Západočeská univerzita v Plzni

Fakulta aplikovaných věd

Katedra informatiky a výpočetní techniky

Diplomová práce

Vizualizace časově proměnných objemových dat

Plzeň, 2012 Martin Prantl

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 9. května 2012

Martin Prantl

Abstract

This work discusses problems related to visualization of voluemtric data in static images and time-varying data sets. Whole work is divided into four main parts. In the first part it is described theory of GPGPU programming. Next part describes loseless and lossy data compression. Third part is focused on static volume data images visualization. Several techniques are discussed and tested. Last part of the work describes existing solutions and discuss new proposed technique for time varying data sets compression and vizualization. Proposed method is based on classic compression algorithm, like Huffman encoding or LZ types of compression. Those algorithms are followed with classic ray-casting algorithm for data visualization.

Obsah

1 Úvod

Vizualizace volumetrických, neboli objemových dat, je v dnešní době velmi důležitým odvětvím. Tato data si můžeme představit jako trojrozměrné obrázky. Jednotkou těchto dat je voxel (jedná se o složeninu anglických slov *volumetric element*, česky objemový prvek), který je analogií pixelu z 2D grafiky.

Cílem metod je převést data z voxelové podoby do zobrazitelného modelu. Zdroje objemových dat mohou být velmi rozdílné. Data mohou být vyprodukována na základě matematického modelu - chování určitých fyzikálních veličin (simulace kapalin) nebo se s nimi setkáváme v medicíně, jako výstup CT (počítačové tomografie). Lékařství je v současné době asi hlavním odvětvím, kde se s tímto typem dat setkáváme v největší míře.

Základním cílem vizualizace volumetrických dat je umožnit uživateli jejich prohlížení klasickými zařízeními, nejčastěji monitory. Proto je potřeba data transformovat do průmětny obrazovky. V zásadě existují dva základní rozdílné přístupy, které lze různým způsobem kombinovat.

První pracuje s extrakcí jedné konkrétní iso-plochy a vzniká tak prostorová trojúhelníková síť, se kterou lze dále pracovat klasickým způsobem jako s polygonálním modelem. Například budeme-li mít voxelový model člověka, bude obsahovat veškeré informace o tkáních, tekutinách, kostech apod. Vizualizovat a prohlížet chceme ovšem pouze kostru. Nastavíme proto konkrétní práh a pomocí vhodných metod získáme extrakcemi iso-ploch pouze model kostry. Nejznámějším zástupcem této kategorie je pravděpodobně metoda Marching Cubes a její různé deriváty.

Druhý typ je univerzálnější a umožňuje zobrazení celého objemu. V případě vizualizace člověka bychom viděli průhlednou pokožku, cévy, svalstvo a nakonec pevnou kostru. Samozřejmě záleží na naší primární oblasti zájmu, takže není problém mít průhlednou také kostru. Metody založené na tomto principu vycházejí ze sledování paprsků. V praxi se tyto metody označují jako ray-casting nebo ray-tracing. Pro volumetrická data se častěji používají metody ray-castingu, kdy nepočítáme odražené paprsky, ale pracujeme pouze s primárním paprskem. Ten prochází objem a kumuluje získané hodnoty.

Hlavní problém všech metod pro vizualizace volumetrických dat je rozlišení dat. V případě klasických 2D obrázků na tento problém narážíme v menším rozsahu, ovšem u volumetrických dat se jedná o zásadní problém. Rozlišení 1024x1024x1024 nám již obsadí v nekomprimované podobě 1GB paměti. Přitom takovéto rozlišení, bereme-li v potaz porovnání s 2D obrazovými daty, není nikterak vysoké. Pro medicínské potřeby navíc rozlišení hraje zásadní roli, kdy vlivem nízké kvality dat může dojít i ke ztrátě zásadních informací.

Zatím jsme uvažovali pouze statická data, řekněme jednotlivé "obrazy". Ovšem často potřebujeme pracovat také s daty, která jsou časově proměnná (vezměme si například animace). Budeme-li pracovat s výše zmíněným rozlišením, dostáváme se do značných velikostí dat. Například pro video bychom na jednu jedinou sekundu spotřebovali při toku 25 snímků za vteřinu (což je považováno za plynulý obraz), 25GB v nekomprimované podobě. Takovéto datové objemy jsou pro současné generace počítačů a grafických karet stále velké pro zpracovávání v reálném čase.

2 Seznam použitých pojmů

Tato kapitola slouží jako abecední seznam zkratek a základních pojmů použitých v textu práce. Většina pojmů je definována také průběžně v textu, tato kapitola pouze usnadňuje orientaci v textu práce.

Alfa-kanál. Jeden z prvků textury nebo obrazu, obsahující informace o průhlednosti dat.

Cache. Typ vyrovnávací paměti, která je zařazena mezi dva subsystémy s různou rychlostí a vyrovnává tak rychlost přístupu k informacím. Urychluje přístup k často používaným datům. Typická pro CPU, na GPU se využívá hlavně pro urychlení práce s texturami.

Compute shader. Speciální případ shaderu pro programování GPGPU pomocí technologie Direct Compute.

CUDA. Hardwarová a softwarová architektura firmy NVidia, která umožňuje GPGPU programování.

Direct Compute. Technologie firmy Microsoft využívaná ke GPGPU programování. Podporována na grafických kartách s rozhraním DirectX 11. V omezené podobě je možno provozovat tuto technologii i na kartách s DirectX 10.

DirectX. Rozhraní (sada knihoven) firmy Microsoft pro přístup ke grafickému hardwaru. Nejnovější aktuální verze je 11.

Fixní pipeline. Jinak také vykreslovací řetězec grafické karty. Označuje sekvenci procesů, jejichž aplikací na vstupní data získáme obraz scény.

Fragment. Nezpracovaná jednotka obrazu. Často pojem spojován s Pixel Shadery, které pracují s jednotlivými fragmenty obrazu a jejichž výstupem jsou pixely.

GPU. Zkratka z anglického *General Procesing Unit*. Pojem je používán pro obecné označení grafického čipu, či grafické karty.

GPGPU. Zkratka z anglického *General Purpose Graphics Procesing Unit*. Tento termín označuje přístup programování grafických karet, které nejsou využity přímo pro grafické operace, ale pro obecné výpočty.

HLSL. Jazyk pro programování shaderů od firmy Microsoft.

LOD. Zkratka z anglického *Level Of Detail*. Optimalizace složitosti objektu podle vzdálenosti od pozorovatele. Vzdálenější objekty lze vykreslovat s nižší kvalitou.

OpenCL. Technologie vlastnostmi podobná řešení CUDA. Nestojí za ní ovšem konkrétní firma, je nezávislá na použité grafické kartě.

OpenGL. Rozhraní (sada knihoven) pro přístup ke grafickému hardwaru. Narozdíl od řešení DirectX není spojeno s konkrétní firmou, za jeho vývojem stojí otevřené konsorcium ARB, jehož členy je mnoho firem. Nejnovější aktuální verze je 4.

Pixel. Zkratka z anglického *Picture Element*. Nejmenší jednotka digitální rastrové grafiky.

RGBA. Označení čtyř barevných složek obrazu. R pro červenou, G pro zelenou a B pro modrou složku. A označuje alfa kanál.

Ray-tracing. Metoda vykreslování scény založená na sledování paprsku. Jednotlivé světelné paprsky se ve scéně lámou a odráží podobně jako v reálném světě.

Ray-casting. Speciální případ ray-tracingu, kdy ovšem paprsek postupuje stále přímým směrem.

Shader. Program běžící na grafické kartě ovlivňující způsob vykreslování. Syntaxe spojená s použitým grafickým rozhraním. V případě DirectX se jedná o jazyk HLSL

Texel. Z anglického *Texture Element*. Jedná se nejmenší jednotku textury. Občas v praxi zaměňováno za pojem pixel.

Textura. Tento pojem označuje souvislé uložení dat v paměti grafické karty. Lze jimi reprezentovat 2D či 3D obrázek, popř. obecná data. Datový typ může být různorodý, konkrétní povolené typy záleží na použitém grafickém rozhraní.

Voxel. Z anglického *Volumetric Element*. Obdoba texelu pro 3D textury. Jedná se o nejmenší jednotku objemu.

Vlákno. Označuje oddělený proces. Jednotlivá vlákna mohou běžet současně na více procesorech a pracovat nad stejnými daty. Množina vláken sdílí jeden paměťový prostor.

3 GPU výpočty

První grafické karty se na trhu objevily počátkem 70. let 20. století. Jednalo se, z dnešního pohledu, o primitivní zařízení, která neuměla vykreslovat 3D grafiku a pracovala převážně s čarami a kružnicemi. V pozdější fázi vývoje přibyla možnost kreslení obrázků ve 2D prostředí. Hlavním hybatelem vývoje grafických karet byl v té době převážně herní průmysl a z toho vyplývala i hlavní funkcionalita karet.

Počátkem 90. let 20. století se začaly grafické karty přeorientovávat z 2D na 3D zobrazení. Objevila se podpora trojúhelníkových sítí, texturování a jednoduché osvětlovací modely. Bohužel, veškeré operace byly pevně určeny HW grafické karty (tzv. fixní pipeline). Od roku 2000 grafické karty přinesly možnost uživatelsky upravit funkcionalitu pomocí shaderů. Jedná se o programy umožňující řídit vykreslovací operace grafické karty. Zpočátku podporovaly pouze operace s vrcholy (Vertex) a pixely (Pixel). Karty měly pevně stanovený počet shaderů pro jednotlivé typy operací. Dnešní generace karet již mají unifikované shadery, což znamená, že umožňují spouštět různé typy shaderů a nejsou pevně přiděleny jednomu konkrétnímu typu. Jedna shaderovací jednotka tak může zastoupit vrcholové, pixelové a další operace.

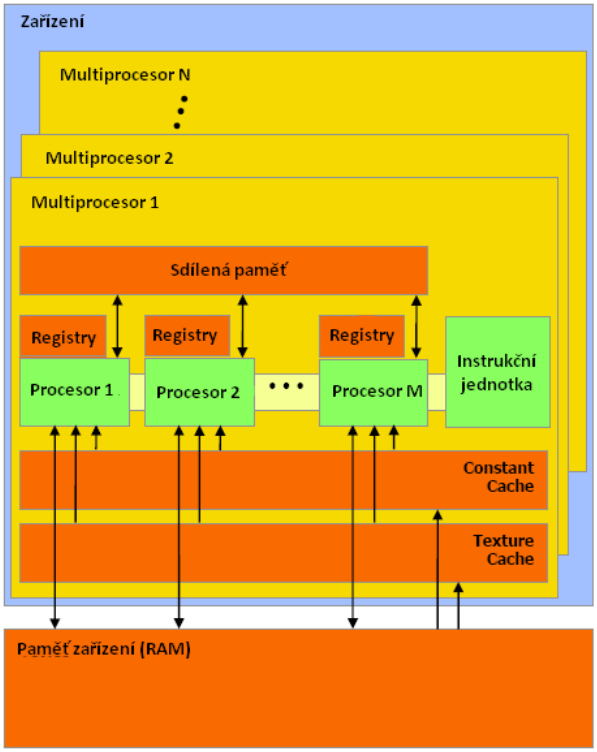
V současné době se grafické karty stávají užitečné i pro běžné výpočty, které nejsou spojeny přímo s vizualizacemi. Tuto funkcionalitu přinesly grafické karty podporující rozhraní DirectX 10 (resp. OpenGL 3). Na moderních grafických kartách můžeme počítat libovolné algoritmy, jako jsou například hledání cest v grafech, složité matematické výpočty (deformace struktur, hledání cest v grafech) atd. Celé toto nově vznikající odvětví se souhrnně nazývá GPGPU (General-Purpose computing on Graphics Processing Unit). Dnešní grafické karty jsou osazeny velkým počtem výpočetních jader a umožňují efektivní paralelizaci algoritmů. Podpora a obliba GPGPU programování stále stoupá, a proto se s aplikacemi napsanými touto cestou budeme s velkou pravděpodobností setkávat stále častěji.

Pro programování GPGPU je možno použít několik dále uvedených jazyků. Jakýmsi nepsaným standardem se stalo rozhraní OpenCL, které je spustitelné na kartách různých výrobců a není závislé na použitém vizualizačním API. Na kartách společnosti NVidia je primárně podporováno rozhraní CUDA. Oba jazyky jsou si velmi podobné a svojí syntaxí připomínají jazyk C. Třetím velkým hráčem na trhu jsou pak výpočetní shadery (Compute Shader). Ty jsou provázány s grafickým API (např. DirectX) a jejich syntaxe využívá jazyka HLSL / Cg. Funkčností a možnostmi jsou všechny rozhraní velmi podobná, programování přes shadery je ovšem obtížnější a méně vhodné pro rozsáhlé aplikace.

Koncept všech rozhraní je z logiky přístupu identický, jelikož na grafické kartě jsou HW výpočetní jednotky vyrobeny stejným způsobem. Hierarchie z hlediska softwaru se pak drobně liší v závislosti na použité knihovně. Využití rozhraní CUDA je v praxi značně diskutabilní vzhledem k omezení pouze na jednoho výrobce. Alternativa v podobě OpenCL nemá zase přílišnou podporu ze strany DirectX. Jako nejlepší řešení pro GPU výpočty s následným zobrazením s použitím knihoven DirectX se tak stávají výpočetní shadery.

3.1 Typy pamětí na GPU

Základní architektura GPU je vyobrazena na obrázku 1. Grafická karta je rozdělena na N multiprocesorů, kdy každý z nich je složen z M procesorů. Každý z těchto procesorů může spustit najednou určitý počet vláken. Jednotlivé hodnoty N, M a počtu vláken jsou samozřejmě závislé na konkrétním modelu grafické karty. Každé vlákno v rámci procesoru má přístup do několika typů paměti. Pro zápis a čtení je k dispozici paměť registrů, sdílená paměť a paměť RAM. Pouze pro čtení jsou pak určeny texturovací a konstantní cache paměti. Výhoda tohoto přístupu je snadná škálovatelnost, kdy lze snadno rozlišit mezi výkonem modelů a přitom zachovat stejný popis z hlediska softwarové vrstvy.



Obrázek 1: Architektura grafické karty

Jak již bylo zmíněno, karta je osazena různými typy pamětí:

- *hlavní (globální) RAM* paměť je nejpomalejší (udává se až 150x pomalejší než ostatní typy), ale také největší úložiště dat. Při komunikaci s hostitelem se data posílají právě přes hlavní paměť. Rychlost přenosu je limitována propustností paměti a rychlostí sběrnice. V dnešní době je typická kapacita kolem 1GB, začínají se ale pomalu objevovat modely osazené až 4GB paměti.

- *Cache konstant a textur* se využívá jako vyrovnávací paměť při komunikaci s hlavní pamětí. Je určena pouze ke čtení, nelze do ní programově zapisovat. Typické použití je při čtení textur, kdy se najednou z hlavní paměti načte více vzorků.

Předpokládá se ovšem nenáhodný přístup, v opačném případě naopak dochází k teoretickému zpomalení. Typická kapacita je kolem 16KB.

- *Sdílená paměť* v rámci multiprocesoru je určená pro čtení a zápis vláken v rámci skupiny. Typická velikost je kolem 48KB, což není velké úložiště. V určitých případech může ovšem při správném použití značně urychlit výpočet. Jedním z příkladů může být nejdříve zapisovat dočasné výsledky do této paměti a teprve finální výsledek zapíše pouze jedno vlákno do paměti hlavní.

- *Registry* jsou pak nejmenší a nejrychlejší datové úložiště. Jsou podobné klasickým registrům u procesorů, mají ovšem větší kapacitu. Na jeden multiprocesor připadá přibližně 32 tisíc 32-bitových registrů. Ty se pak dále svojí přístupností dělí mezi jednotlivá vlákna.

Občas se můžeme setkat také s pojmem tzv. *lokální paměť*. Jedná se o označení paměti v rámci jednoho vlákna pro zápis a čtení. Explicitně data ukládáme do tohoto typu paměti. Pokud je obsah dostatečně malý, je uložen v registrech. Jakmile kapacita registrů přestává stačit, data jsou automaticky ukládána do globální paměti. Pro koncového uživatele (programátora) tak není třeba řešit, kde data budou, a na úrovni zdrojového kódu pracujeme, jako bychom měli dostatečnou paměť. Samozřejmě musíme brát v potaz rychlostní omezení, pokud program začne pracovat s globální pamětí, protože se vyčerpá kapacita registrů. Většina překladačů je ovšem schopna problém detekovat již v době překladu a následně upozornit na snížení výkonu.

Práce s pamětí má jeden značně limitující prvek. Nelze pracovat s dynamickými datovými strukturami. Všechny datové struktury musí mít známé velikosti již při spuštění aplikace, není možné za běhu vytvořit např. nové pole o velikosti určené pomocí proměnné.

