsou viditelné z pohledu kamery. Právě k tomuto slouží ořezávání, neboli anglicky culling.

Jelikož je scéna tvořena 100 sektory, lze zobrazit pouze ty sektory, které jsou viditelné z pohledu kamery. Z toho důvodu se před každým vykreslením sektoru kontroluje, jestli je viditelný. Algoritmus nejprve zjistí, ve kterém sektoru se nachází kamera. Poté převezme parametr popisující index, ve kterém se má nacházet právě zobrazovaný sektor. Mezi těmito dvourozměrnými indexy vytvoří vektor a změří úhel mezi tímto vektorem a vektorem směru pohledu kamery. Pokud je úhel do 45°, lze označit sektor jako viditelný. Proces je znázorněn na obrázku 4.25.

Tímto se snížila výpočetní náročnost pro GPU, ale zvýšila se režie pro CPU. Je důležité zvolit takový počet sektorů, který nepovede k příliš znatelnému zatížení CPU. Mohlo by se zdát, že čím více sektorů je použito, tím více se zvyšuje efektivita využití GPU, protože nemusí zobrazovat polygony, které nejsou zrovna viditelné. To je z části pravda, nicméně se nesmí zapomínat na fakt, že více sektorů vede i k více voláním funkce draw, což přidává



Obrázek 4.24: Znázornění uložení atributů v paměti GPU. Jednotlivé instance, jako například instance 1 a 2, mají jako vstup 4 atributy. Jedná se o pozici vetrexu, normálový vektor vertexu a UV koordináty, které jsou uloženy ve vertex bufferu. Tyto atributy jsou individuální pro každý vertex. K nim se přidá atribut instance (což může být právě pozice instance), který je pro dobu vykreslování instance neměnný, jak je tomu ukázano na obrázku.



Obrázek 4.25: Znázornění 20x20 sektorů. Šedou barvou jsou vybarveny ty sektory, které nejsou viditelné. Zelenou barvou jsou naopak znázorněny viditelné sektory. Oranžovou barvou je znázorněná pozice kamery. Kamera je na obrázku kolmo otočena směrem nahoru. Viditelné jsou pouze ty sektory, které svírají mezi kamerou úhel maximálně 45°.

režii. Pokud bude počet sektorů příliš velký, výpočetní čas režie překoná výpočetní čas, který byl ušetřen ořezáváním.

4.5.4 Řazení sektorů

Další metoda, jak urychlit výpočet grafické karty na úkor režie CPU, je řazení sektorů. Pokud je aktivní depth test, tak se při fragmentaci před vykonáním fragment shaderu testuje, zdali před fragmentem v depth bufferu již neexistuje nějaký fragment, který je blíže kameře, než právě zpracovávaný. Pokud se tam nachází, tak není program pro výpočet barvy fragmentu dokončen, ale fragment se zahodí a pokračuje se následujícím fragmentem. Toto zahození fragmentu taktéž zvyší efektivitu GPU.

Vegetace se po terénu generuje zcela náhodně. Její pozice jsou roztroušené po terénu a nejsou nijak seřazené směrem ke kameře. Z toho důvodu dochází k tomu, že se může nejdříve vykreslit objekt, který je nejdále od kamery, poté se vykreslí objekt, který je blíže kameře a nakonec se vykreslí objekt, který je nejblíže ke kameře. V takovém případě dochází k tomu, že je depth test vyhodnocen vždy jako pravdivý a vůbec nemusí docházet k zahazování fragmentů. Plýtvá se výkonem na to, co stejně bude přepsáno něčím jiným. A toto se může opakovat mnohokrát.

Z toho důvodu je vhodné mít objekty ve scéně seřazené vzhledem ke kameře. Ty objekty, které jsou nejblíže, by se měly zobrazovat jako první a ty objekty, které jsou nejdále, by se měly zobrazovat jako poslední. Nejefektivnější by bylo, kdyby byly seřazeny i samotné polygony, nikoliv jen instance objektů. U scény, která je tvořena jednotkami, maximálně desítkami objektů, by bylo možné tohoto docílit, nicméně pokud se bavíme o scéně, kde jsou miliony až desítky milionů objektů, jak je tomu v našem případě, tak toho docílit pravděpodobně možné není.

Z toho důvodu se řadí celé sektory, což sice není dokonalé a dochází tak pořád ke zbytečným vykreslováním v rámci sektoru, nicméně tato nedokonalost nepředstavuje takový problém, protože sektory jsou poměrně malé.

Pojem řazení však není úplně přesný. Pouze se nejdříve vykreslují ty sektory, které jsou ke kameře nejblíže, a to za pomocí algoritmu znázorněnému na obrázcích 4.26.

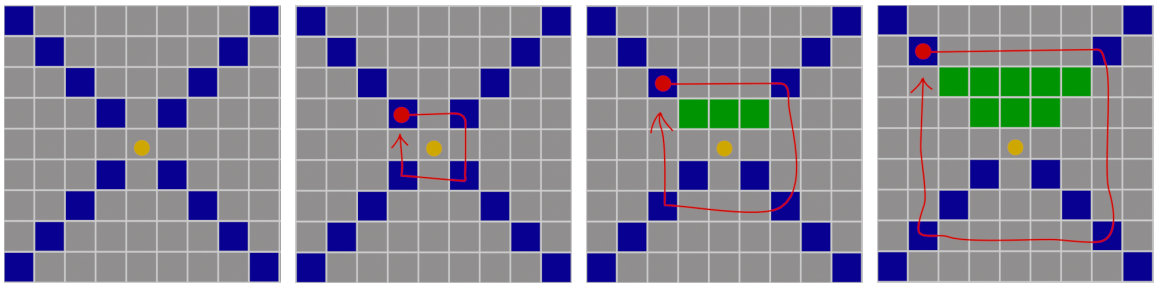
Tímto je fáze optimalizace téměř u konce. Poslední věc, kterou zbývá ve scéně udělat, je výběr algoritmu, kterým se bude daný sektor zobrazovat.

4.5.5 Level of detail

Jak bylo napsáno v úvodní sekci, použitá metrika pro výběr algoritmu pro reprezentaci LOD je vzdálenost od kamery. Čím blíže se objekt ke kameře nachází, tím vyšší je jeho detail. Nastavení úrovně detailu LOD je ve veggie odlišné u každého typu rostliny. Aplikace veggie pro určení LOD nevypočítává vzdálenost mezi kamerou a jednotlivými objekty, ale vypočítává vzdálenost mezi sektorem, na kterém se nachází kamera, a sektorem, na kterém se nachází objekt. To znamená, že ke změnám LOD dochází v rámci celého sektoru, nikoli v rámci konkrétního objektu.

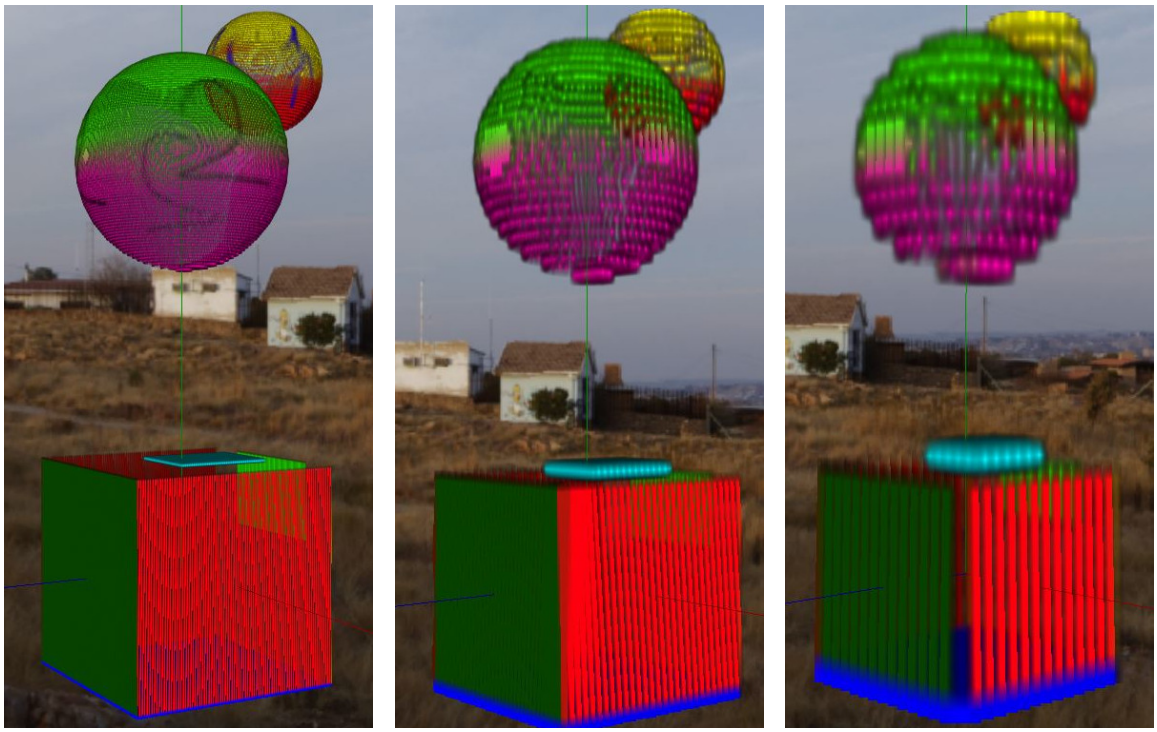
U metody převodu na objem je jednotlivých úrovní detailu dosaženo změnou rozlišení 3D textury. Se změnou rozlišení 3D textury se mění i počet plátů, kterými je objekt reprezentován. Pokud je rozlišení 3D textury 64x64x64, tak je objekt reprezentován 64 pláty.