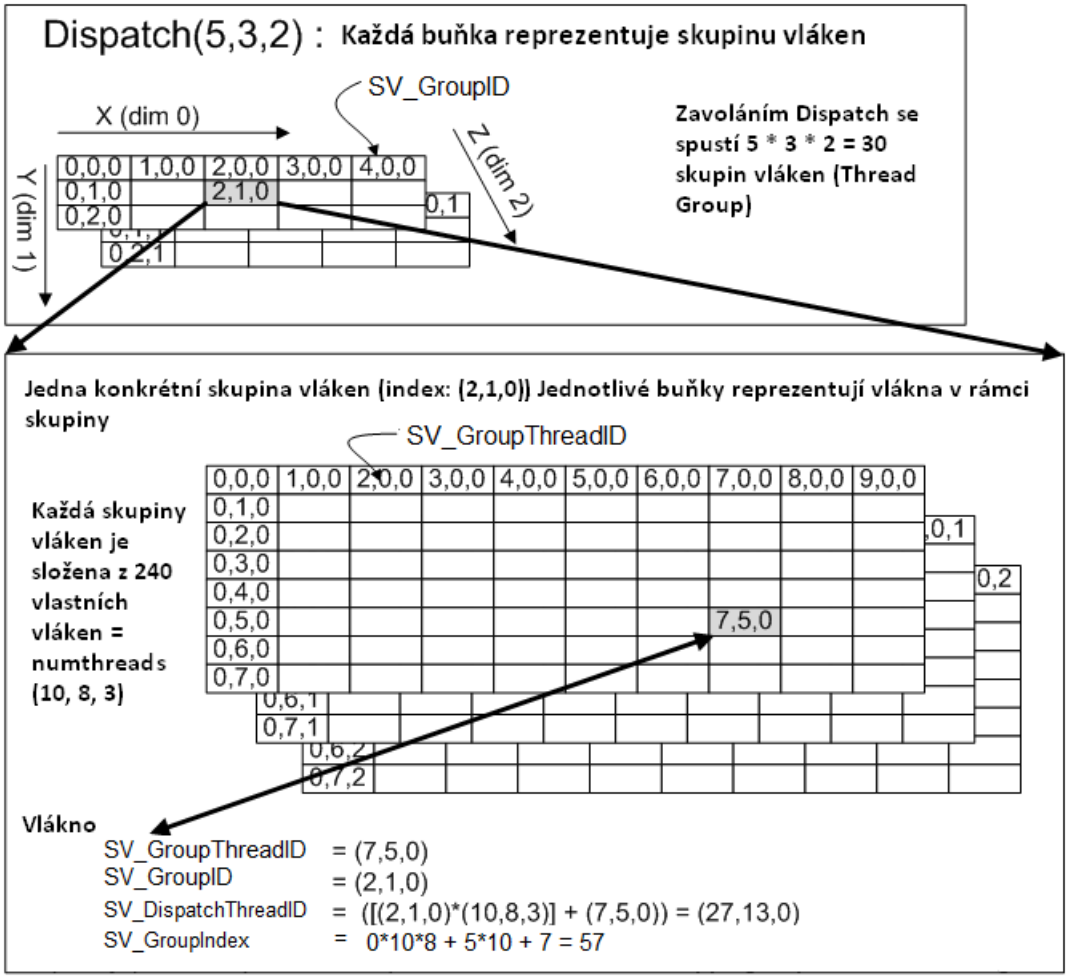
3.2 Výpočetní shadery

Výpočetní shadery představují snadné řešení pro GPGPU programování. Využívají syntaxe a funkce klasických shaderů, od nichž se strukturou prakticky neliší. Zjednodušeně by se dalo říci, že se podobají pixel shaderům s několika zásadními rozdíly. Nejsou závislé na žádné části vykreslovacího řetězce, ale mohou jeho jednotlivé kroky ovlivňovat. K jejich spuštění není třeba vykreslovat žádnou geometrii a mohou číst a zapisovat z různých míst (na rozdíl od pixel shaderů, které mají přístup k zápisu pouze jednoho konkrétního fragmentu). Umožňují také synchronizaci vláken. V další části této kapitoly se veškeré informace vztahují k řešení společnosti Microsoft a jejich výpočetním shaderům ve verzi 5. Ta je dostupná od rozhraní DirectX 11.

3.2.1 Architektura

Základní struktura je popsaná na obrázku 2. Výpočet je spuštěn zavoláním metody Dispatch, kde vstupní parametry reprezentují počet spuštěných skupin vláken. Z aplikace není možno nijak nastavit počet vláken v rámci jedné skupiny, toto musí být

nastaveno přímo v kódu vlastního shaderu jako numthreads(x, y, z). Pro většinu výpočtů je nezbytné identifikovat konkrétní vlákna a skupiny. Tyto informace jsou dostupné v konstantách shrnutých v tabulce 1.



Obrázek 2: Struktura výpočetních shaderů (zdroj: [24])

Konstanta (SV_)	Typ	Popis	Výpočet
GroupID	uint3	Udává index skupiny, do které náleží aktuální vlákno. Hodnota z intervalu [0,0,0] - Dispatch(x,y,z).	-
GroupThreadID	uint3	Index vlákna v rámci jedné skupiny. Hodnota z intervalu [0,0,0] - numthreads(x,y,z).	-
DispatchThreadID	uint3	Index v rámci všech vláken	GroupID * numthreads + GroupThreadID
GroupIndex	uint	Udává index vlákna s přihlédnutím k jeho rodičovské skupině. Index je přepočtený do jednho rozměru.	GroupThreadID.x + GroupThreadID.y * numthreads.x + GroupThreadID.z * numthreads.x * numthreads.y

Tabulka 1: Konstanty popisující rozložení vláken a skupiny

3.2.2 Datové typy

HLSL podporuje širokou skupinu datových typů. Základní jsou *skalární* a *vektorové hodnoty*. Vektory mohou mít maximálně čtyři složky. Rozšířením vektorů je *matice*, která je složená z vektorů, maximální podporovaný rozměr je 4x4. Podporovány jsou také základní vestavěné objekty, *textury* a *buffery*. Pro popis práce s texturami slouží *sampler*. Pro přenos základních dat z aplikace lze použít také konstantní bufferu, *cbuffer*. Jejich obsah je určen pouze ke čtení a je globálně viditelný.

Kromě vestavěných datových typů je možno vytvářet také uživatelské struktury pomocí klíčového slova *struct*. Podporován je také *typedef*.

Dříve zmíněné objekty (*textury* a *buffery*) jsou rozděleny na dvě základní skupiny. První z nich umožňuje pouze data číst, druhá pak číst i zapisovat. Pro snadnou orientaci mají obě skupiny stejná jména úložišť, pouze v druhém případě je předřazen prefix RW. Buffery se chovají jako klasická pole, indexovatelná přes hranaté závorky. Na rozdíl od textur nedochází k žádným interpolacím hodnot a čteme pouze hodnoty uložené v bufferu. Současná verze shaderů preferuje použití strukturovaných bufferů před klasickými. Jednotlivé typy jsou uvedeny v následujícím přehledu.

- (RW)StructuredBuffer<typ>
- (RW)Texture(1,2,3)D<typ>
- (RW)Texture(1,2,3)DArray<typ>
- (RW)Buffer<typ>
- (RW)ByteAdressBuffer

Textury jsou indexovány v rozsahu <0, 1>, v případě RGBA textur jsou získané barvy škálovány do intervalu <0, 1>. Z textur lze také získat neinterpolované hodnoty pomocí metody *Load*. Metoda nepracuje s texturovacími souřadnicemi v rozsahu <0, 1>,

ale klasicky, jakoby se jednalo o matici v rozsahu <0, velikost>. Načítáme tak konkrétní hodnoty texelů. Bohužel se nikde nepodařilo zjistit, zda tento přístup využívá texturovací cache či nikoliv.

Samozřejmostí je i podpora klasických polí, ta jsou ovšem dostupná pouze v rámci shaderu a nelze je naplnit z řídící aplikace. Nejčastější zastoupení mají pole uložená ve sdílené paměti v rámci skupiny. Označení těchto polí je uvozeno klíčovým slovem *groupshared*.

Hlavní nevýhoda je v dostupných číselných datových typech. Z celočíselných typů je totiž dostupné pouze 32-bitový celé číslo ve znaménkové nebo bezeznaménkové variaci (*int* a *uint*). Jiné celočíselné typy k dispozici nejsou. To je značný problém v některých algoritmech, kdy potřebujeme například pracovat s jednotlivými čísly o velikosti 8-bit. Problém s neexistencí jiných, než 32-bitových čísel, lze snadno vyřešit s využitím bitových operací.

Podpora datových typů s plovoucí řádovou čárkou je na vyšší úrovni. Máme k dispozici jak celý, 32-bitový *float*, tak i jeho zkrácenou 16-bitovou verzi. Pro práci se zkrácenou verzí je potřeba pracovat s touto verzí i v řídící aplikaci. Běžně jazyky tento datový typ nepodporují, rozhraní DirectX poskytuje za tímto účelem speciální datový typ *FLOAT16*. Nic nám ovšem nebrání použít implementaci třetí strany nebo vlastní. Nejnovější generace grafických karet umožňují také používat datový typ *double*, nicméně výpočty jsou pak pomalejší.

3.2.3 Programování shaderů

Výpočetní shadery jsou velmi úzce spjaty s grafickým API a není možné je provozovat bez jeho použití. K provázání vstupních a výstupních typů slouží tzv. pohledy (view). K dispozici jsou dva základní typy - *Shader Resource View* (SRV) a *Unordered Access Views* (UAV). SRV je určeno pouze k provázání zdrojů pro čtení, druhý typ umožňuje také zapisovat výsledky. UAV i SRV zdroje jsou přístupné také z Pixel Shaderu. V ukázce algoritmu 1 je znázorněn postup vytváření bufferu a jeho provázání s shaderem. Před vlastním spuštěním shaderu se nastaví příslušné zdroje. Po ukončení shaderu je vhodné tyto zdroje opět uvolnit, jinak může v případě další práce s nimi docházet k nespecifikovaným chybám.

Algoritmus 1 Vytvoření bufferu a provázání se shaderem

```
// Vytvoření strukturovaného bufferu
ID3D11Buffer *pStructuredBuffer;
D3D11_BUFFER_DESC sbDesc;
sbDesc.BindFlags = D3D11_BIND_UNORDERED_ACCESS | D3D11_BIND_SHADER_RESOURCE;
sbDesc.CPUAccessFlags = 0;
sbDesc.MiscFlags = D3D11_RESOURCE_MISC_BUFFER_STRUCTURED;
sbDesc.StructureByteStride = sizeof(uint);
sbDesc.ByteWidth = sizeof(uint) * dataSize * 2;
sbDesc.Usage = D3D11_USAGE_DEFAULT;
pd3dDevice->CreateBuffer(&sbDesc, 0, &pStructuredBuffer);

//Naplnění daty
deviceContext->UpdateSubresource( pStructuredBuffer, 0, NULL, data, 0, 0 );

// Vytvoření UAV
ID3D11UnorderedAccessView *pStructuredBufferUAV;
D3D11_UNORDERED_ACCESS_VIEW_DESC sbUAVDesc;
sbUAVDesc.Buffer.FirstElement = 0;
sbUAVDesc.Buffer.Flags = 0;
sbUAVDesc.Buffer.NumElements = dataSize * 2;
sbUAVDesc.Format = DXGI_FORMAT_UNKNOWN;
sbUAVDesc.ViewDimension = D3D11_UAV_DIMENSION_BUFFER;
pd3dDevice->CreateUnorderedAccessView(pStructuredBuffer,
        &sbUAVDesc, &pStructuredBufferUAV);
```

V algoritmu 2 je malá ukázka vlastního výpočetního shaderu. Samotný shader není v praxi nikterak funkční a slouží pouze pro demonstraci kódu. Z aplikace jsou data uložena do strukturovaného bufferu Compressed (určeného pouze pro čtení), z kterého se provádí během dekomprese čtení. Dekomprimované výsledky se ukládají do strukturovaného bufferu určeného pro zápis (ale i čtení). Zde je vidět již dříve zmíněné použití prefixu RW. Jako pomocná paměť je použita právě sdílená paměť uvozena klíčovým slovem *groupshared*. Po zápisu jednotlivých vláken do této paměti je využit synchronizační krok, kdy se všechna vlákna musejí setkat na bariéře.

Algoritmus 2 Ukázka kódu výpočetního shaderu

```
struct CompressType
{
        uint data;
        uint offset;
};

cbuffer Settings  : register(b2)
{
        uint compressedSize;
        uint decompressedSize;
};

StructuredBuffer<CompressType> Compressed : register(t0);
RWStructuredBuffer<uint> Decompressed : register(u0);
groupshared uint searchBuffer[SEARCH_BUFFER_SIZE];

[numthreads(10, 1, 1)]
void Simple( uint3 DTid : SV_DispatchThreadID, uint3 groupID : SV_GroupID )
{
        for (int i = 0; i < SEARCH_BUFFER_SIZE; i++)  { searchBuffer[i] = 0; }
        GroupMemoryBarrierWithGroupSync();

        RunDecompression();
}

void RunDecompression()
{
        [allow_uav_condition]
        for (uint i = 0; i < compressedSize; i++)
        {
                if (Compressed[i].offset > i) break;
                //decompress
        }
        Decompressed[index] = decompressedValue;
}
```

Před cyklem je v kódu v hranatých závorkách uvedena podmínka pro kompilaci kódu. Ta říká kompilátoru, jakým způsobem má vyhodnotit a optimalizovat cyklus. V tomto případě, použitá *allow_ uav_ condition* znamená, že cyklus může být předčasně ukončen na základě předem neznámých dat. S tímto přepínačem nesmí cyklus obsahovat žádné synchronizační bloky. Více informací lze nalézt na oficiálních stránkách firmy Microsoft [24].

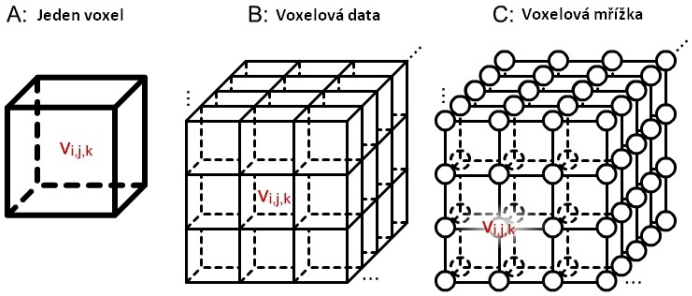
3.2.4 Ladění

Ladění kódu shaderu je problematické. Paralelní aplikace se obtížně ladí i pokud je spouštíme pouze na CPU. Běh na GPU přidává další problém v nemožnosti krokovat kód a v případě chyby není problém zablokovat grafický ovladač, což ve většině případů vede k nutnosti restartu systému. Microsoft, jako hlavní vývojář rozhraní DirectX11, poskytuje možnost ladění v podobě programu PIX, ovšem jeho funkcionalita je sporná. Ve většině případů program nefunguje tak, jak by měl. Lepší řešení poskytují firmy NVidia a AMD. Jejich řešení má hlavní nevýhodu v potřebě vlastnit dvě grafické karty. Na první se spouští vlastní shader, zatímco druhá slouží jako grafická karta systému a umožňuje nám prohlížet výsledky z první.

4 Volumetrická data

Volumetrická data můžeme reprezentovat několika způsoby. Dřívější grafické karty podporovaly pouze pole 2D textur, postupem času byla přidána podpora pro 3D textury. S dnešní generací karet je možné pracovat s daty stejným způsobem jako na CPU, tudíž mít data uložena v klasickém jedno nebo vícerozměrném poli. Kromě surových (raw) dat je možnost využívat také různé struktury, jako jsou například stromy nebo tabulky. Zde je ovšem třeba brát v úvahu rozdílnou architekturu CPU a GPU a s tím spojené změny v efektivitě algoritmů.

Nejjednodušší reprezentaci tvoří surová, neboli tzv. raw data. V této podobě můžeme na data pohlížet jako na trojrozměrné pole, kde hodnota v každé buňce odpovídá hodnotě voxelu (viz. obrázek 3, B). Analogický případ z 2D světa by byl, kdybychom měli klasický obrázek, kde hodnota v každém bodě tvoří vlastní hodnotu pixelu. Na voxelová data se ovšem můžeme dívat i jako na mřížku (viz. obrázek 3, C), kde hodnoty jsou v jejích uzlech. Tento přístup se využívá u metod extrakce iso-ploch, jako je například Marching Cubes (viz. dále).



Obrázek 3: Raw voxelová data

Raw datová reprezentace je velmi jednoduchá, přináší s sebou ovšem problém s velikostí dat. Ta závisí čistě na rozlišení a bitové velikosti jednotlivých voxelů. Výhodou je pak na druhou stranu rychlost zpracování, kdy nemusíme uvažovat žádné dekomprese ani komprese dat a můžeme pracovat s daty prakticky okamžitě.

Problém s velikostí dat je značně omezující, pokud se týká vizualizace dat s velkým rozlišením nebo časově proměnných dat. Čím větší rozlišení, tím větší paměťové nároky. Mohli bychom data na grafickou kartu průběžně nahrávat během vykreslování, rychlost přenosu dat má ale za následek značné zpomalení vykreslování. Z těchto důvodů je třeba použít určitý způsob komprese dat. Zde můžeme pracovat se ztrátovou nebo bezeztrátovou kompresí. V případě obou typů kompresí ovšem musíme zajistit náhodný přístup k datům, nemělo by význam data nejdříve rozbalit a tím pádem opět získat původní velikost. Dostupné metody pracují s obměnami základních algoritmů, které jsou různým způsobem kombinovány. Konkrétní příklady jsou uvedeny v kapitolách 7 a 8.

5 Komprese dat

Obecný popis dané problematiky uvádí např. Salomon [31]. Komprese dat je proces, při kterém vstupní data konvertujeme na výstupní o menší velikosti. Existuje mnoho kompresních metod jejichž vhodnost je vázána na určitý typ dat. Všechny metody mají ovšem společný základ - odstranění redundance[1] v datech a s tím spojené zvýšení entropie. Entropie (viz vztah 1) udává střední hodnotu informace jednoho kódového znaku. Záleží tak na statistickém rozložení zdroje. Sekvence náhodných čísel má maximální míru entropie. Snižujeme-li náhodnost dat, klesá i entropie. Pomocí entropií lze vyjádřit také redundanci dat, a to jako rozdíl maximální entropie dat a aktuální entropií.

$$H = -\sum_{i=1}^{n} P_i log_2 P_i \qquad P_i = pravděpodobnost\ výskytu\ symbolu \tag{1}$$

Obecným příkladem redundancí dat mohou být anglické texty. Ty obsahují velmi často písmeno E, zatímco Z se objevuje velmi zřídka. Tento jev je označován jako *abecední redundance*. Na základě toho můžeme písmeno E reprezentovat kratším kódem, naopak Z kódem delším. Druhým typem je *kontextová redundance*. Ta vychází z navzájem propojených skupin. Například. písmeno A je často následováno písmenem U. Podobná pozorování jako u textů lze nalézt i u obrazových dat. Jedná se o určité barvy pixelů nebo sousednost v rámci okolí pixelu.

U komprimovaných souborů je většinou redundance malá, proto je již nelze dále dobře komprimovat. Kdybychom byli schopni komprimovat již jednou komprimované soubory stále dokola, skončili bychom po určitém počtu kroků s velikostí 1 bit. To samozřejmě není možné, jelikož 1 bit nemůže obsáhnout informace z celého obecného datového souboru.

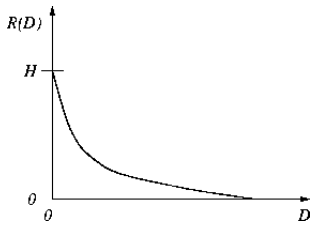
Možnosti komprese silně závisí na datovém souboru. Většina souborů nelze komprimovat prakticky žádným způsobem, protože obsahují data, která jsou již nějakým způsobem komprimována. Jedná se obsahově náhodné soubory a v takovýchto souborech se jen obtížně hledá velká redundance dat.

Komprimujeme-li data, můžeme pracovat s dvěma různými skupinami algoritmů - ztrátovými a bezeztrátovými. Rozdíl mezi metodami je značný. Pokud komprimujeme a dekomprimujeme soubor bezeztrátově, dostaneme vždy originální data. U ztrátové komprese se nám již nikdy nepodaří obnovit originální soubor. Každá ze skupin, již podle svého názvu, je určena pro jiný typ dat. Zatímco pro textové a binární soubory je jediná možnost bezeztrátová komprese. Pro data, kde je jejich primární cíl vizuální vjem (obrazy a zvuková data) lze využít i ztrátové kompresní algoritmy. Kompresní algoritmy musí být navrženy v tomto případě tak, aby odstraněná data příliš nepoškodila vizuální dojem dat.

U bezeztrátových algoritmů je kompresní limit omezen entropií dat. Komprimovaná data nelze zhustit více, než dovoluje hodnota míry jejich entropie. Díky tomu nelze data opakovaně komprimovat a dosahovat stále lepších výsledků. Podobné pravidlo lze uplatnit i pro ztrátové komprese. Pro zdroj, u kterého je známa statistická povaha a

[1] Nadbytečnost dat; data, která jsou v datovém souboru duplicitní, nepotřebná nebo zbytečně rozsáhlá

pro danou míru zkreslení D, je možné nalézt funkci R(D). Tato funkce se označuje jako Rate-Distortion Function a její průběh je vyobrazen v grafu na obrázku 4. Tato funkce udává pro přípustnou míru zkreslení D nejlepší dosažitelný kompresní poměr R(D). Podrobnější informace jsou uvedeny v [1].



Obrázek 4: Rate-Distortion Function

Většina kompresních algoritmů pracuje lineárně od začátku do konce souboru a navíc v sériovém módu. Některé, hlavně ztrátové algoritmy, umožňují pracovat s nezávislými bloky dat samostatně.

Důležité je mít možnost nějakým způsobem porovnávat algoritmy a určovat jejich kvalitu a vhodnost pro daný typ dat. Jedním z možných měřítek je určení kompresního poměru. Vztah pro jeho výpočet je uvedena ve vztahu 2. Vypočtená hodnota 0.6 znamená, že komprimovaná data zabírají 60% prostoru původních dat (neboli velikostní úspora je 40%). Hodnoty větší než jedna znamenají, že komprese neměla požadovaný efekt a velikost dat narostla.

$$Kompresní\ poměr = \frac{velikost\ výstupu}{velikost\ vstupu} \tag{2}$$

V některých publikacích můžeme také nalézt hodnotu *bpb* (bit-per-bit), která udává kolik bitů komprimovaného souboru odpovídá bitům původního souboru. Toto značení je ve velké míře využíváno u obrazových dat, kde značí počet bitů na pixel obrazu.

Opakem kompresního poměru je tzv. kompresní faktor (viz. vztah 3). Hodnoty větší než 1 určují kompresi. Čím větší hodnota, tím větší komprese.

$$Kompresní\ faktor = \frac{velikost\ vstupu}{velikost\ výstupu} \tag{3}$$

Pro ztrátové komprese je důležitým faktorem měření chyby oproti originálním datům. Existuje více druhů testů, ve většině literatury se ale setkáváme s metodami PSNR a MSE. MSE (z anglického *mean squared error*, česky střední kvadratická chyba) je definována pro dva černobílé obrazy I, K (originální a komprimovaný) o rozměrech m x n jako

$$MSE = \frac{1}{m \cdot n} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} ||I(i,j) - K(i,j)||^2 \tag{4}$$

PSNR (z anglického *peak signal-to-noise ratio*, česky špičkový poměr signálu k šumu) vyjadřuje poměr mezi maximální možnou energií signálu a energií šumu. Většinou bývá vyjádřeno pomocí logaritmického měřítka. Vztah pro výpočet PSNR je uvedený ve

vztahu 5. Hodnota MAX_Ije maximální možná hodnota pixelu v obrázku (pro šedotónový obraz je to 8bit = 255). Pro barevné obrazy se jedná o sumu přes všechny složky dělenou počtem složek. Pro data v plovoucí řádové čárce se tato hodnota volí jako 1.0.

$$PSNR = 10 \cdot log(\frac{MAX_I^2}{MSE}) \tag{5}$$

Obvyklé výsledky PSNR testu se pohybují mezi 20 a 40 dB. Vyšší hodnoty znamenají lepší kvalitu rekonstruovaného obrazu. Vzhledem k tomu, že jde o logaritmickou funkci, a vzhledem k charakteru různých zdrojových obrazů, není možné porovnávat výsledky jednotlivých testů mezi sebou. Relevantní porovnatelné hodnoty získáme pouze pro porovnávání různých typů komprese proti stejnému výchozímu souboru.

5.1 Bezeztrátová komprese

Bezeztrátové komprese můžeme rozdělit na dvě skupiny s ohledem na použitá objemová data. První z nich umožňuje vykreslování zároveň s dekompresí, protože komprimovaná data umožňují náhodný přístup. Druhým zástupcem jsou pak klasické datové kompresní algoritmy, kde je potřeba nejdříve data dekomprimovat a pak je možno teprve provést vlastní vykreslování. Mezi tyto algoritmy se řadí např. Huffmanovo kódování, Lempel-Ziv algoritmy atd. Podrobnější informace o různých dalších algoritmech lze nalézt v [31].

Existuje široká paleta kompresních algoritmů a jejich kombinací. Neexistuje žádný univerzální algoritmus, který by za všech okolností produkoval nejlepší kompresní poměr. Vždy je třeba brát v úvahu povahu dat, rychlostní aspekty komprese / dekomprese, ale třeba i rozšířenost formátu v případě využití pro dlouhodobé archivace.

5.1.1 Vynechávání prázdného prostoru

Nejjednodušší kompresí je eliminace prázdného prostoru a velkých stejných bloků. Celý objem rozdělíme na menší bloky. Přístup je podobný konstrukci oktalového stromu (octree). Jednotlivé části potom uložíme jako samostatná data spolu s jejich skutečnou velikostí. Prázdné bloky (nebo bloky obsahující stejnou hodnotu) se nám tak zmenší na blok o velikosti 1. Kromě vlastní komprese dat se zrychlí i vlastní vizualizace, kdy lze snadno přeskakovat prázdné bloky. Celý přístup vzdáleně připomíná metodu RLE (Run-Length Encoding) [31].

5.1.2 Oktalové stromy

Oktalové stromy jsou metodou komprese umožňující náhodný přístup k datům. Metoda je obecně označena jako Sparse Voxel Octree (SVO) [35]. Data jsou rozdělena do jednotlivých bloků, které odpovídají listům stromu. Pokud všech osm listů nese stejnou hodnotu, listy se sloučí a nahradí jejich rodičovský uzel, ze kterého se tím pádem stane list. Takto postupuje celá konstrukce stromu. Ve finálním kroku se snadno určí vnitřní oblasti, které nemohou být viditelné a tyto oblasti lze úplné vynechat. Následné vykreslování je pak urychleno vynecháváním prázdných oblastí a zároveň můžeme měnit kvalitu podle vzdálenosti oblasti od pozorovatele. Tento tzv. Level Of Detail (LOD)

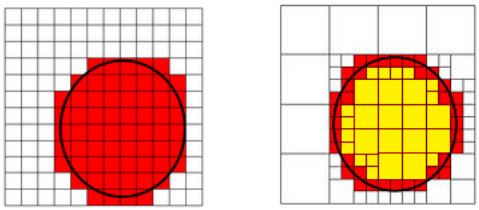
je velmi vhodný pro velká data, kdy data dále od kamery není potřeba vykreslovat s vysokou kvalitou. Na této metodě částečně staví projekt GigaVoxels [5].

Hlavní nevýhoda metody spočívá v jejím zaměření. Metoda není primárně určena pro vizualizaci průhledných dat, ale pouze pro povrchový popis objektů složených z voxelů. Je tak tím pádem na úrovni extrakce iso-ploch a nezobrazí vnitřní strukturu. Jistě, není problém vnitřní strukturu do dat zahrnout. Během konstrukce stromu nebudeme vynechávat vnitřní oblasti. Pak se ovšem z komprese stane efekt přesně opačný a data naopak svojí velikost zvětší. To je dáno potřebou uložit stromovou strukturu, kdy vnitřní uzly zabírají většinu prostoru. Stromová struktura lze v tomto případě uložit v komprimované podobě, což ovšem neřeší problém vzniklý s následnou vizualizací.

Během vizualizace je potřeba procházet stromem. Vzhledem k velikosti stromu je potřeba tento strom uložit v globální paměti grafické karty. Algoritmy pro průchody stromy jsou rekurzivní, popř. je lze přepsat na iterační verze s využitím zásobníku navštívených uzlů. Zde vzniká problém s využitím těchto algoritmů na GPU, kde žádný zásobník není k dispozici. Můžeme si naprogramovat vlastní implementaci pomocí polí, ovšem ta se ve většině případů nepodaří uložit do sdílené paměti a jejich odložení do globální paměti je pro algoritmus z výkonnostního hlediska nevýhodné. Z tohoto důvodu nebyla algoritmům postaveným na stromových strukturách věnována větší pozornost.

Na obrázku 5 je ukázán příklad reprezentace raw dat pomocí stromu. Pro názornost je využito pouze 2D verze stromu (tzv. quadtree). Na pravé části obrázku jsou vidět původní, nekomprimovaná data. Každá buňka představuje jednu hodnotu. V případě červených buněk se jedná o vlastní data, bílé buňky značí prázdný prostor.

Vlevo potom vidíme data komprimovaná pomocí stromové struktury. V případě komprese a uvažování pouze obalových dat (iso-povrchů) můžeme vše, co je vyobrazeno žlutou barvou vynechat z finální reprezentace. Datová úspora je vidět na uložení prázdných oblastí, které se značně zredukovaly. V případě, že chceme zachovat vnitřní strukturu dat, uložíme do výsledné reprezentace i strukturu vyobrazenou žlutou barvou.



Obrázek 5: Strom vytvořený z raw dat

5.1.3 Algoritmy z rodiny Lempel-Ziv

Jedná se o slovníkové algoritmy [31], [25] s nesymetrickou dobou komprese a dekomprese. Dekomprese je ve většině případů značně rychlejší, než vlastní komprese. Základní verze algoritmu pochází z roku 1977 a je označována jako LZ77 (Lempel–Ziv–rok 1977). V pozdějších letech byla základní verze různým způsobem pozměněna a zvyšoval se tak kompresní poměr. Dekomprese u těchto algoritmů je rychlá, ovšem předpokládá značné

využití paměti a sériový přístup. Z tohoto důvodu není příliš vhodné tyto metody implementovat na grafických kartách.

Nejjednodušší verzí je právě původní verze z roku 1977. Algoritmus pracuje se dvěma částmi textu zároveň. První představuje aktuální okno - to, které komprimujeme, a druhé představuje posuvné okno - tzv. sliding window. V posuvném okně se snažíme nalézt co nejdelší řetězec obsažený v aktuálním okně, na který posléze v aktuálním okně vytvoříme odkaz. Komprese je účinná, pokud se v datech často opakují stejné shluky, v opačném případě může naopak dojít k nárůstu velikosti dat. Kompresní poměr je ovlivněn velikostí posuvného okna - musí být dostatečně velké, abychom mohli najít požadovaný řetězec a zároveň dostatečně malé, aby komprimace netrvala příliš dlouho.

Základní verze algoritmu LZ77 má velkou nevýhodu v komprimaci jednotlivých znaků. Každé kódové slovo má totiž stejnou strukturu: [pozice, délka, následující znak]. Pozice udává, o kolik znaků zpět se musíme posunout, délka pak udává kolik následujících znaků od této pozice zkopírujeme na konec výstupu. Znak je pak následující znak. Ukázka je uvedena na textu abracadabra.

```
Vstup: abracadabra
Výstup: [0,0,a][0,0,b][0,0,r][3,1,c][2,1,a][7,3,a]
```

Z uvedeného příkladu je jednoznačně vidět nevýhoda. Pokud vstupní data obsahují prakticky pouze unikátní znaky, komprese data značně zvětší. Tento problém částečně odbourává alternativní verze algoritmu, označovaná jako LZSS (Lempel–Ziv Storer–Szymanski). V následující ukázce je řetězec před a po kompresi pomocí této modifikované metody.

```
Vstup: abracadabra
Výstup: abracad[7,4]
```

V tomto případě je potřeba pro každé kódové slovo navíc jeden bit, který určí, zda následující kód reprezentuje přímo znak nebo kódované slovo. V případě, že bychom v textu nedokázali nalézt žádné shody a vstup by byl totožný s výstupem, zvětšila by se velikost výstupních dat o <velikost dat> bitů. Výhodou algoritmu LZSS je snadná implementace, pokud již máme hotový LZ77. Pouze přidáme do kódu podmínky pro nekomprimování krátkých sekvencí. Tento algoritmus je například součástí formátu RAR. V literatuře se často setkáváme s chybným označením algoritmů. Verze LZSS se někdy označuje jako LZ77, což způsobuje nejednoznačnost výkladu algoritmu.

V roce 1978 byl vytvořen následovník, LZ78. Do slovníku se postupně ukládají unikátní nové fráze. Každý krok LZ78 pošle na výstup dvojici (i,a), kde i je index fráze do slovníku a a je znak následující bezprostředně za nalezenou frází. Slovník je reprezentován jako strom s očíslovanými uzly. Jdeme-li z kořene stromu do daného uzlu, dostaneme frázi ze vstupního textu. Zásadní problém je velikost slovníku, kdy může zaplnit celou volnou paměť. Možným řešení, je např. smazání slovníku a tvorba nového.

Dalším rozšířením metody LZ78 vznikl algoritmus označovaný jako LZW. Ten můžeme najít například v obrázcích TIFF, GIF, formátu PDF, je základem pro ZIP apod. Algoritmus postupně rozpoznává a ukládá do tabulky řetězce znaků a tyto řetězce nahrazuje ve výstupním textu přirozenými čísly z předem definovaného intervalu. Tento

interval určuje výslednou abecedu a platí pro něj, že například při kódování řetězce znaků zobrazených v osmibitovém zobrazení je prvních 255 čísel vyhrazeno pro zobrazení samostatných znaků z původní abecedy. Čísla nad 255 se pak přidělují jednotlivým nalezeným řetězcům. Přitom se vytváří tabulka (slovník) již rozeznaných řetězců. Dekompresní algoritmus potřebuje pouze komprimovaná data, slovník si dokáže postavit znovu během dekomprese.

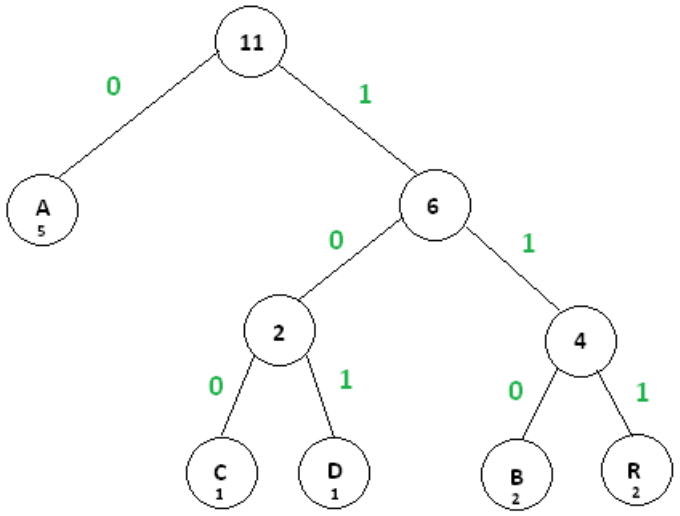
Algoritmy z rodiny LZ jsou jedny z nejpoužívanějších. Jejich kompresní poměr se v praxi velmi často vylepšuje dalšími kompresními algoritmy. Nejrozšířenější je využití Huffmanova kódování. S touto kombinací se můžeme setkat prakticky ve všech známých formátech, počínaje nejstarším ZIP, přes RAR až po nejnovější 7-Zip.

5.1.4 Huffmanovo kódování

Tento typ bezeztrátové komprese pracuje s četnostmi jednotlivých znaků v souboru. Podle jejich četnosti jsou pak jednotlivé znaky kódovány. Znaky s největší četností jsou kódovány s použitím méně bitů než znaky vyskytující se v souboru ojediněle. Tyto ojedinělé znaky mohou pak zabírat i více než 8bitů, naopak frekventované znaky lze uložit s použitím méně než 8bitů.

Vlastní komprese se skládá ze dvou kroků. V prvním kroku vytvoříme tabulku četností, z níž následně postavíme binární strom. Tento strom nemá jednoznačnou interpretaci, nicméně v praxi se dodržují určitá pravidla pro jeho stavbu. Jednotlivé znaky jsou pak kódovány podle cesty, po níž se k nim stromem dostaneme. V druhé fázi procházíme jednotlivé znaky vstupních dat a nahrazujeme je kódovými slovy dle stromu. Při dekompresi musíme mít k dispozici tuto tabulku kódových slov, což je nevýhoda algoritmu.