U metody prolínání statických billboardů se jednotlivých úrovní LOD dosáhne změnou počtu billboardů, kterými je objekt nahrazen. Se změnou počtu billboardů se mění i počet textur, které jsou na billboard nanášeny. Toto je znázorněno na obrázku 4.28.



(a) (b) (c) (d)

Obrázek 4.26: Ukázka algoritmu zobrazování sektorů. Nejprve se utvoří diagonály ze směru od toho sektoru, na kterém se nachází kamera (obrázek (a)). Mezi těmito diagonálami se postupně vykreslují jednotlivé sloupce a řádky. Na obrázku (b) je znázorněna první iterace algoritmu. Nejprve se vykreslí sektor vyznačený červeným bodem. Odtud se kolem středového bodu kamery opíše čtverec. Na obrázku (c) je znázorněná druhá iterace algoritmu. Zelené sektory značí již vykreslené sektory. Algoritmus začíná v druhém sektoru na diagonále od středového sektoru. Na obrázku (d) je znázorněna třetí fáze algoritmu. Tento algoritmus se opakuje a končí až ve vzdálenosti rovné počtu sektorů po diagonále.



(a) (b) (c)

Obrázek 4.27: Znázornění úrovní detailu při použití metody převodu na objem. Na obrázku (a) je objekt reprezentován 3D texturou s rozlišením 256x256x256 voxelů, u obrázku (b) to je textura s rozlišením 64x64x64 voxelů a u obrázku (c) má textura rozlišení 32x32x32 voxelů.

(a) (b)

Obrázek 4.28: Znázornění úrovní detailu při použití metody prolínaní statických billboardů. Na obrázku (a) je použito pro účely demostrace 30 billboardů pro reprezentaci stromu. U obrázku (b) je použit pouze 1 billboard pro reprezentaci.

Je nutné podotknout, že ve finální aplikaci veggie se metoda převodu na objem nepoužívá. Je to z toho důvodu, že nepřinesla příliš pozitivní výsledky, jak bude poukázáno v následující kapitole.

Kapitola 5

Měření

Tato kapitola se zabývá finálním měřením implementovaných metod. Na základě získaných údajů z měření jsou poté utvořeny závěry a zhodnocení užitých metod.

5.1 Zhodnocení metody převodu na objem

Úvodem je třeba poukázat na výsledky implementované metody převodu na objem. Metoda převodu na objem nepřinesla žádné pozitivní výsledky. Z důvodu využití geometry shaderu a také z důvodu už samotného principu metody, který je výpočetně velmi náročný, není pravděpodobně zatím technicky možné tuto metodu použít přesně pro tento typ zobrazení vegetace. Původní myšlenka zobrazení vegetace touto metodou sice byla podložela výzkumem [4], nicméně je třeba podrobněji konkretizovat limity této metody. V daném výzkumu byl sice reprezentován strom společně s travnatým porostem, nicméně se jednalo pouze o jeden konkrétní strom, se kterým se dalo interagovat.

Aby bylo možné tuto metodu použít pro zobrazení velkého množství vegetace, je nutné vytvořit 3D texturu, nikoliv však jednoho kusu vegetace, ale 3D texturu například sektoru obsahujícího několik desítek až stovek kusů vegetace. Přesně tímto způsobem bylo docíleno zobrazení velkého množství vegetace v již zmíněném výzkumu Volumetric Billboards [4]. Toto však nebyl původní cíl práce. Cílem práce je zobrazení velkého množství vegetace takovým způsobem, aby bylo možné se k daným kusům vegetace přiblížit a aby došlo ke zvýšení detailu zobrazovaného modelu. Také bylo cílem, aby každá rostlina byla reprezentována svou vlastní pozicí. Implementace metody převodu na objem ukázala, že tuto metodu není možné tímto způsobem použít.

5.2 Zhodnocení metody prolínání statických billboardů

Pro prezentační účely i účely měření byla v aplikaci veggie vytvořena scéna společně s grafickým rozhraním pro úpravu atributů týkájících se jak vzhledu scény, tak změny příslušných parametrů LOD pro jednotlivé typy rostlin. Kromě toho je možné regulovat množství zobrazované vegetace. Scéna, která byla použita pro měření, je zobrazena na obrázku 5.1.