Ukázku kódovacího stromu lze vysvětlit na našem oblíbeném vstupním řetězci "ABRACADABRA ". Vytvoření kódových slov z binárního Huffmanova stromu je uvedeno na obrázku 6.



Obrázek 6: Huffmanův strom pro slovo "abracadabra". Četnosti jednotlivých písmen jsou uvedeny v uzlu s daným písmenem. Finální kódová slova jsou A = 0, B = 110, C = 100, D = 101, R = 111

S použitím stromu resp. získaných kódových slov z obrázku 6 zakódujeme vstupní řetězec, jak je uvedeno na následující ukázce (svislá lomítka pro větší přehlednost oddělují jednotlivé kódované znaky). Na první pohled získáme nejlepší kompresi z dosud ukázaných algoritmů. Musíme si ovšem uvědomit, že spolu s takto zakódovaným textem je potřeba distribuovat také kódovací strom. Tato skutečnost data zvětší a degraduje v tomto konkrétním případě kompresní převahu Huffmanova kódování.

Vstup: ABRACADABRA
Výstup: 0|110|111|0|100|0|101|0|110|111|0

Algoritmus se často využívá jako doplněk pro slovníkové komprese, můžeme se s ním ale také setkat ve ztrátových kompresích. Mezi ně patří např. JPEG, Ogg/Vorbis, MP3 a další.

5.1.5 Aritmetické kódování

Další kompresní metoda založená na statistice dat, podobně jako Huffmanovo kódování. Základní myšlenka metody je komprimovat celá vstupní data do jediného reálného čísla z intervalu $<0, 1)$ (použitá notace $<a, b/$ znamená, že znak "a" do intervalu patří, zatímco znak "b" již do tohoto intervalu nepatří). Jednotlivým symbolům je na základě pravděpodobnosti výskytu v datové množině přiřazena odpovídající část intervalu $<0, 1)$. Při kódování je pak celý interval postupně omezován. Pro každý příchozí symbol se vybere z aktuálního intervalu odpovídající část a ta se stane novým základním intervalem. Takto postupně omezujeme interval, dokud máme k dispozici vstupní znaky. Celou zprávu pak reprezentuje libovolné číslo z výsledného intervalu.

Pro lepší pochopení problematiky je dobré uvést konkrétní příklad, převzatý z [31]. Mějme tři vstupní znaky s pravděpodobností zastoupení v textu, jak je uvedeno v

tabulce 2. Vstupní interval <0, 1) je na základě těchto znaků rozdělen na tři části, <0, 0.4), <0.4, 0.9) a <0.9, 1.0), kterým jsou přiděleny jednotlivé symboly.

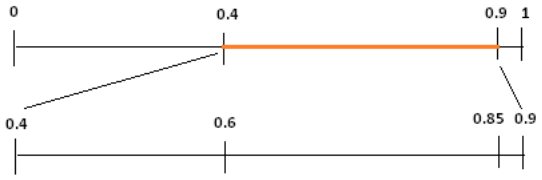
Symbol	Pravděpodobnost
A	0.4
B	0.5
C	0.1

Tabulka 2: Vstupní pravděpodobnosti aritmetického kódování

Pro zakódování řetězce BBBC je posloupnost kroků následující:

- Počáteční interval <0, 1)
- Načteme symbol B - redukce intervalu na <0.4, 0.9) - viz. obrázek 7
- Načteme symbol B - redukce intervalu na <0.6, 0.85)
- Načteme symbol B - redukce intervalu na <0.7, 0.825)
- Načteme symbol C - redukce intervalu na <0.8125, 0.8250)
- Finální kód popisující vstup je libovolné číslo z intervalu <0.8125, 0.8250)

Každý z podintervalů má jasně vymezené hranice, které lze spočítat vždy z předchozího intervalu.



Obrázek 7: Jeden krok aritmetického kódování

Problém s využitím roszahu intervalu <0, 1) je omezení plynoucí z aritmetiky v plovoucí řádové čárce. Tento problém lze vyřešit změnou hranic intervalu. Místo původní reprezentace se často využívá interval <0, 65535). Není problém využít i jiný, uživatelsky definovaný rozsah podle aktuálního problému. Tím, že pracujeme s celými čísly namísto plovoucí řádové čárky, můžeme upravovat intervaly pomocí logických operací.

Pro správnou funkčnost algoritmu je nutné také vyřešit problém s "blízkými" čísly. V plovoucí řádové čárce se jedná např. o hodnoty intervalu 0.2997 a 0.3001. V celočíselné reprezentaci získáme hodnoty 2997 a 3001. Tato situace se označuje jako *podtečení*. Identifikovat lze v celočíselné aritmetice snadno, nerovnají se nejvýznamnější číslice (tzv. MSB, z anglického *Most Significant Bit*) obou hodnot. Tento problém je řešen pomocí bitových posunů obou číslic a zapisování posunutých hodnot na výstup.

Algoritmus se často využívá jako doplněk pro slovníkové komprese, můžeme se s ním ale také setkat ve ztrátových kompresích.

5.1.6 Shrnutí

Výše uvedený výběr algoritmů se může na první pohled zdát nelogický. Většina algoritmů pochází již ze 70. let a mohlo by se zdát, že mají svojí největší slávu za sebou. V určitém ohledu je to možné, vzhledem ke značné obecné konzervativnosti při přechodu na nové formáty. Na druhou stranu, tyto algoritmy jsou součástí mnoha jiných a zároveň jsou velmi jednoduché a rychlé.

V dnešní době přinášejí větší kompresní poměry, v řádu jednotek až desítek procent, kompresní algoritmy z rodiny Context Mixing využité v komprimačních programech z rodiny PAQ . Tyto algoritmy jsou volně dostupné, ale vzhledem k velkému množství revizí prakticky zatím nepoužitelné. Použitelnou a kvalitní implementaci lze najít v [21]. Obecně je problémem těchto algoritmů nízká rychlost komprese a relativní složitost kódu.

Lze nalézt i algoritmy s většími kompresními poměry. Oproti zaběhnutým slovníkovým LZ metodám přinášejí lepší kompresní poměry v jednotkách až desítkách procent. Většina z nich dostupná v placeném komprimačním programu WinRK. Bohužel, tyto algoritmy jsou licencované a uzavřené.

Pro potřeby této práce je rozhodující jednoduchost algoritmu, zejména jeho dekompresní části. Důvod tohoto přístupu je dále uveden v kapitole 8.

5.2 Ztrátová komprese

Až do této chvíle jsme popisovali bezeztrátovou reprezentaci a kompresi dat. Použijeme-li ztrátové komprese, jsme schopni dosáhnout lepších kompresních poměrů za cenu ztráty informace. Využít lze klasické algoritmy známé z kompresí obrázků (volumetrická data jsou vlastně 3D obrázky). Často je komprese omezena podmínkou na dekompresi, která by měla umožňovat náhodný přístup do obrazu bez nutnosti dekomprese celých dat. To omezuje použití některých algoritmů, jako jsou například klasická JPEG komprese.

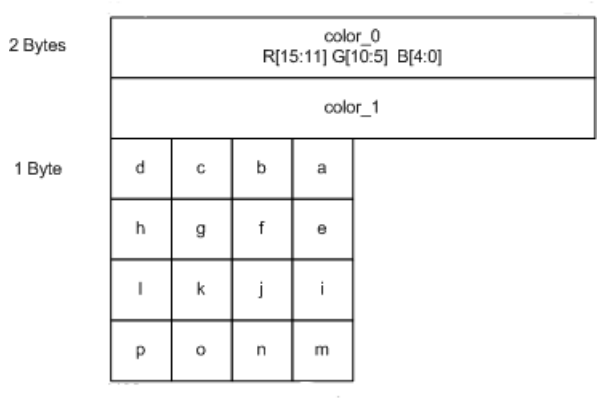
5.2.1 DXT komprese

Častým kompresním algoritmem, využívaným hlavně pro textury, jsou metody DXTn [23]. V různých zdrojích lze tyto metody nalézt také pod jmény S3TC (S3 Texture Compression) nebo BC (Block Compression). Jedná se o metody s konstantním poměrem komprese, kdy jsou bloky o určité velikosti z původních dat uloženy pomocí menšího bloku. Často se jedná o blok 4x4 zmenšený na 1x1.

Nejjednodušší verzí je DXT1 (BC1) pracující s bloky 4x4. Namísto uchovávání 16 hodnot (barev) se uloží pouze dvě referenční barvy (jako 16-bitové hodnoty) a k nim šestnáct 2-bitových barevných indexů. Tyto indexy určují jednu ze čtyř možných barev pro každý texel. První dvě barvy jsou uložené referenční hodnoty, zbylé dvě barvy jsou pak interpolace retenčních. V jednom případě je větší váha na první, v následujícím na druhé barvě.

```
color_2 = 2/3 * color_0 + 1/3 * color_1
color_3 = 1/3 * color_0 + 2/3 * color_1
```

Jedná se základní algoritmus, který nepodporuje průhlednost uloženou v alfa kanálu. Pro lepší představu je vizualizace na obrázku 8. Hodnoty a - p jsou zmíněné indexy v rámci bloku 4x4.



Obrázek 8: Komprese DXT1

Další verze tohoto algoritmu jsou pak ve své podstatě pouze pozměněnou verzí základního přístupu, kdy je přidána podpora průhlednosti, popř. zvětšením indexů možnost dosáhnout lepší kvality komprese za cenu horšího kompresního poměru.

Algoritmy lze použít pro komprese volumetrických dat, ovšem výsledky nejsou příliš uspokojivé. Relativně malý kompresní poměr spojený s velkou ztrátou kvality není ve většině případů dostatečný.

5.2.2 Vektorová kvantizace

Na základě velké ztráty kvality u DXT algoritmů přišel Schneider s algoritmem L3VQ pracujícím s vektorovou kvantizací [12] a [32]. Vektorová kvantizace pracuje s kódovou množinou vektorů. Tato množina je vytvořena ze zdrojových dat, která jsou rozdělena na bloky určité velikosti. Každý blok je v komprimovaných datech nahrazen odkazem do kódové tabulky. Kvalita rekonstruovaných dat záleží na velikosti kódové tabulky. Výhoda je velmi snadná dekomprese, která lze počítat v reálném čase.

Podobně jako DXT algoritmy, i zde se pracuje s pevným kompresním poměrem, který závisí pouze na rozlišení vstupních dat. Z tohoto algoritmu bylo v průběhu následujících let po jeho vzniku odvozeno několik dalších algoritmů, nicméně základní myšlenka je ve své podstatě stále stejná jako v citovaném původním zdroji. Práce obsahuje také porovnání kompresí DXT s autorovým algoritmem L3VQ.

Komprese je relativně pomalá, ovšem algoritmus je vyvážen velmi rychlou dekompresí s možností náhodného přístupu na libovolný voxel. K jeho dekompresi není potřeba dekomprimovat žádné jiné voxely. Nevýhoda algoritmu je stejná jako v případě DXT kompresí. Zmenšení dat má za následek změnu reprezentace dat. Tato metoda tak není vhodná pro komprese vědeckých dat, ale hodí se spíše pro multimediální využití (hry, filmy).

Jednotlivé kroky algoritmu jsou uvedeny v sekci Příloha A.

5.2.3 Waveletová transformace

U většiny algoritmů pracujících s objemovými daty se setkáváme s kompresí založenou na waveletech (více informací vi.z Salomon [31]). Zde uvedený popis je pouze stručné shrnutí metody pro doplnění používaných metod.

Vlastní algoritmus se skládá z několika kroků. Data jsou rozdělena na jednotlivé bloky stejné velikosti. Každý tento blok je komprimován samostatně s různými parametry. V rámci bloků je, vzhledem k použitým filtrům propustí, potřeba provést normalizace úrovní. Vstupní hodnoty jsou pak rozloženy okolo nuly.

Nad předzpracovanými daty je spuštěna diskrétní waveletová transformace (DWT). Výsledek transformace je ovlivněn použitým filtrem (např. Haarův, Daubechies). Počet úrovní transformace závisí na konkrétní implementaci. Koeficienty z každé úrovně jsou dále kvantizovány a nahrazeny indexem. Výsledné transformované bloky jsou dále komprimovány, nejčastěji statistickými kompresními algoritmy (aritmetické kódování).

Výhodou waveletových kompresí je, že mimo ztrátové komprese umožňuje i bezeztrátovou. Metoda je velmi často využívána v praxi. Nejznámější zástupcem metody je formát JPEG2000, resp. jeho revize JP3D [4]. Zatímco JPEG2000 komprimuje samostatně jednotlivé řezy, JP3D bere v úvahu i řezy mezi sebou a komprimuje celá data najednou v rámci objemu. Pro objemová data je JPEG2000 součástí formátu DICOM [3].

6 Vizualizace dat

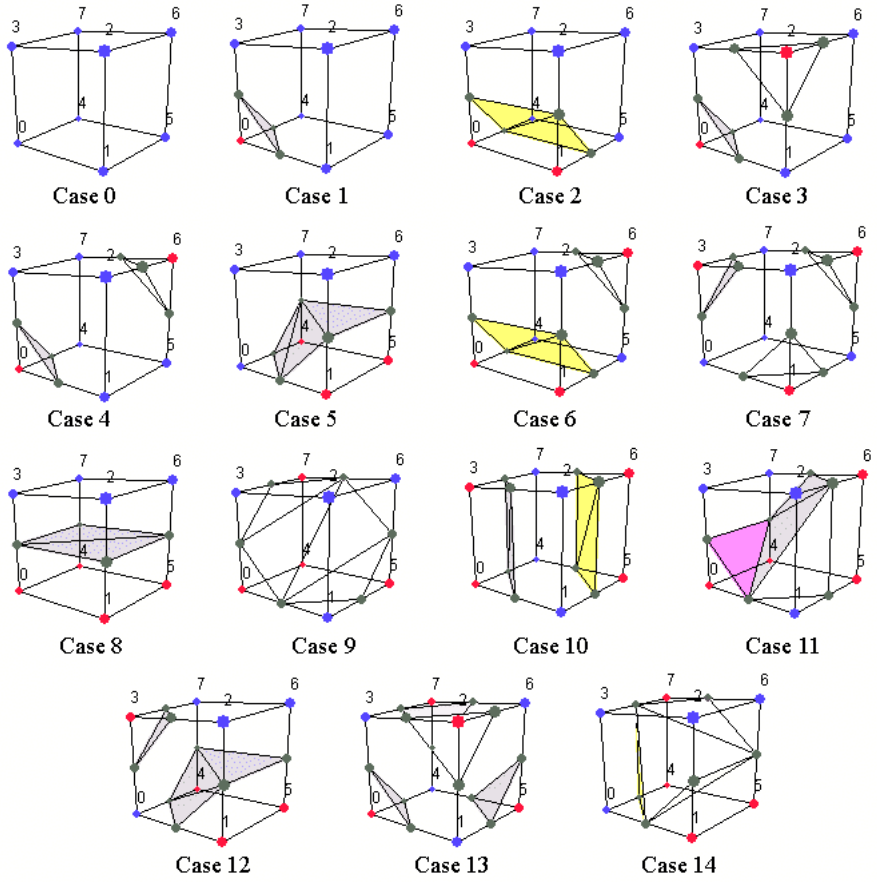
Jak bylo zmíněno v úvodní kapitole, existují dva základní přístupy pro vizualizaci dat z objemových reprezentací. První typ převádí data na trojúhelníkovou síť (iso-plochu). Základní metodou v této oblasti je již po dlouhá léta Marching Cubes [17]. Metoda je velmi jednoduchá, ale zároveň neefektivní, co se týče počtu vzniklých trojúhelníků povrchu. Pro svojí jednoduchost je používána jako základní a hrubý vizualizátor dodnes.

Druhou množinou metod jsou postupy založené na sledování paprsku. Tyto metody jsou obecně vhodnější pro globální vizualizace, kdy nás nezajímá pouze jedna iso-plocha, ale potřebujeme informace o celém objemu zobrazit najednou.

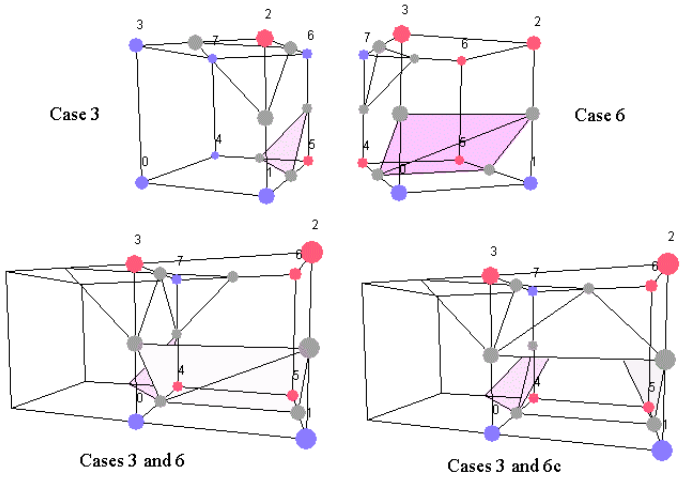
6.1 Extrakční metody

V této oblasti existuje mnoho různých metod. Vzhledem k zaměření práce byla otestována pro porovnání pouze základní metoda Marching Cubes [17]. Jedná se o nejjednodušší O(n) algoritmus sloužící pro extrakci iso-ploch z volumetrických dat. Poprvé byl představen v roce 1987. Algoritmus existuje také ve dvourozměrné verzi, která slouží pro extrakci iso-čar. Tato modifikace je označována jako Marching Squares.

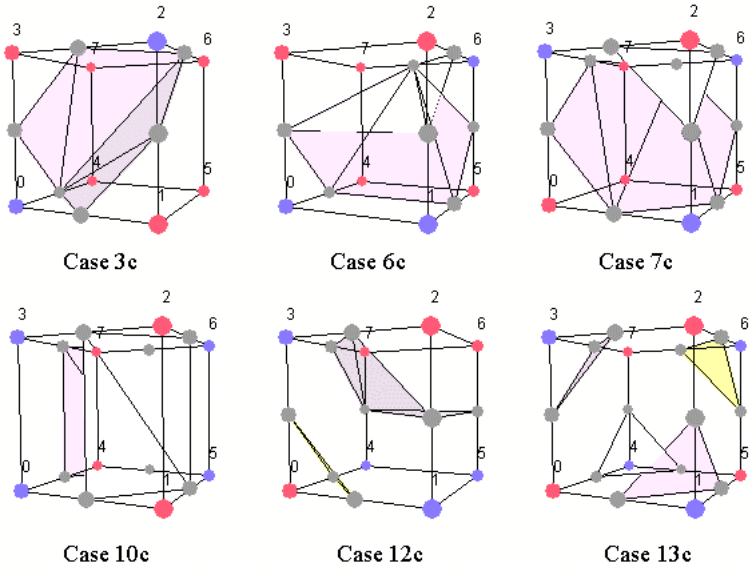
Vstupní voxelová síť je rozdělena na jednotlivé krychlové buňky, které jsou vstupem do extrakční části. Každá z buněk má ohodnocené vrcholy podle hodnot voxelů v jejích vrcholech. Celý proces si můžeme představit tak, že nad voxelovou mřížkou stavíme další mřížku buněk. Pro každých osm sousedních voxelů získáme jednu buňku. Následně jsou vrcholy buňky ohodnoceny pravdivostní hodnotou true / false na základě zvoleného prahu. Je-li iso-hodnota ve vrcholu menší než tento práh, získává vrchol ohodnocení false, v opačném případě true. Na základě těchto ohodnocení vzniká 28 možných konfiguračních stavů. S využitím symetrie lze počet stavů snížit na 15. Těchto unikátních 15 stavů je možno vidět na obrázku 9. Drobným problémem je nerozhodnutelnost některých stavů (z hlediska triangulace) a s ním spojený vznik děr v síti (vyobrazeno na obrázku 10). Nejsnadnější řešení je rozšíření tabulky o doplňkové stavy, uvedené např. v [11], které problém vyřeší (viz. obrázek 11).



Obrázek 9: Základní stavy algoritmu Marching Cubes



Obrázek 10: Nerozhodnutelné stavy u algoritmu Marching Cubes



Obrázek 11: Doplňkové stavy

Vzniklá plocha, složená z trojúhelníků, má své řídící body umístěny na hranách buňky. V jaké části hrany se bod nachází určíme lineární interpolací na základě souřadnic vrcholů a jejich iso-hodnot. K výpočtu použijeme následující vzorec

$$u = \frac{prah - iso_1}{iso_2 - iso_1}$$

$$v_i = v_1 + u(v_2 - v_1)$$

$v_{1,2}$ jsou souřadnice vrcholů hrany, $prah$ je zvolená prahová hodnota a $iso_{1,2}$ jsou iso-hodnoty v příslušných vrcholech.

6.2 Metody založené na sledování paprsku

V praxi se používají spíše metody založené na ray-castingu, než na ray-tracingu. Rozdíl mezi nimi je zásadní. Ray-tracing sleduje primární parsek, který se dále dělí a umožňuje tak komplexnější efekty (odrazy, stíny...). Ray-casting naopak vyšle pouze primární parsek, který se nijak neláme, postupně projde objemem a kumuluje hodnoty protnutých voxelů. Celková hodnota je po dodatečných úpravách (např. stínování) zanesena ve výsledném obrazu pro konkrétní pixel, kterému paprsek odpovídal.

Abychom mohli metodu aplikovat, musíme nejprve pro každý pixel obrazu spočítat odpovídající paprsek. Základní struktura a rozložení paprsků je vyobrazeno na obrázku 12. Seznam použitého značení je uveden v tabulce 3. Kamera sleduje bod L, který je umístěn za projekční rovinou. Vzdálenost kamery od projekční roviny je označena jako D. Naším cílem je vytvořit ortonormální bázi $[\overrightarrow{u}, \overrightarrow{v}, \overrightarrow{w}]$ v bodě C (neboli střed obrazovky). K určení těchto tří vektorů použijeme rovnice 6 - 10. Vektor [0, 1, 0] v rovnici 9 značí vektor, který směřuje vzhůru. Zde předpokládáme, že pozorovatel má

výšku určenou osou Y. Vzniklý systém nám umožňuje vyjádřit libovolný bod obrazu P, jak je uvedeno v rovnici 11.

Označení	Význam
D	vzdálenost kamery od projekční roviny; D = |C - E|
H	výška projekční roviny
fovY	pozorovací úhel v ose Y
C	střed obrazovky
E	pozice kamery
L	bod sledovaný kamerou
Width	horizontální rozlišení obrazu
Height	vertikální rozlišení obrazu
$\overrightarrow{u}, \overrightarrow{v}, \overrightarrow{w}$	ortonormální báze
$\otimes$	vektorový součin

Tabulka 3: Vysvětlení použitého značení v rovnicích

$$D = \frac{H}{2} \cdot \frac{1}{tan(\frac{fovY}{2})} \tag{6}$$

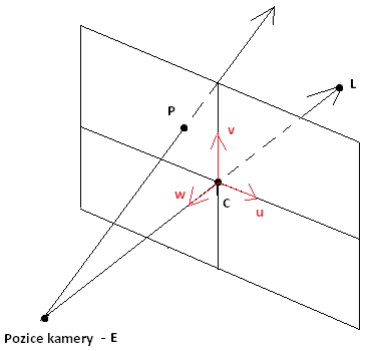
$$\overrightarrow{w} = \frac{E - L}{|E - L|} \tag{7}$$

$$C = E - D\overrightarrow{w} \tag{8}$$

$$\overrightarrow{u} = [0, 1, 0] \otimes \overrightarrow{w} \tag{9}$$

$$\overrightarrow{v} = \overrightarrow{w} \otimes \overrightarrow{u} \tag{10}$$

$$P = [a, b, e] : \quad P = C + a\overrightarrow{u} + b\overrightarrow{v} + e\overrightarrow{w} \tag{11}$$



Obrázek 12: Projekční rovina a paprsky od kamery

Bod P je určený v rámci projekční roviny. Souřadnice bodu ovšem neodpovídají přímo souřadnici pixelu, protože počítáme se středem C uprostřed obrazu, zatímco indexace pixelů bere jako počáteční bod [0,0] levý horní roh. Souřadnice bodu tedy převedeme tak, aby odpovídaly našemu obrazu se středem v bodě C. Finální výpočet je pak uveden v rovnici 14.

$$[x, y] - originální\ souřadnice\ pixelu$$

$$[x\acute{}, y\acute{}] - souřadnice\ v\ projekční\ rovině$$

$$x\acute{} = x - \frac{Width}{2} \tag{12}$$

$$y\acute{} = \frac{Height}{2} - y \tag{13}$$

$$P = C + x\acute{}\overrightarrow{u} + y\acute{}\overrightarrow{v} \tag{14}$$

Novou souřadnici bodu použijeme k výpočtu směru paprsku podle rovnice 15. Počáteční pozice paprsku je určena bodem P.

$$\overrightarrow{Dir} = \frac{P - E}{|P - E|} \tag{15}$$

Výsledný paprsek je ještě potřeba upravit, aby odpovídal matici modelu. Pokud bychom ho nechali bez této transformace, model by měl vždy matici rovnu identitě. Bod P a směr Dir přenásobíme inverzní transpozicí matice modelu. Směr je potřeba ještě dodatečně normalizovat. Nyní máme dostupné paprsky v každém z bodů obrazu a můžeme spustit vlastní ray-casting.

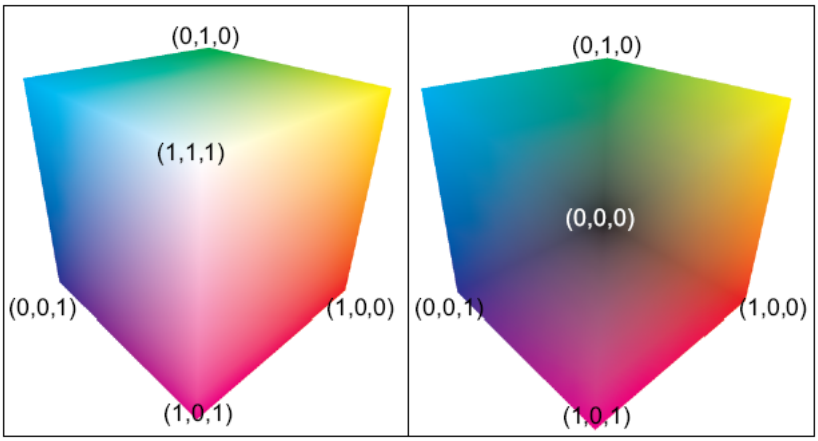
V rámci ray-castingu musíme nejprve identifikovat, zda paprsek protíná objemová data či nikoliv. Okolo objemových dat vytvoříme obalový box a vypočítáme průnik paprsku s tímto obalem. Pokud paprsek objem neprotne, nemusíme dále počítat a můžeme algoritmus ukončit. V daném pixelu bude nastavena výchozí barva pozadí (nebo jiná, námi definovaná na základě požadavků). V opačném případě spustíme vlastní průchod objemem, viz. algoritmus 3. Hodnoty *tNear* a *tFar* odpovídají parametrům paprsku při průsečíku s obalovým tělesem. Celý algoritmus iterace je ukončen buď dosažením prahové hodnoty průhlednosti v kumulovaném alfa kanálu, nebo opustí-li paprsek objem. Data jsou reprezentována jako raw v podobě 3D textury. Výhoda je automatická interpolace hodnot mezi voxely, podobně jako u klasické 2D textury.

Algoritmus 3 Základní algoritmus Ray-castingu

```
float t = tNear;
float3 pos = ray.P + ray.dir * tNear;
float3 step = ray.dir + stepSize;
float4 dstColor = float4(0.0f);
for (int i = 0; i < MAX_STEPS; i++)
{
        texCoord3D = float3(pos / dataSize);
        float value = tex3D(volumeData, texCoord3D);

        float4 actColor = float4(value);
        actColor.w *= 0.5f;
        actColor.xyz *= actColor.w;

        dstColor = (1.0f - dstColor.w) * actColor + dstColor;

        if (dstColor.w > maxOpacity) break;
        t += stepSize;
        if (t > tFar) break;
        pos += step;
}
```

Algoritmus 3 má zásadní výhodu ve své univerzálnosti. Není potřeba řešit žádné speciální případy, kdy je pozorovatel uvnitř objemu nebo pokud je objem zcela mimo obrazovku apod. Bohužel, za univerzálnost se platí, v tomto případě výkonem.

Kroky z rovnic 6-15 lze použitím metody navržené v [15] eliminovat, resp. převést na jiný typ problému. Tato metoda v předzpracování vykreslí dvojici obalových těles našeho objemového modelu. Ve většině případů se jedná o krychli, popř. kvádr. V rámci jednoho vykreslení pracujeme s ořezáváním na přilehlé, podruhé na odlehlé stěny. Pozice krychle je transformovaná na pozici modelu. Kromě vlastní pozice je potřeba pro každý vrchol znát také jeho reprezentaci v rámci jednotkové krychle (lze použít texturovací souřadnice). V pixel shaderu potom vykreslíme jednotkové souřadnice jako barvu, kterou reprezentují. Grafická karta se sama postará o interpolaci souřadnic mezi sousedními vrcholy. Výsledná dvojice obalových těles je vyobrazena na obrázku 13. Výsledný směr paprsku v každém bodě je pak dán rozdílem "barev" (které odpovídají souřadnicím) v tomto bodě.



Obrázek 13: Vykreslené obalové těleso pro eliminaci výpočtů směru paprsku. V levé části je vykreslená přední strana, v pravé části pak zadní stěny krychle

Hlavní nevýhodou přístupu pomocí vykreslování obalového tělesa je situace, kdy se s kamerou nacházíme uvnitř objemových dat a tudíž bychom přední stěny krychle neviděli. V tomto případě musíme určit řez krychlí a místo přední stěny vykreslit hodnoty řezu. Opět nám stačí určit krajní body řezu a grafická karta automaticky zařídí interpolaci hodnot.

Kromě klasického sledování paprsku pomocí algoritmu 3 můžeme použít také další přístupy. Algoritmy sledování paprsku primárně počítají s tím, že máme možnost pracovat s celým objemem jednotně. Na základě kroku algoritmu se nám ale může stát, že některé buňky čteme opakovaně či jiné naopak přeskočíme. Je-li naším cílem procházet všechny hodnoty ležící v cestě paprsku pouze jedenkrát, musíme využít některý algoritmus z oblasti DDA (Digital Differential Analyze) nebo algoritmy založené na 3D verzi Bresenhama. Rozdíl mezi DDA a Bresenhamem je v několika bodech uvedených v tabulce 4. Nevýhoda v těchto případech je vzniklá zubatost hran, popř. problikávání modelu vlivem zaokrouhlovacích chyb.

	DDA	Bresenham
aritmetika	float	int
zaokrouhlování	na nejbližší celé číslo	-
přesnost	nižší	vyšší
použité operace	+,-,/,*	+,-

Tabulka 4: Rozdíl mezi DDA a Bresenhamem

Pro reprezentaci dat stromem lze použít rychlou iterační metodu uvedenou v [29] [2]. Metoda lze snadno implementovat i na GPU, protože rekurzivní část algoritmu lze přepsat na iterační kroky. Bohužel efektivita implementace na GPU není příliš vysoká z důvodu architektury GPU. Během iterace je potřeba udržovat cestu sestupu paprsku stromem. Ta se nevejde do registrů ani do sdílené paměti a musí být umístěna v globální

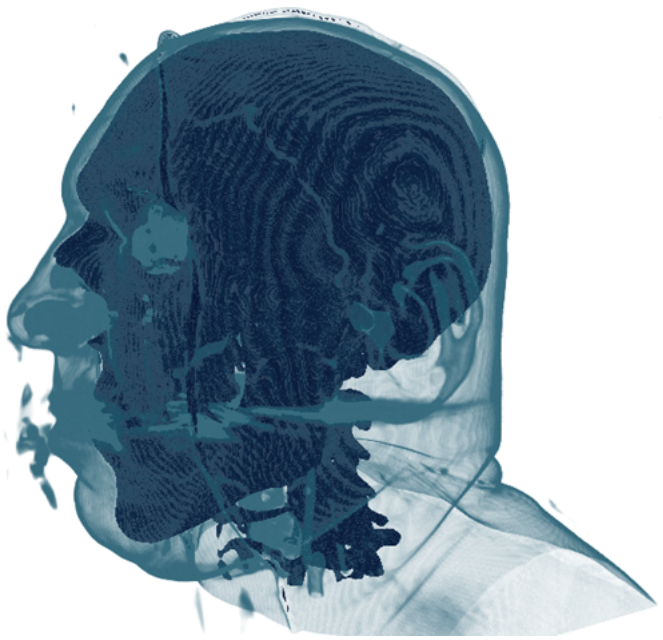
[2] Algoritmus popsaný v článku je chybně. Na stránce 4 v tabulce 1 je prohozena 0 za 2 a naopak (tzn.: $0 \rightarrow 2$ a $2 \rightarrow 0$)

paměti, což snižuje znatelně rychlost. Obecně nejsou stromové struktury na GPU příliš vhodné právě pro nutnost rekurzivního průchodu. Existují sice algoritmy, které tuto potřebu částečně eliminují, ovšem ty jsou určené pro binární stromy (např. BSP). Tyto algoritmy i tak ovšem využívají zásobník, pouze omezují jeho velikost tak, aby nebylo třeba používat globální paměť. V případě, kdy je zásobník naplněn se iterace restartuje a zásobník se plní znovu.

Obecný problém, který se týká všech algoritmů pro rasterizace je zvýraznění jednotlivých voxelů. S tímto problémem se nesetkáváme u 3D textur vzhledem k automatické interpolaci hodnot mimo vlastní mřížku. Možné řešení je vypočítat pro každou buňku procentuální podíl, jaký zabírá protnutá část a na základě toho upravit kumulaci dat v daném bodě.

6.3 Barevné interpretace dat

Finálním krokem může být v některých případech obarvení dat. Pokud data již přímo vyjadřují hodnoty RGB, nemusíme se tímto krokem zabývat a můžeme použít tyto hodnoty přímo. V případě jiných reprezentací můžeme data dodatečně obarvit podle předem zvolených prahů přes tzv. přenosovou (transfer) funkci, která může být předpočítaná v podobě textury. Přenosové funkce často kromě vlastní barvy ukládají také informaci o průhlednosti dat. Tato hodnota je uložena v alpha kanálu a umožňuje zobrazovat najednou různé vrstvy, např. průsvitná kůže a pod ní pevná kost. Příklad výsledku při použití takové funkce je vidět na obrázku 14. Popis jedné z možných metod pro generování přenosové funkce v textuře lze nalézt v [10].



Obrázek 14: Ukázka přenosové (transfer) funkce (zdroj: [10])

V případě jednoduchých celočíselných dat postačí tato klasická přenosová (transfer) funkce. Její velikost v rámci 8 nebo 16 bitových dat je v poměru k velikosti dat zane-

dbatelná. Pro více bitová data je již velikost případné textury problematická. V tomto případě se jako rozumnější volba ukazuje použití přenosových funkcí vypočtených v reálném čase. Musíme brát ovšem v úvahu určité limity spojené s vizualizací. Automaticky spočtené funkce se nemohou rovnat uživatelsky definovaným funkcím nebo manuální volbě parametrů, abychom byli schopni zdůraznit určité aspekty dat.

Hlavní problém s přenosovými funkcemi je použitý typ. Podle typu dat může být přenosová funkce velmi různorodá. Nalézt univerzální řešení znamená udělat určité kompromisy. Jedním z navržených je přechod založený na Gausově funkci [13]. Její výhodou je jednoduchost, kdy je potřeba pouze několik operací k vypočtení finální hodnoty. Metodu lze snadno rozšířit také do více dimenzí, což je užitečné pro data zobrazující více informací v jednom snímku (rychlost, teplotu, tlak atd.).

7 Statická data

Základem volumetrických zobrazování jsou statická data, neboli jeden snímek. Pro zobrazování těchto dat lze využít mnoho algoritmů, z nichž je v rámci této práce implementováno a otestováno pouze několik. Většina metod se v praxi neustále opakuje, mají stejný základ, pouze jiné nadstavby. Z toho důvodu byly zvoleny základní algoritmy, na kterých ostatní následně staví a rozšiřují je.