Scéna byla měřena celkově stokrát. Každé měření probíhalo po dobu 2-3 minut, kdy se měřila průměrná hodnota FPS. Padesát měření proběhlo na integrované grafické kartě Intel(R) HD Graphics 4600 a dalších 50 na grafické kartě NVIDIA GeForce GTX 765M. Grafy naměřených hodnot jsou zobrazeny na obrázcích 5.2 a 5.3.

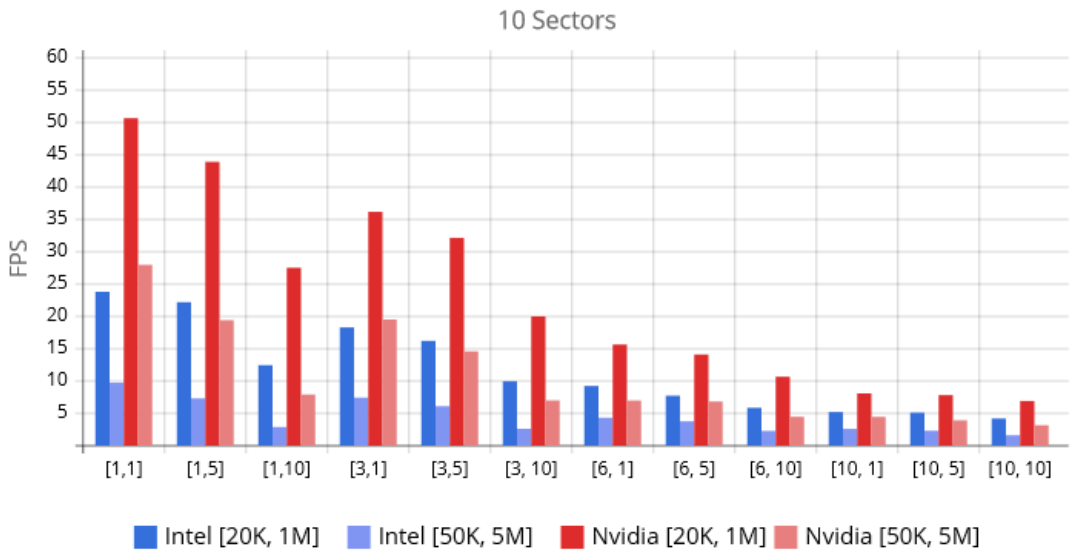
Obrázek 5.1: Obrázek scény, která byla použita při měření. Všechny následující popsané hodnoty, které byly získány měřením, se vztahují k tomuto pohledu. Změnou LOD se vzhled scény při každém měření může trochu změnit.

Celou tabulku naměřených hodnot lze nalézt v příloze. Z naměřených hodnot vyplývá, že pokud nedojde k rozdělení scény na sektory (údaje s hodnotou *Sectors* rovnou *1*), tak se nedá hovořit o real-timovém vykreslování scény. Hodnoty FPS dosahují maximálně jednotek. To je způsobeno tím, že pokud se vykresluje pouze jeden sektor, vykresluje se veškerá vegetace najednou. V takovém případě není použit culling, ani řazení sektorů.

Při rozdělení scény na 10 sektorů a bez použití metody prolínání statických billboardů se průměrné hodnoty FPS zvýšily u obou grafických karet 2-3 násobně. To je způsobeno tím, že se část vykreslování nahradila režií na straně CPU. Aplikací metody prolínání statických billboardů došlo ke znatelnému zvýšení hodnot FPS.