Jednotlivé metody v této kapitole jsou nejdříve základním způsobem popsány, následně jsou provedena různá výkonnostní měření pro jejich porovnání. Je nutno konstatovat, že kvalita obrázků uvedených v této práci je silně ovlivněna vlastní kvalitou tisku této diplomové práce. Některé detaily tak mohou zanikat. Zde uvedené obrázky proto slouží spíše jako základní ilustrační náhled.

7.1 Některé dostupné metody

Metody založené na extrakci dat se ve většině případů zaměřují na zjednodušení trojúhelníkové sítě. Základní metoda Marching Cubes [17] je sice velmi jednoduchá, ovšem produkuje velké množství trojúhelníků. Ty jsou navíc často ještě příliš úzké. Tento problém odstraňuje např. Dietrich [6]. Vzhledem k cílení práce na vizualizace nebyly extrakční techniky dále studovány.

V oblasti vizualizace statických snímků bylo v průběhu let navrženo mnoho různých algoritmů. Hlavním cílem těchto metod je zaručit náhodný přístup k datům v případě, že chceme provádět dekompresi spolu s vizualizací. V případě nutnosti data nejprve rozbalit a pak je teprve vizualizovat bychom se potýkali opět s problémem velikosti snímků a musel by se aplikovat odlišný postup vizualizace.

Mezi metodami s náhodným přístupem se nejčastěji v literatuře vyskytují algoritmy založené na dříve zmíněné vektorové kvantizaci. Jedním z prvních algoritmů v této oblasti byl právě testovaný L3VQ [32]. Metody založené na kvantizaci lze nalézt také v [16]. Zde je navíc datová množina před vlastní kvantizací upravena pomocí informací získaných z histogramu.

Další, velmi často využívanou metodou, jsou komprese založené na waveletové transformaci. V této oblasti je za standard považován formát JPEG2000, resp. jeho trojrozměrná revize JP3D. Další oblasti využití a práce s wavelety lze nalézt v [9]. Tato metoda spojuje waveletové komprese s kvantizacemi. Podobné spojení metod lze nalézt také např. v [27].

V rámci statických snímků musíme uvažovat také jejich výsledné vyobrazení. Zde se velmi často využívají přenosové funkce, např. [13]. Další možností dodatečného vylepšení obrazu je využít metodu popsanou v [7]. Podle autorů má ovšem tato metoda určité problémy s polo-průhlednými daty. Je navržené řešení, které ovšem snižuje výkon. Metoda je dále využívána a vylepšena např. v [30].

7.2 Testované algoritmy

7.2.1 Extrakce iso-ploch pomocí Marching Cubes

Tato metoda se velmi snadno paralelizuje, nepotřebuje pro svůj běh žádné speciální struktury a tudíž je vhodná pro implementaci na GPU. Jediný problém, na který zde narážíme, je generování trojúhelníků. Ty je potřeba vkládat do globálního seznamu. Ten může být buď reprezentován jako paměť vertex bufferu pro pozdější vizualizaci nebo obecné datové úložiště definované uživatelem. V každém případě ovšem nastává problém s vkládáním, protože vlákna mohou přistupovat k zápisu v různém pořadí a být v různý čas přerušena. Zápis do sdílené paměti tak musíme ošetřit například zámkem. To celou extrakci znatelně zpomaluje.

7.2.2 Klasická verze sledování paprsku

Tato metodu byla implementována jak v jazyce CUDA, tak pomocí výpočetních (Compute) shaderů. Obě verze jsou z algoritmického hlediska naprosto totožné. Snaha o identické programové kódy nešla za všech okolností dodržet, ale ve většině případů se kód shoduje.

Algoritmus pracuje s výpočtem paprsku dle rovnic 6 - 15 uvedených v kapitole 6.

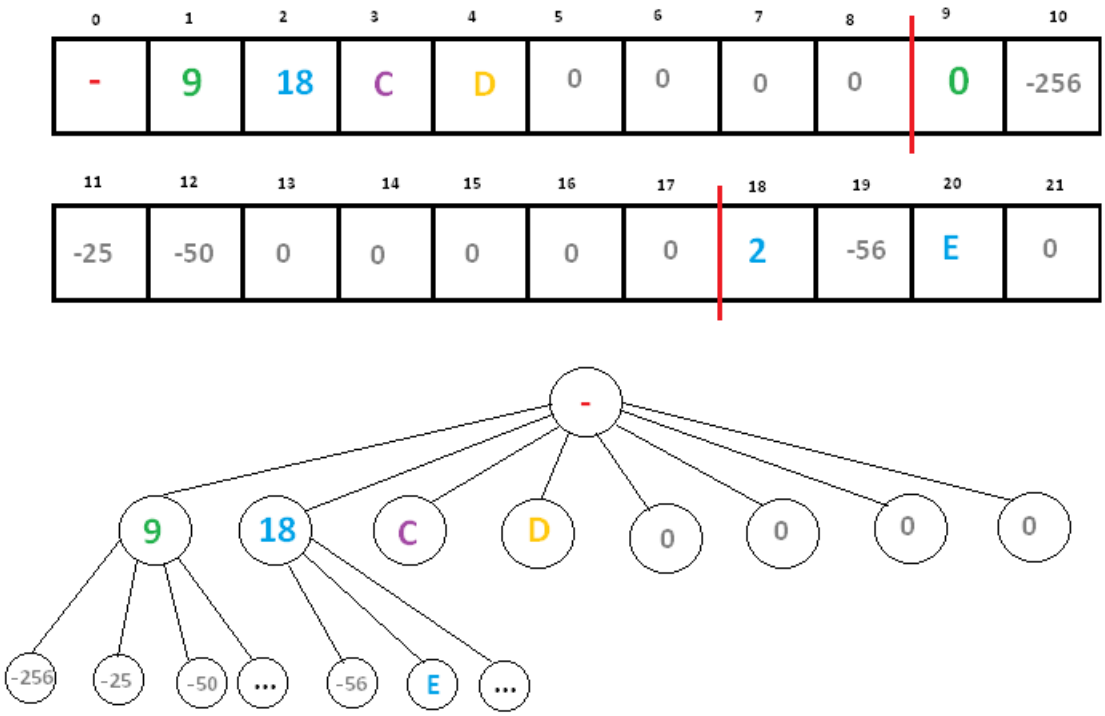
Dvojí implementace byla zvolena z důvodu porovnání technologií. Jelikož shadery pracují přímo s DirectX11, byla zde domněnka, že by mohly shadery přinést vyšší výkon než využití CUDY. Tam je potřeba textury a data plnit odlišným způsobem. Využito je rozhraní, které není transparentní a není tak jasné, co se na pozadí skutečně odehrává za operace. Zároveň není vykreslování příliš pohodlné a i dokumentace k propojení CUDY s DirectX11 je značně nedostačující. V případě shaderů je jejich propojení s grafickým API DirectX11 přeci jenom užší a více propracované. V části 7.4 nazvané Testy jsou pak porovnání shrnuta.

7.2.3 Průchod oktalovým stromem

Pracujeme-li se stromy klasickým přístupem na CPU, využijeme k jejich reprezentaci nejčastěji ukazatele. Můžeme stromové struktury také reprezentovat pomocí polí, nicméně tento přístup se na CPU prakticky nepoužívá. Pro programátora je přeci jenom pohodlnější a přehlednější pracovat s ukazatelem "left" než ručně počítat index levého potomka. Navíc během výpočtu se zvyšuje možnost zanesení chyby a její následné hledání není příliš pohodlné.

Na GPU s využitím CUDY jsou ukazatele také dostupné, ovšem pro nahrání dat na GPU musíme použít nějakou pomocnou reprezentaci. Strom byl uložen jako 1D pole, kdy hodnoty v poli jsou indexy potomků. Pokud se jedná o list, je uložená hodnota záporná nebo nula. Reprezentace stromu je vyobrazena na obrázku 15. Každý uzel je reprezentován devíti čísly. První číslo udává index rodiče, dalších osm jsou pak postupně indexy potomků nebo přímo hodnoty. Pořadí je pevně dané a vyjadřuje pozici uzlu v rámci stromu.

Nevýhoda je obecná náročnost stromových struktur pro vizualizace poloprůhledných dat, tak jak bylo uvedeno v kapitole 4.



Obrázek 15: Uložení oktalového stromu jako 1D pole

K průchodu stromem byl použit algoritmus uvedený v [29], který byl upraven na iterační verzi využívající externí frontu pro kumulaci navštívených uzlů. Tato fronta je uložena v globální paměti, a proto limituje výkon. Každé vlákno má svojí frontu, do níž zapisuje a čte. Fronta by mohla být uložena ve sdílené paměti pro rychlejší přístup, nicméně každý záznam ve frontě se skládá ze šesti 32-bitových čísel v plovoucí řádové čárce (pro určení vstupního a výstupního parametru z buňky) a jednoho 32-bitového celočíselného indexu buňky, což nám dává 7 x 32bit. Při omezené velikosti sdílené paměti na 48KB bychom do ní mohli uložit cca. 1700 hodnot pro všechna vlákna, která ji sdílí. Běží-li v rámci bloku 256 vláken, dostaneme pouze 6 hodnot, což je pro velká data příliš malá hodnota. Tuto hodnotu bychom mohli zvětšit použitím 16-bitové reprezentace desetinných čísel, ovšem výsledný počet stavů by i tak byl příliš nízký. Po zaplnění by bylo potřeba zařídit, aby se fronta odsunula do globální paměti a vyprázdnila. V případě potřeby by se pak fronta opět nahrála z globální paměti. Toto urychlení nebylo, vzhledem k malé praktické použitelnosti oktalového stromu pro naše potřeby, implementováno.

Rekurzivní algoritmus z minulého odstavce byl nahrazen vlastní iterační implementací, která nepotřebuje využívat zásobníky ani fronty. V každém kroku na základě předchozí pozice vypočítáváme hloubku, průsečíky s uzlem (k tomu lze využít např. algoritmus z [20]) a novou pozici. Algoritmus je výpočetně značně náročnější, ale obsahuje menší počet zásahů do globální paměti. Z té je navíc potřeba pouze číst. Algoritmus ovšem není vhodný pro reprezentaci průhledných dat, hodí se spíše pro ukončení průchodu po nalezení povrchu. Kompletní algoritmus je uvedený v sekci Příloha B. Tato verze pracuje se stromovou reprezentací pomocí pole, jak bylo uvedeno na obrázku 15.

7.2.4 L3VQ

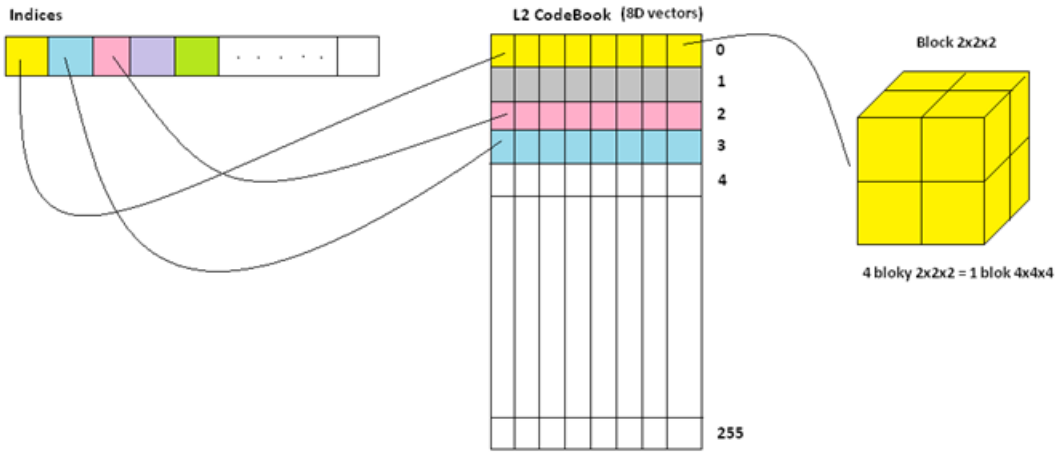
Ze ztrátově kompresních formátů byl testován algoritmus L3VQ (základní kompresní myšlenka je shrnuta v sekci Příloha A, další informace lze nalézt v originální publikaci viz. [12] a [32]). Zabývat se DXT algoritmy je zbytečné, neboť kvalitativně jsou na tom podobně a navíc nabízejí horší kompresní poměry. Kompresní algoritmus je oproti dekompresi citelně pomalejší, ovšem to nám ve většině případů nijak nevadí. V rámci komprese nebylo uvažováno žádné urychlení algoritmu, protože to není pro naše účely potřeba. Během testů rychlosti vykreslování nás kompresní čas nezajímá. Data komprimujeme většinou pouze jednou a následně je hlavní faktor pro výkon dekompresní čas.

Pro dekompresi během vykreslování lze využít dvě cesty. První z nich, použitá v originálním článku, pracuje s metodou podobnou deferred renderingu[3]. Obraz se rozdělí na několik desítek (stovek) plátů a každý plát se vykreslí samostatně. V dalším kroku se jednotlivé pláty prolnou do výsledného obrazu. V rámci toho získáme filtraci dat bez nutnosti jejího ručního počítání pro každý voxel. Každý voxel tak čteme pouze jako v případě nearest neighbour filtrace. Této metodě k dekompresi stačí pouze jednoduchý kód uvnitř pixel shaderu. V dnešní době jsou ale grafické karty na jiné úrovni, než v době psaní článku [32] a jejich výkon je na plnou dekompresi dostatečný.

Druhá možnost je provést na GPU kompletní dekompresi stejným způsobem jako by byla vypočtena na CPU. Tato možnost byla zvolena v rámci ukázkové implementace.

V první ukázce algoritmu 4 je dekomprese v CPU podání. Struktura *compressedData* uchovává informace o komprimovaných datech. V analogii s obrázkem 16 je hodnota na pozici *blockIndex* žlutý blok vpravo. Hodnota *indexL2* určuje hlavní offset v tabulce *L2Codebook* (žlutá řádka). V rámci tohoto bloku máme 8 hodnot pro úroveň L2. Z indexů *i*, *j*, *k* určíme 1D index v rozsahu 0 - 7. Indexy *i*, *j*, *k* jsou ovšem po přepočítání v rozsahu 0 - 63. Tento rozsah pomocí pohledové tabulky přemapujeme na požadovaný 0 - 7. Tento offset nám zaručuje hodnotu v rámci získané řádky (*indices*). Výslednou dekomprimovanou hodnotu určíme z hodnoty na příslušné pozici v řádce *indices*. Tu z intervalu 0 - 255 přenesme na interval 0 - 1 a vynásobíme škálovacím koeficientem pro kvalitu L2.

[3]Jedná se o metodu vykreslování, kdy jsou data nejprve vykreslena do pomocných textur. V dalších průchodech se pak již nepracuje se skutečnou geometrií, ale s pomocnými texturami. Z nich lze získat potřebné informace, např. normály, hloubku, materiály.



Obrázek 16: Ukázka vztahu mezi indexy a kódovací množinou

Algoritmus 4 Dekomprese dat z bloku L2 na CPU

```
uint8 index = compressedData.volumeData[blockIndex].indexL2;
uint32 offset = lookUp4x4x4To2x2x2[i + 4 * j + 16 * k];
uint8 codeWord = compressedData.codeBookL2[8 * index + offset];
double value = compressedData.scaleL2 * (static_cast<double>(codeWord) / 127.5 - 1.0);
```

V případě dekomprese na GPU (algoritmus 5) si do textury předpočítáme dekomprimované hodnoty. Zvýší se nám tím sice potřeba úložného prostoru na texturu 4krát (původní data měla velikost 1 byte, nyní máme float hodnoty o velikosti 4 byte), ovšem urychlíme čas dekomprese. V předpočítané textuře máme hodnoty odpovídající poslední řádce z algoritmu 4.

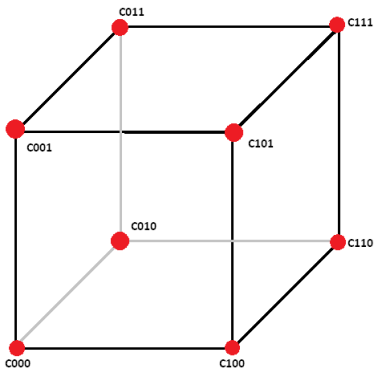
Algoritmus 5 Dekomprese dat z bloku L2 na GPU

```
const uint lutTable[64] =
{
0,0,1,1, 0,0,1,1, 2,2,3,3, 2,2,3,3,
0,0,1,1, 0,0,1,1, 2,2,3,3, 2,2,3,3,
4,4,5,5, 4,4,5,5, 6,6,7,7, 6,6,7,7,
4,4,5,5, 4,4,5,5, 6,6,7,7, 6,6,7,7
};

uint offset = lutTable[index.modIndex.x + 4 * index.modIndex.y + 16 * index.modIndex.z
    ];
float value = CodeBookL2_buf[8 * L2Index + offset];
```

Podobným způsobem dekomprimujeme také blok L1. Pro blok L3 není potřeba dekomprese, protože hodnota uložena v bloku L3 odpovídá přímo hodnotě, kterou budeme používat jako referenční.