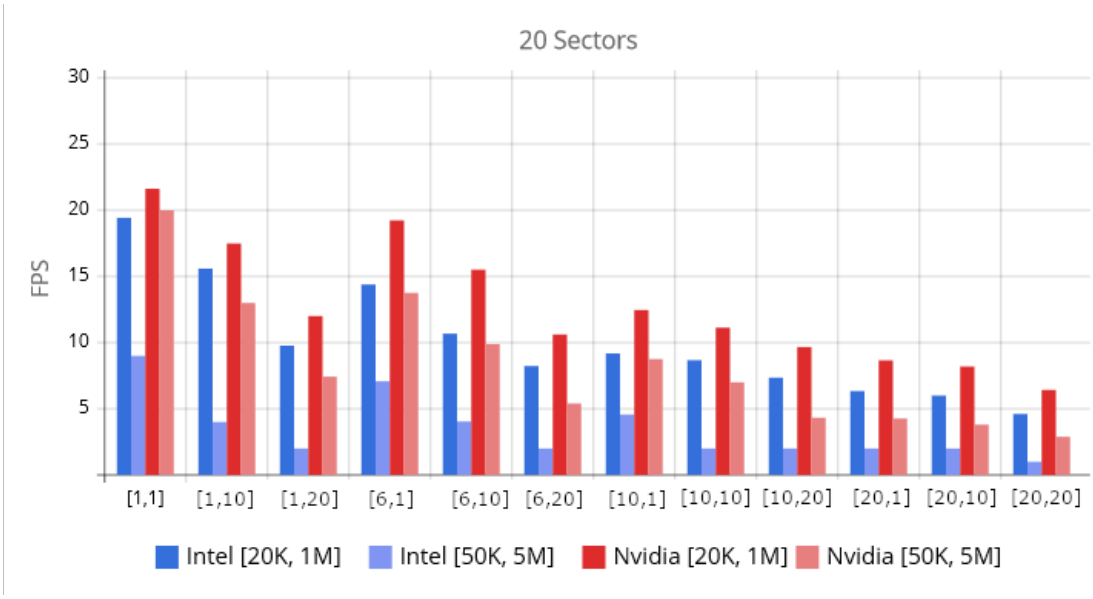
Kamera taktéž byla ve stejné scéně umístěna do středu lesa mezi jednotlivé stromy. Zde FPS dosahovaly hodnoty 20-25. Nižší FPS je způsobeno především vysokým počtem polygonů, které jsou velmi blízko kameře.

Je nutné podotknout, že při měření se taktéž objevily výpočetní limity samotného CPU. Jádro CPU které zodpovídalo za režii a výběr sektorů k vykreslování, bylo znatelně vytíženo už při rozdělení scény na 10 sektorů (tudíž celkově 100 sektorů). Hodnota vytížení jádra CPU byla přibližně 70-80 %. U rozdělení scény na 20x20 sektorů už bylo vytížení jádra téměř 100 %. Nicméně ostatní jádra CPU nebyla vytížena téměř vůbec. Pokud by tedy režie byla lépe rozdělena mezi jednotlivá jádra, je pravděpodobné, že optimální hodnota počtu sektorů by byla vyšší než 10 a možná i vyšší než 20. Pro konfiguraci systému, hardwaru i implementaci použitých při měření se však ukázalo, že je efektivnější použít pouze 10 sektorů místo 20 sektorů. Původně se očekávalo, že optimální hodnota bude kolem 40 sektorů, nicméně měření tuto domněnku neprokázala. Dalším velmi zajímavým zjištěním bylo, že grafická karta Intel lépe zvládá režii, než grafická karta Nvidia. U grafické karty Nvidia docházelo k nejlepším výsledkům při nízké režii a vyššímu počtu zobrazovaných polygonů (fragmentů), zatímco grafická karta Intel zvládala režii lépe, nicméně zaostávala ve vykreslování samotné grafiky. Tento efekt jde nejlépe vidět na měření, kdy bylo použito 20