Trilinearní filtraci je potřeba vypočítat pomocí vlastního algoritmu, nelze použít automatickou filtraci poskytovanou texturami. Zde by totiž došlo k filtraci komprimovaných hodnot a následně bychom již nemohli tyto hodnoty správně dekomprimovat. Pro případnou trilineární filtraci je potřeba si algoritmus napsat vlastní. Jako vstupní

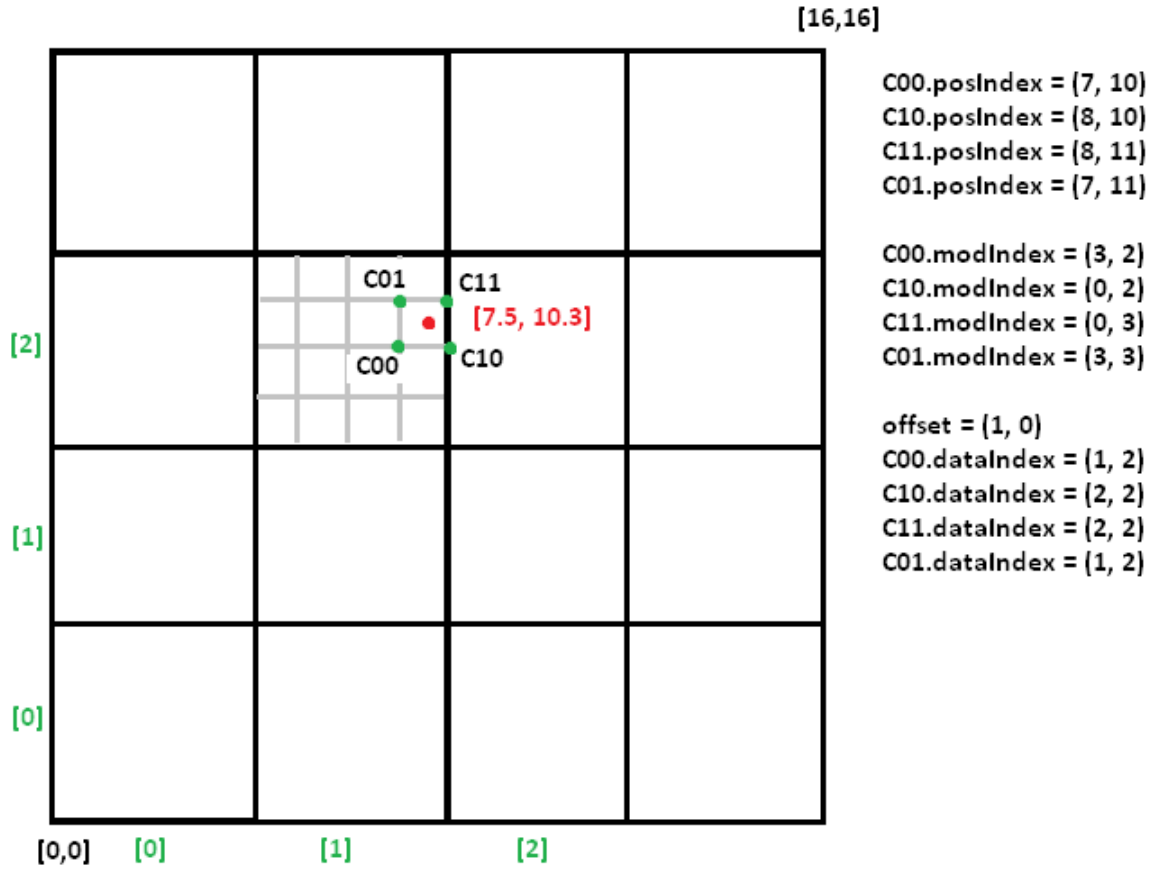
hodnota algoritmu postačuje pouze pozice *pos* ve světových souřadnicích. Pomocí této pozice určíme souřadnice jednotlivých rohů *C000* - *C111* (jednotlivé hodnoty komprimovaných voxelů v algoritmu značené jako *Cxyz* odpovídají značení na obrázku 17).



Obrázek 17: Značení komprimovaných voxelů pro popsaný způsob trilineární filtrace

Jako vstupní hodnota algoritmu postačuje pouze pozice *pos* ve světových souřadnicích. Pomocí této pozice určíme souřadnice jednotlivých rohů *C000* - *C111*. Abychom určili, jaké hodnoty musíme načítat, pracujeme s pozicí modulo čtyřmi. Tím zjistíme souřadnici konkrétního bloku (každý blok se skládá ze čtyř dekomprimovaných hodnot). Výpočet hodnoty *offset* nám určuje, které okolní hodnoty budeme potřebovat načíst, abychom dekomprimovali voxel na aktuální pozici. Ve většině případů tak potřebujeme načíst osmkrát stejný blok a ten pak dekomprimovat. V nejhorším případě, pokud bychom se nacházeli ve výchozí pozici *C111*, budeme potřebovat načíst všech osm sousedních bloků. Na základě této myšlenky můžeme eliminovat počet čtení z globální paměti a ručně zařídit cache, čímž se nám trilineární filtrace počtem přístupů do paměti přiblíží nearest neighbour filtraci. Podobný přístup je využíván při dekompresích DXT textur, kdy se do texturovací cache načte celý blok a následná filtrace využívá těchto načtených hodnot.

Popsaný algoritmus se nejlépe dokumentuje na obrázku 18. Pro jednoduchost a názornost je uvedena pouze 2D verze. Červený bod o souřadnicích [7.5, 10.3] je vstupní pozice. Zelené body jsou jednotlivé body použité pro interpolaci. Z výpočtu je vidět, že potřebujeme dva body z dlaždice [1, 2] a dva body z dlaždice [2, 2]. Místo čtyř nezávislých čtení můžeme číst z paměti pouze dvakrát, čímž ušetříme značnou část výkonu. Hodnota *modIndex* slouží k určení pozice uvnitř dlaždice a musí být kombinována s vypočteným indexem dlaždice (*dataIndex*).



Obrázek 18: Ilustrační rozkreslení jednoho konkrétního kroku algoritmu trilineární filtrace pro vstupní pozici [7.5, 10.3]. Prezentována je pouze 2D verze pro snažší pochopení ilustrace.

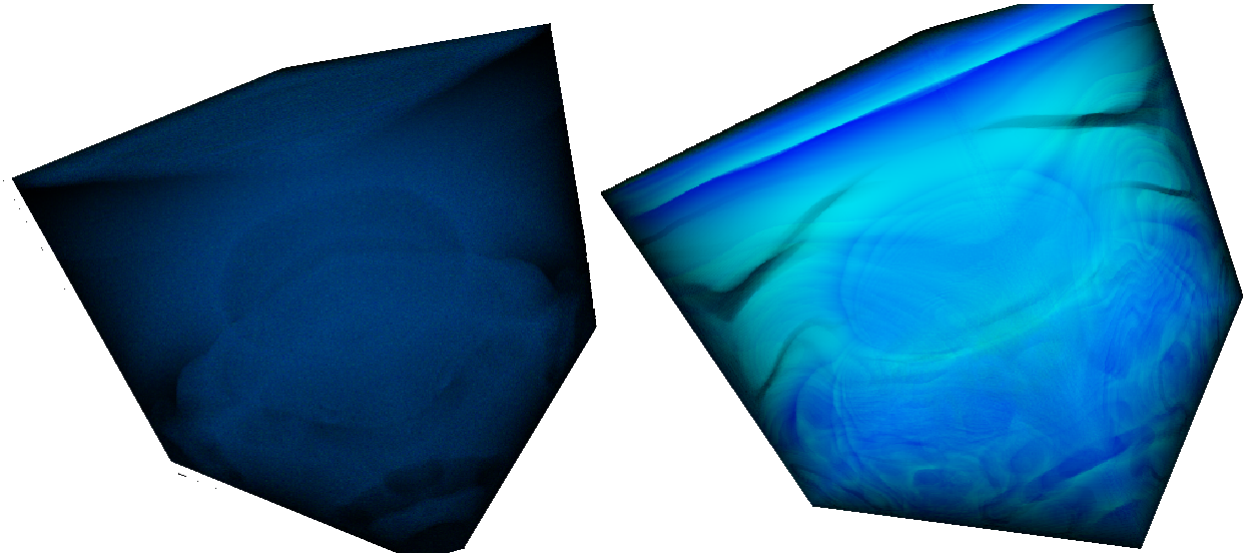
7.3 Vizualizace

Vlastní vizualizace zobrazuje data pomocí sledování paprsku, metodou ray-casting. Na základě získaných vzorků je třeba data dodatečně upravit. V případě jednoduchých 8 nebo 16 bitových dat nebyly využity žádné přenosové funkce a datová hodnota přímo reprezentovala odstín šedi (případně dvojici barev). Stejný přístup lze aplikovat i pro použití u 32 bitových dat. Pokud jsou ovšem vizualizovaná data tvořena desetinnými čísly, narážíme na limitace. Vzhledem k nepřesnostem nám v datech vznikají artefakty a šum. Použití klasické předpočítané přenosové funkce má příliš malý rozsah, jeho zvětšení pak zvětšuje potřebný prostor pro data, což je problém (viz. další kapitola 8).

Pro jednoduchost byla použita klasická statistická metoda pracující s Gausovou funkcí. K úpravě funkce bylo využito poznatků z [13]. Z vizualizovaných dat byla ve fázi předzpracování vypočítána střední hodnota a rozptyl. Tyto hodnoty pak byly využity během vizualizace, kdy se do klasického vzorce pro Gaussovu funkci 16 dosazovaly získané vzorky. Výsledná hodnota se pohybuje v rozsahu 0 - 1 a na základě toho je spočtena výsledná barva. Tato hodnotu je pak z desetinného rozsahu převedena na

RGBA. Výsledek a porovnání s původní verzí je možné vidět na obrázku 19.

$$hodnota = \exp(-\frac{(vzorek - stredni\_hodnota)^2}{2 \cdot rozptyl}) \tag{16}$$



Obrázek 19: Vlevo převod 32 bitové float hodnoty přímo na RGBA reprezentaci, vpravo použití Gaussovy přenosové funkce na raw data

V případě využití L3VQ komprese dojde u dat s desetinnými hodnotami ke ztrátě kvality. Data jsou zkreslena množstvím šumu a nepřesností. Použití přenosové Gaussovy funkce vylepší vizuální výsledek a v datech jsou pozorovatelné charakteristické rysy. Použitím jiných přenosových funkcí bychom možná dosáhli lepšího vizuálního výsledku, ovšem možných funkcí při změnách vstupních parametrů je prakticky nekonečné množství.

Obecně je problematické tento typ dat zachytit na statickém obrázku. V reálné aplikaci lze modelem otáčet a pohybovat, čímž se vizuální kvalita opět změní v závislosti na pozorovacím úhlu a případným dodatečným nastavením průhlednosti dat.

7.4 Testy

7.4.1 Testovací data

V rámci testovacích dat bylo použito několik různých datových setů. Cílem bylo pracovat s daty obsahující různé charakteristické prvky a různá rozlišení pro porovnán metod a rychlostí vykreslování. Některé datové množiny byly vygenerovány pomocí matematických funkcí, u jiných se jedná o snímky pořízené z 3D scannerů. Další testovací data lze nalézt v [2].

Nejjednodušším typem dat je model oblouku, označený jako *Syn64*. Tento model byl zvolen jako výchozí pro testování metod vzhledem k jeho malému rozlišení 64x64x64. Model je velmi jednoduchý a je možnost na jeho struktuře porovnat algoritmy v rámci

kvality komprese. Zde jsou předpokládány vzniky artefaktů na obloukové klenbě, zatímco podpěry jsou vzhledem k rovným tvarům snadno komprimovatelné. Data obsahují velké množství prázdného prostoru a mají homogenní vnitřní strukturu.

Model *bloku motoru* o rozměrech 256x256x256 byl vybrán vzhledem k jeho využití v testech vektorové kvantizace v práci [12], kde je možno vidět i kompresi s využitím DXT algoritmu. Datová množina obsahuje velké množství prázdného místa, model nezabírá celý prostor.

Jako největší dostupný datový set byl zvolen model *karoserie vozu Porsche 911* s rozlišením 560x1024x348. Jedná se o největší dohledatelný model obsahující reálná data, nikoliv výstup matematické simulace. Model zabírá prakticky celý prostor, uložena je i vnitřní struktura vozu. Nevýhodou dat je zanesený šum, který způsobuje nepřesnosti vizualizace.

Model lidské hlavy, *ctmayo* o rozlišení 128x128x128 byl použit vzhledem k různorodé vnitřní struktuře. Hlavně u metody založených na extrakci povrchů jsou viditelné změny mezi různě nastavenými prahy. V případě sledování paprsku a vizualizace všech vrstev v jednom obrázku vzniká problém se splýváním hodnot a je potřeba využít přenosové funkce k rozdělení a lepší vizualizaci. Vzhledem k detailní neznalosti dat a použitelnosti v praktické vizualizaci nebylo toto testováno.

Byl testován také model vypočteného 3D fraktálu, *stent8*, o rozlišení 512x512x175. Vzhledem k pouze povrchové struktuře dat a nulové vnitřní stavbě byl tento model testován pouze v extrakčním algoritmu Marching Cubes. Model byl vygenerován v bezplatné verzi programu Incendia.

7.4.2 Měření

V první části testů jsou data rozdělena na jednotlivé bloky a porovnává se, zda a o kolik je rychlejší vykreslovat data uložená přímo v grafické paměti s daty, která se posílají na grafickou kartu průběžně. Tato verze testů není obsažena ve finálním programu (viz. Příloha C).

V tabulce 5 jsou ukázány výsledky provedených testů. Veškeré testy vykreslovaly do textury s rozlišením 640x480. Tyto testy mají za cíl pouze ukázat rychlost sběrnice, resp. poklesu rychlosti vykreslování, pokud jsou data na kartu posílána průběžně. Porovnány jsou metody Marching Cubes a Ray-Casting. V případě Marchnig Cubes bylo testováno také vykreslování čistě na CPU bez použití grafické karty. Dle předpokladů je tento přístup pro aplikace reálného času nepoužitelný. V případě zasílání dat v každém snímku jsou poklesy rychlostí značné. Pro vykreslování v reálném čase tak není příliš vhodné data rozdělit na jednotlivé bloky a ty pak zpracovávat samostatně. Pokud bychom potřebovali na grafickou kartu nahrát data, která přesahují dostupnou paměť, je vhodnější použít komprese. V případě, že ani komprese nesplní požadovaný efekt, pak nezbývá nic jiného, než využít postupného nahrávání dat na grafickou kartu, s čímž je spojen úbytek výkonu. Jako ve všech případech, i zde musíme brát opět v úvahu cíl a styl použití dat.

Model	Rozlišení dat	MC (C, 1)	MC (C, 2)	MC (CPU)	3D textura (DC, 1)	3D textura (DC, 2)
Syn64	64x64x64	550	150	4	880	350
ctmayo	128x128x128	105	80	< 1	660	83
motor	256x256x256	10	< 1	< 1	490	12
stent8	512x512x175	< 1	< 1	< 1	320	5

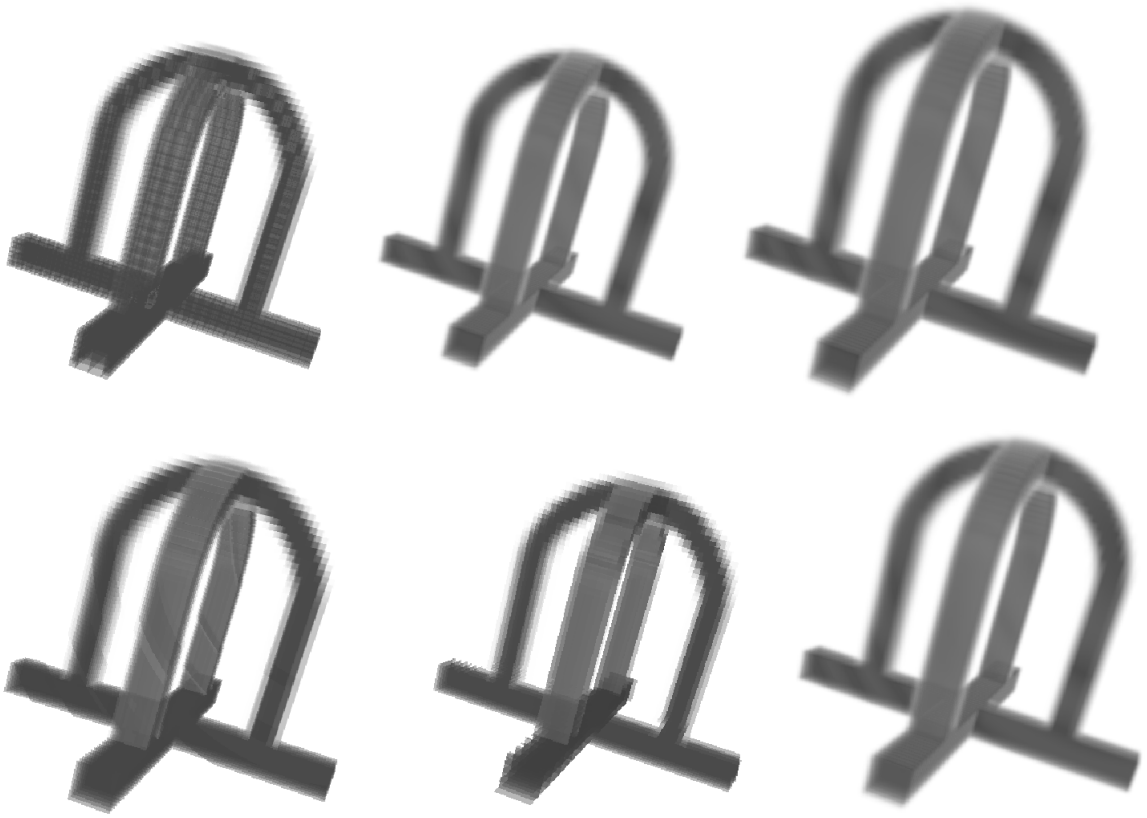
Tabulka 5: Výsledky testů - MC = Marching Cubes; 3D textura = klasické sledování paprsku (ray-casting); DC = DirectCompute, C = CUDA, CPU = algoritmus spuštěn přímo na CPU; 1 - data naplněna na GPU pouze při inicializaci, 2 - data posílána na GPU v každém snímku

Druhá část testů je zaměřena na vlastní vykreslování. Všechny dostupné hodnoty jsou naměřeny v programu, který je součástí této práce (viz. Příloha C). Aplikace pro vykreslování od třetích byly také testovány, ovšem zde dochází k problému s interpretací výsledků. Většina programů má uzamknutou snímkovou frekvenci na 60 FPS. Pro žádná data se nepodařilo klesnout pod tuto hodnotu. Dalším problémem je to, že většina programů data nejdříve extrahuje do raw dat a teprve potom spouští vlastní vykreslování. U profesionálních programů zase není k dispozici informace o použitém postupu vykreslování.