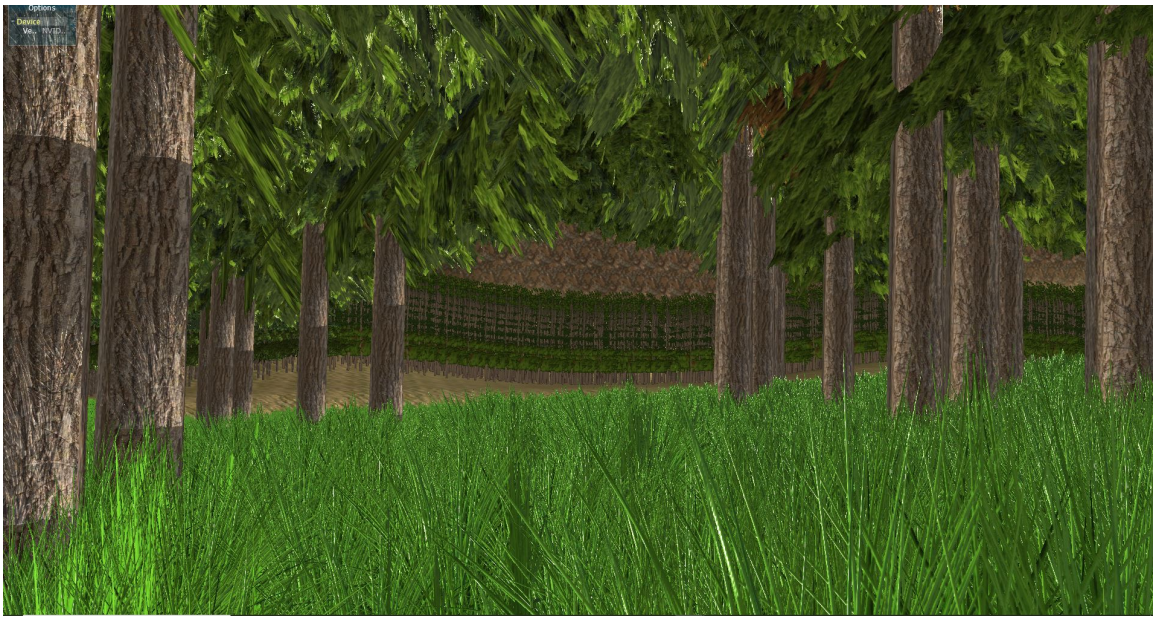


Obrázek 5.2: Graf zobrazující hodnoty FPS zaznamenaných během měření při rozdělení scény na 10x10 sektorů. U popisků na ose X jsou ve hranatých závorkách zaznamenány hodnoty dvou parametrů použitých při určování LOD. První parametr značí vzdálenost v sektorech od kamery, ve které se zobrazují modely velkých a středních rostlin původním polygonálním modelem. Všechny rostliny za touto hranicí se zobrazují metodou prolínaní statických billboardů. Druhý parametr určuje hranici vzdálenou v sektorech od kamery, po kterou se vykreslují původní polygonální modely malých rostlin. Za touto pomyslnou hranicí se malé rostliny nevykreslují. U popisků sloupců grafu se nachází název použité grafické karty při měření a dvojice hodnot značících počty zastoupení rostlinstva ve scéně během měření. První hodnota značí sumu velkých a středně velkých rostlin. Druhá hodnota značí sumu počtu všech malých rostlin.

sektorů a ve scéně se nacházelo 1 000 000 kusů trav a 20 000 stromů. Pokud by byla zobrazena veškerá vegetace, dá se hovořit o přibližně 20M polygonech nacházejích se ve scéně. Při tomto měření hodnoty FPS u grafické karty Intel a Nvidia dosahovaly podobných výsledků. Naměřené hodnoty (19,43; 15,59; 9,77) u Intelu se příliš neliší od hodnot (21,62; 17,49; 12,01) u Nvidie. Nicméně pokud došlo k navýšení počtu polygonů na 5 000 000 kusů trav a 50 000 stromů, objevil se už znatelný rozdíl. Stejné měření jen s vyšším počtem polygonů, vedlo na hodnoty FPS u Intelu (9,69; 4,79; 2,73) a u Nvidie na hodnoty (20,72; 13,24; 7,43). Nyní je rozdíl mezi grafickými kartami znatelný a zároveň se ukázalo, že mnohonásobně vyšší počet polygonu u grafické karty Nvidia nepřinesl tak znatelné snížení FPS, jako tomu bylo u Intelu.



Obrázek 5.3: Graf zobrazující hodnoty FPS zaznamenaných během měření při rozdělení scény na 20x20 sektorů. Význam jednotlivých popisků je totožný, jako u grafu 5.2.



Obrázek 5.4: Kamera umístěna ve středu lesa. Je vidět zobrazení velmi husté trávy. LOD stromů se při dostatečné vzdálenosti od kamery zvýší.

Obrázek 5.5: Ukázka stejné scény jako na obrázku. po pužití metody prolínání statických billboardů. FPS se zvýšily z 5,9 na 36,18.

Kapitola 6

Závěr

Jak již bylo zmíněno, metoda převodu na objem se sice prokázala jako nevyhovující, nicméně pokud by se nepřeváděly do 3D textury jednotlivé modely, ale přímo celé sektory, tak by tato metoda měla smysl a dala by se teoreticky použít. Samozřejmě takové množství 3D textur by bylo velmi paměťově náročné. Proto by bylo nutné použít kompresi 3D textury, což už je implementováno v samotném hadrwaru dnešních grafických karet. V takovém případě by pravděpodobně metoda mohla přinést zajímavější výsledky. Tyto změny by mohly být námětem na případné pokračování práce.

Metoda prolínání statických billboardů oproti metodě převodu na objem přinesla pozitivní výsledky. Tuto metodu lze tedy použít pro zobrazení velkého množství rostlin.

Dalším námětem na rozšíření jsou animace. Samotné modely by se daly rozšířit o animace například pomocí kosterních animací. Tyto kosti by mohly být rozpohybovávány pomocí simulace větru. Mohlo by se určovat, jak rychle a jakou silou se má s danými kostmi pohybovat. Díky tomu by teoreticky vznikl pohyb připomínající pohyb reálných rostlin.

Během měření také bylo poukázáno na špatné rozdělení zátěže CPU mezi jednotlivá jádra, kdy se většina optimalizačních metod vykonávala pouze na jednom jádře. Pokud by se vykonávání těchto metod paralelizovalo mezi jednotlivá jádra, pravděpodobně by mohlo dojít ke zvýšení efektivity, což by mohlo vést k vyšším hodnotám FPS při vykreslování.

Literatura

[1] Bao, G., Meng, W., Li, H., Liu, J. a Zhang, X. Hardware Instancing for Real-Time Realistic Forest Rendering. In: *SIGGRAPH Asia 2011 Sketches*. New York, NY, USA: Association for Computing Machinery, 2011. SA '11. DOI: 10.1145/2077378.2077398. ISBN 9781450311380. Dostupné z: https://doi.org/10.1145/2077378.2077398.

[2] Behrendt, S., Colditz, C., Franzke, O., Kopf, J. a Deussen", O. *Realistic real-time rendering of landscapes using billboard clouds*. 3, 1. vyd. Dresden, Saxony, Germany: University of Konstanz, 2005. Dostupné z: http://graphics.uni-konstanz.de/publikationen/Behrendt2005Realisticrealtime/Behrendt2005Realisticrealtime.pdf.

[3] Blinn, J. F. Simulation of Wrinkled Surfaces. In: *Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA: Association for Computing Machinery, 1978, s. 286–292. SIGGRAPH '78. DOI: 10.1145/800248.507101. ISBN 9781450379083. Dostupné z: https://doi.org/10.1145/800248.507101.

[4] Decaudin, P. a Neyret, F. Volumetric Billboards. *Computer Graphics Forum*. 2009, sv. 28, č. 8, s. 2079–2089. Dostupné z: http://phildec.users.sf.net/Research/VolBlbCGF09.php.

[5] Drebin, R. A., Carpenter, L. a Hanrahan, P. Volume Rendering. *SIGGRAPH Comput. Graph.* New York, NY, USA: Association for Computing Machinery. červen 1988, sv. 22, č. 4, s. 65–74. DOI: 10.1145/378456.378484. ISSN 0097-8930. Dostupné z: https://doi.org/10.1145/378456.378484.

[6] Jakulin", A. Interactive Vegetation Rendering with Slicing and Blending. 1. vyd. Ljubljana, Slovenia: Faculty of Computer and Information Science, University of Ljubljana, Ljubljana, Slovenia. 2000, č. 3. Dostupné z: http://www.stat.columbia.edu/~jakulin/slicing-and-blending/trees-electronic.pdf.

[7] Kligard, M. A Practical and Robust Bump-mapping Technique for Today's GPUs. Monroe StreetSanta Clara: [b.n.]. Říjen 2001, č. 1. Dostupné z: https://www.cg.tuwien.ac.at/courses/Realtime/slides/VU.WS.2013/PracticalBumpMap.pdf.

[8] Luebke, D., Reddy, M., Cohen, J. D., Varshney, A., Watson, B. et al. *Level of Detail for 3D Graphics*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002. ISBN 9780080510118.

[9] Marsalek, L., Hauber, A. a Slusallek, P. High-speed volume ray casting with CUDA. In: *2008 IEEE Symposium on Interactive Ray Tracing*. 2008, s. 185–185. DOI: 10.1109/RT.2008.4634648.

[10] Meissner, M., Pfister, H., Westermann, R. a Wittenbrink, C. Volume Visualization and Volume Rendering Techniques. In: *Eurographics 2000 - Tutorials*. Eurographics Association, 2000. DOI: 10.2312/egt.20001035. ISSN 1017-4656.

[11] Minařík", A. *Kosterní animace pro GPUengine*. Brno, 2019. Master thesis. Vysoké učení technické v Brně. Dostupné z: https://www.vutbr.cz/www_base/zav_prace_soubor_verejne.php?file_id=197157.

[12] Segal, M. a Akeley, K. *The OpenGL Graphics System: A Specification* [online]. 2019 [cit. 2021.04.17]. Dostupné z: https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.core.pdf.

[13] Reeves, W. T. a Blau, R. Approximate and Probabilistic Algorithms for Shading and Rendering Structured Particle Systems. *SIGGRAPH Comput. Graph.* New York, NY, USA: Association for Computing Machinery. červenec 1985, sv. 19, č. 3, s. 313–322. DOI: 10.1145/325165.325250. ISSN 0097-8930. Dostupné z: https://doi.org/10.1145/325165.325250.

[14] Sturman, D. Interactive key frame animation of 3-D articulated models. In: *Graphics Interface*. 1984, sv. 86 [cit. 2021.04.17]. Dostupné z: http://graphicsinterface.org/wp-content/uploads/gi1984-6.pdf.

Příloha A

Tabulka měření

Tabulka obsahuje sloupec *Sectors* značící počet sektorů na které byla scéna rozdělena, sloupec *AmountS* značící počet kusů malé vegetace na scéně (tráva), sloupec *SectorL* značící vzdálenost v sektorech od kamery, ve kterých se pro vykreslení použije originální model velkých a středních rostlin, sloupec *SectorS* mající stejný význam jako *SectorL*, nicméně se jedná o mále rostliny, sloupec *FPS Intel* značící počet naměřených FPS na grafické kartě Intel a sloupec *FPS Nvidia* značící počet naměřených FPS na grafické kartě Nvidia.

Tabulka A.1: Tabulka naměřených hodnot

Sectors	AmountS	AmountL	SectorL	SectorS	FPS INTEL	FPS NVIDIA
1	1 000 000	20 000	1	1	3,71	5,90
1	5 000 000	50 000	1	1	1,23	2,23
10	1 000 000	20 000	1	1	23.81	50,67
10	1 000 000	20 000	1	5	22.21	43,92
10	1 000 000	20 000	1	10	12.46	27,54
10	1 000 000	20 000	3	1	18.31	36,18
10	1 000 000	20 000	3	5	16.22	32,16
10	1 000 000	20 000	3	10	9.98	20.02
10	1 000 000	20 000	6	1	9.25	15.66
10	1 000 000	20 000	6	5	7.75	14.12
10	1 000 000	20 000	6	10	5.87	10.67
10	1 000 000	20 000	10	1	5.22	8.1
10	1 000 000	20 000	10	5	5.13	7.83
10	1 000 000	20 000	10	10	4.21	6.92
10	5 000 000	50 000	1	1	9.76	27.98
10	5 000 000	50 000	1	5	7.33	19.42
10	5 000 000	50 000	1	10	2.9	7.92
10	5 000 000	50 000	3	1	7.45	19.53
10	5 000 000	50 000	3	5	6.12	14.61
10	5 000 000	50 000	3	10	2.63	7.01
10	5 000 000	50 000	6	1	4.33	7,72
10	5 000 000	50 000	6	5	3.79	6,85
10	5 000 000	50 000	6	10	2.29	4,49
10	5 000 000	50 000	10	1	2.61	4,47
10	5 000 000	50 000	10	5	2.32	3,93
10	5 000 000	50 000	10	10	1.63	3,18
20	1 000 000	20 000	1	1	19.43	21.62
20	1 000 000	20 000	1	10	15.59	17.49
20	1 000 000	20 000	1	20	9.77	12.01
20	1 000 000	20 000	6	1	14.39	19.23
20	1 000 000	20 000	6	10	10.68	15.51
20	1 000 000	20 000	6	20	8.23	10.61
20	1 000 000	20 000	10	1	9.18	12.47
20	1 000 000	20 000	10	10	8.67	11.13
20	1 000 000	20 000	10	20	7.35	9.67
20	1 000 000	20 000	20	1	6.34	8.66
20	1 000 000	20 000	20	10	6	8.19
20	1 000 000	20 000	20	20	4.61	6.43
20	5 000 000	50 000	1	1	9,69	20,72
20	5 000 000	50 000	1	10	4,79	13,24
20	5 000 000	50 000	1	20	2,73	7.43
20	5 000 000	50 000	6	1	7.08	13.76
20	5 000 000	50 000	6	10	4.04	9.89
20	5 000 000	50 000	6	20	2,52	5.4
20	5 000 000	50 000	10	1	4.57	8.76
20	5 000 000	50 000	10	10	2,92	7
20	5 000 000	50 000	10	20	2,01	4.33
20	5 000 000	50 000	20	1	2,54	4.27
20	5 000 000	50 000	20	10	2,21	3.8
20	5 000 000	50 000	20	20	1,61	2.89

Příloha B

Obsah přiloženého média

```
/
├── externals
├── libs
├── resources
├── source
├── documentationSource
├── bin
├── CMakeLists.txt
├── readme.txt
└── prezVid.mp4
```

Médium obsahuje soubor **prezVid.mp4**, což je video prezentující aplikaci i implementované algoritmy. Soubor **readme.txt** obsahuje návod k instalaci a další informace. Soubor **CMakeLists.txt** slouží pro vytvoření konfigurace pro překlad za pomocí cmake. Složka **externals** obsahuje všechny knihovny třetích stran, které jsou potřeba pro úspěšný překlad aplikace. Některé mají k dispozici svůj vlastní soubor cmake pro konfiguraci, jiné jej nemají. Tyto knihovny je potřeba nejprve přeložit a poté předat aplikaci cmake. Složka **libs** obsahuje zdrojové soubory některých knihoven. Složka **source** obsahuje zdrojové soubory této práce. Složka **documentationSource** obsahuje zdrojové soubory potřebné pro vygenerování tohoto dokumentu. Složka **bin** obsahuje přeložené binární aplikace pro windows i linux (ubuntu).