Z testovaných programů byl nejzajímavější program Voreen (Volumetric Rendering Engine) [36]. Má velmi pěkně řešené uživatelské rozhraní, snadnou obsluhu a jsou dostupné informace o použitých algoritmech plus zdrojové kódy. Program je stále ve stádiu vývoje, stabilita tak v některých případech nepatří k silným stránkám a práce končí občas restartem celého počítače. Program je velmi rozsáhlý a prostudovat podrobně jeho zdrojové kódy by zabralo ohromné množství času. Bez této znalosti je ovšem prakticky nemožné použít program pro porovnávání výkonu. Z možností načtených objemových dat jsou podporována raw data a DICOM snímky. Bohužel žádné datové sety ve formátu DICOM nebyly k dispozici, aby mohla být tato funkcionalita otestována. Vzhledem k tomu, že se tato práce zaměřuje spíše na aspekt vykreslování dat s určitým typem komprese, bylo by porovnání naměřených údajů s programem Voreen nepoužitelné k vyvození závěrů o efektivitě.

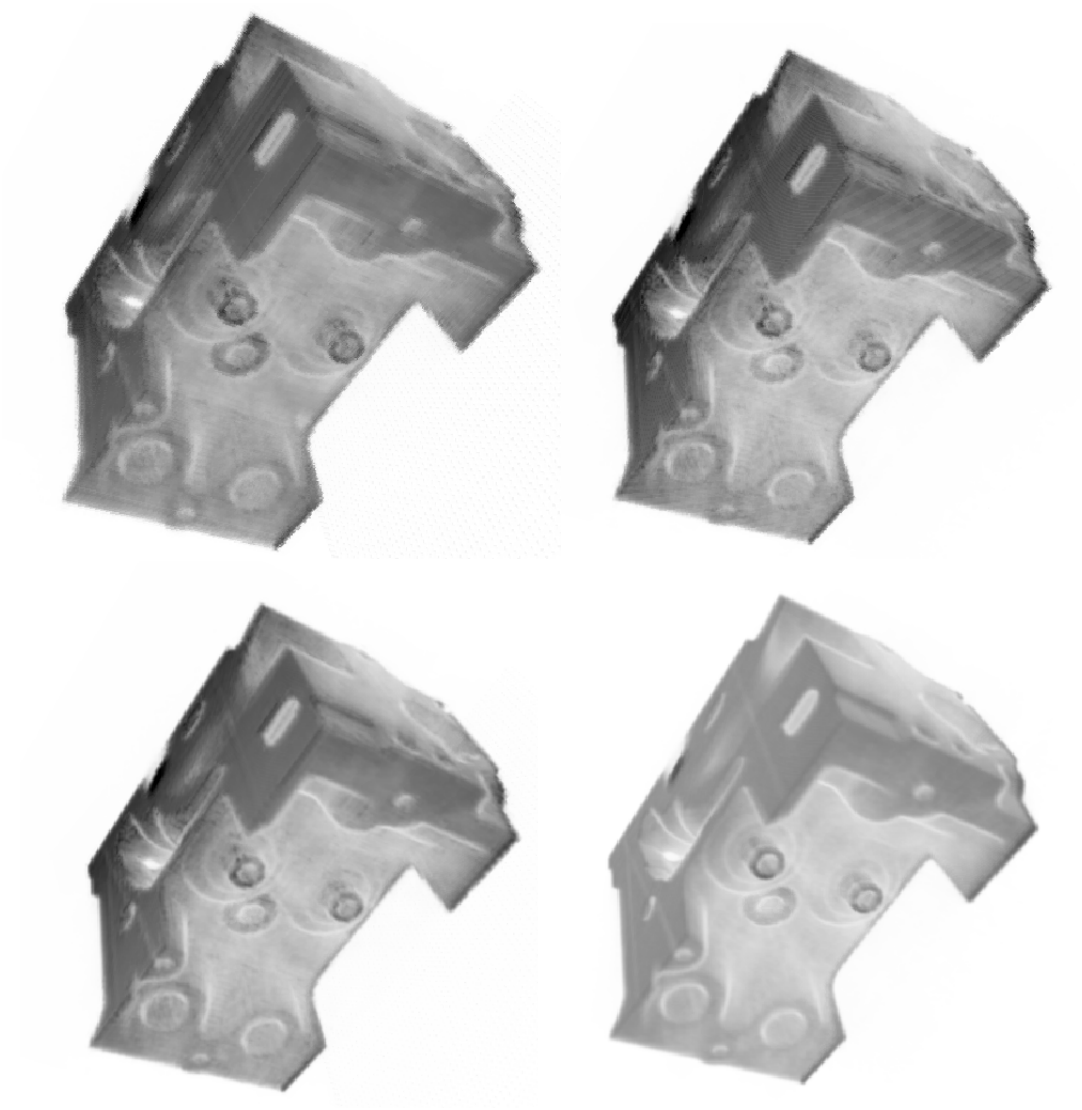
Na obrázcích 20 a 21 jsou porovnány jednotlivé metody vykreslování v případě komprimovaných a nekomprimovaných dat. Naměřené hodnoty snímkové frekvence jsou pak shrnuty v přehledné tabulce 6.

V prvním testu se pracovalo s modelem *Syn64*. Obrázek 20 ukazuje rozdíly mezi jednotlivými způsoby vykreslování dat.



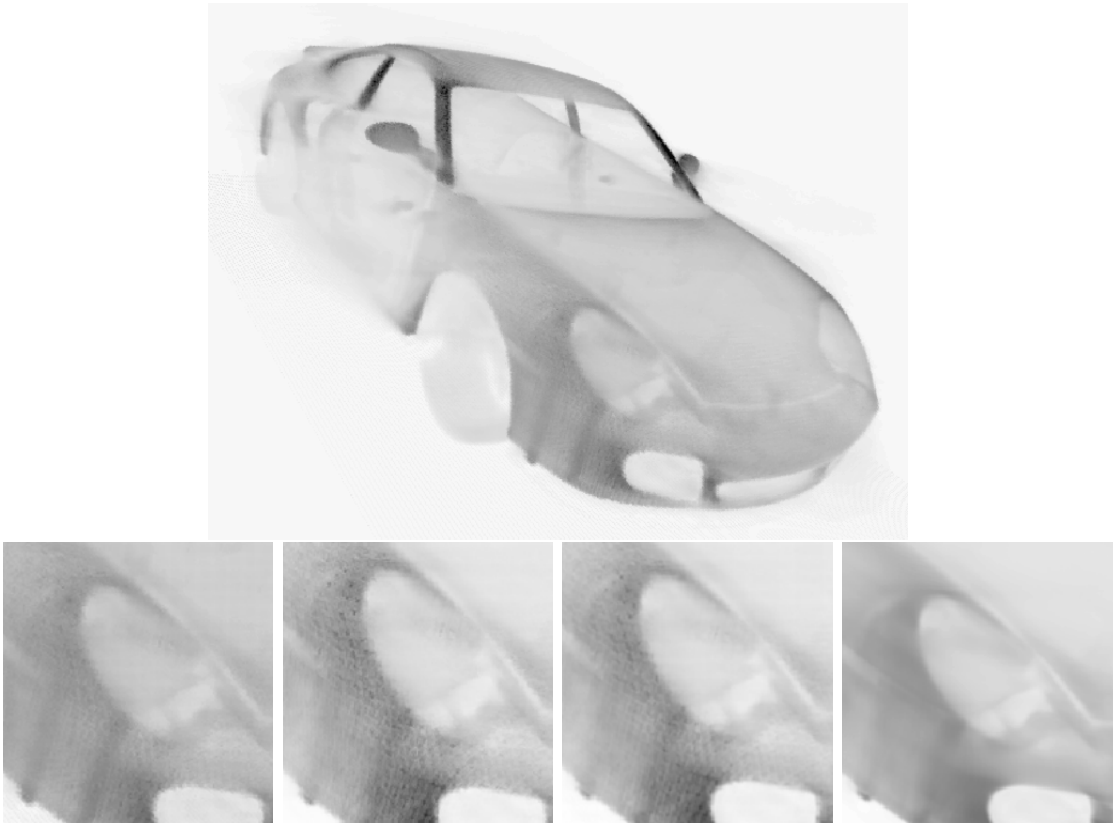
Obrázek 20: Porovnání jednotlivých metod vykreslování, komprese a použitého API. Jednotlivé obrázky zleva doprava: 1) RAW data, CUDA, nearest neighbour filtrace, octree traversace 2) RAW data, trilinear filtrace, raycasting 3) Compute Shader, trilinear filtrace, raycasting 4) L3VQ, nearest neighbour filtrace, raycasting 5) L3VQ, nearest neighbour filtrace, Bressenham 6) L3VQ, trilinear filtrace, raycasting

Obrázek 21 ukazuje rozdíly v kvalitě v případě použití komprese L3VQ a různých typů vykreslování. Použitý model je v tomto případě větší než u předchozího obrázku. Jedná se o model *bloku motoru*. Poslední obrázek v sérii vyobrazuje originální data pro možnost porovnání. Dle očekávání jsou první dva obrázky, vykreslené pomocí nearest neighbour filtrace, kvalitativně velmi podobné. Třetí obrázek je trilineární filtrace nad komprimovanými daty. V aktuálním natočením a přiblížení modelu jsou rozdíly mezi jednotlivými snímky pozorovatelné, nicméně podle typu použití taktéž zanedbatelné. Jistě, pokud bychom hledali na snímku motoru nerovnosti povrchu nebo praskliny, nemůžeme L3VQ kompresi použít. Naopak pro multimediální je výsledek ve většině případů dostatečný. Bude-li model umístěn v rámci virtuálního světa, ve kterém se bude pohybovat pozorovatel, jsou chyby na rozdíl od obrázku 20 prakticky nepostřehnutelné.



Obrázek 21: Porovnání jednotlivých metod vykreslování, komprese a použitého API. Jednotlivé obrázky zleva doprava: 1) L3VQ, nearest neighbour filtrace, Bressenham 2) L3VQ, nearest neighbour filtrace, raycasting 3) L3VQ, trilinear filtrace, raycasting 4) RAW data, trilinear filtrace, raycasting

Kromě dvou výše vyobrazených modelů jsou testy aplikovány také na model *karoserie vozu Porsche 911*. Pro představu modelu je uvedena vizualizace na obrázku 22. Vzhledem k velikosti dat nejsou rozdíly mezi metodami na pořízených snímcích příliš viditelné. Pro jejich vyobrazení je potřeba prohlédnout pouze výřez ze snímku.



Obrázek 22: Model karoserie vozu Porsche 911 a jednotlivých výřezů předního světlometu. Nepřesnosti v modelu způsobeny nepřesnostmi v datovém setu. Zleva 1) L3VQ, nearest neighbour filtrace, Bressenham 2) L3VQ, nearest neighbour filtrace, ray-casting 3) L3VQ, trilineární filtrace, ray-casting 4) RAW data, trilinear filtrace, raycasting

Na těchto třech modelech byla provedena různá porovnání ve vykreslování, pokud jsou nastavené různé parametry. Tyto parametry jsou shrnuty v první části tabulky 6. Všechny testy jsou měřeny pod výpočetními shadery. Výsledek měření je uveden v druhé části tabulky 6.

Metoda	Popis
Metoda #1	RAW data, trilineární filtrace, raycasting, RWStructuredBuffer
Metoda #2	RAW data, trilineární filtrace, raycasting, 3D textura
Metoda #3	RAW data, nearest neighbour filtrace, raycasting, RWStructuredBuffer
Metoda #4	RAW data, nearest neighbour filtrace, raycasting, 3D textura
Metoda #5	L3VQ, trilinerání filtrace, raycasting, RWStructuredBuffer
Metoda #6	L3VQ, nearest neighbour filtrace, raycasting, RWStructuredBuffer
Metoda #7	L3VQ, nearest neighbour filtrace, Bresenham, RWStructuredBuffer

Model	#1	#2	#3	#4	#5	#6	#7
Syn64	480	880	800	890	145	620	660
Motor	20	290	130	330	25	160	125
Karoserie Porsche	<1	35	<1	65	7	47	43

Tabulka 6: Jednotlivé provedené testy s výpočetními shadery a naměřené hodnoty snímkové frekvence. Číslo sloupce určuje použitou metodu. Naměřené hodnoty udávají snímkovou frekvenci.

Testy #2 a #4 byly provedeny s daty uloženými ve 3D textuře. Ta přináší větší výkon v případě čtení. U textur využívají grafické paměti cache, kdy načtou konkrétní texel (voxel) a jeho okolí v rámci jednoho čtení. V případě, že následující data čteme v rámci tohoto okolí, dojde k jejich čtení z cache paměti. Její rychlost se pohybuje zhruba na úrovní sdílené paměti ve skupině vláken. Na druhou stranu, pokud bychom do textury přistupovali skutečně náhodně a cache by tím pádem nebyla prakticky využita, došlo by naopak k poklesu výkonu. V případě ray-castingu k výpadků často nedochází, jelikož paprsek postupně prochází jeden voxel za druhým. Pro aktuální hodnotu v dalším kroku přečte hodnotu jeho souseda. Samozřejmě za předpokladu, že máme krok nastavený na hodnotu, kdy se pohybujeme po sousedech. Pokud krok nastavíme příliš veliký, budeme číst pouze každý n-tý voxel, který leží v dráze paprsku, a pak již k výpadkům docházet může.

Jak je vidět z hodnot v tabulce 6, rozdíl mezi použitou trilineární a nearest neighbour filtrací je ovlivněn rozlišením textury. Pro malá data prakticky žádný rozdíl pozorovatelný není, pro větší se pak postupně zvyšuje. Stále se ovšem nejedná o diametrálně rozdílné výsledky jako v případě testů #1 a #3. To je způsobeno právě využitím cache, kdy není potřeba hodnoty načítat opakovaně z globální paměti.

V testech #1 a #3 jsou použita stejná nastavení, pouze 3D textura byla nahrazena za strukturu RWStructuredBuffer. Data jsou linearizována a přepočet na 3D index se provádí ručně. Tato data jsou uložena v globální paměti. Během čtení ovšem není využita žádná automatická cache. V případě nearest neighbour filtrace se čte pouze jeden vzorek na jeden voxel. V případě trilineární už ovšem načítáme osm hodnot z kterých počítáme filtraci. Zde je rozdíl mezi nastaveními značný a dobře pozorovatelný z výsledků. V případě největšího modelu, karoserie Porsche, jsou již výsledky pod hranicí jednoho snímku.

Testy #5 a #6 pracují s daty komprimovanými pomocí algoritmu L3VQ. Zde není možno data ukládat do textury a využít tak výhod cache. Během vykreslování se totiž

provádí dekomprese dat a v případě použití textury dochází k interpolacím hodnot, což není žádoucí. Existuje možnost načítat data z textury bez interpolací a filtrů, ovšem nikde nelze dohledat, zda tento přístup využívá nebo nevyužívá cache. Při porovnání tohoto přístupu a použití bufferu byl rozdíl zanedbatelný, takže lze usuzovat, že se cache nejspíše nepoužívá.

V případě testu #5 není využita cache, která by v tomto případě šla snadno doprogramovat podle postupu popsaného v oddílu 7.2.4. Pro zachování integrity a porovnání s testem #1 jsem toto vylepšení nezahrnul.

Poslední test #7 je taktéž pro data komprimovaná pomocí L3VQ, ale nevyužívá algoritmus pro klasický ray-casting, uvedený v kapitole 6. Jeho místo převzal Bresenhamův algoritmus. V testech se výkonnostně podobal klasickému ray-castingu, ovšem vizuální výsledek byl horší. Na statickém obrázku tento efekt pozorovatelný není, ovšem v pohybu celý model problikával. Tento jev je způsobený tím, že algoritmus pracuje s celými čísly. Podmínka, který voxel se má číst, pracuje s porovnáním na polovinu. V případě, že v jednom snímku paprsek projde nad ní a v druhém pod ní, čte se pokaždé jiná hodnota. Pokud se model pohybuje, tak se toto stává prakticky stále.

Kromě testů uvedených v tabulce 6 byla dále zkoušena rychlost vykreslování s použitím rozhraní CUDA. Výsledky uvedených měření jsou v tabulce 7. Tato měření jsou uvedena pouze pro hrubé rychlé porovnání.

Model	Rozlišení	3D textura	Oktalový strom
Syn64	64x64x64	210	80
Motor	256x256x256	175	6
ctmayo	128x128x128	170	25

Tabulka 7: Testy s využitím rozhraní CUDA. Naměřené hodnoty udávají snímkovou frekvenci.

V porovnání s hodnotami naměřenými v tabulce 6 se v případě rozhraní CUDA jedná o znatelně nižší hodnoty snímkové frekvence. Z tohoto důvodu jsem se dále již rozhraní CUDA nevěnoval. Použití oktalového stromu u nepřineslo rychlostní výhodu a v případě vizualizace vnitřní struktury dat ani nezmenšilo datovou reprezentaci. Tento efekt byl očekáván a diskutován v kapitole 4. Dále nebyla oktalovým stromům věnována pozornost.

7.5 Zhodnocení

Zobrazení statických dat je základem všech dalších vizualizačních metod. Hlavní omezující faktor zde představuje rozlišení dat. Je potřeba přijít s určitým typem komprese dat na úrovni statického snímku. Tato komprese musí ovšem podporovat náhodný přístup k datům, abychom byli schopni data v reálném čase dekomprimovat a zobrazovat.

Z testovaných algoritmů jsou nejjednodušší komprese DXT, které je přímo podporovány grafickým rozhraním. Tyto komprese mají ale řadu nedostatků, jako např. malý kompresní poměr a značnou ztrátovost dat.

Lepší výsledky lze dosáhnout s metodami založenými na vektorové kvantizaci. Většina dostupných metod vychází z komprese L3VQ uvedené v článku [32]. Tato kompresní

metoda nabízí dobré kompresní poměry (teoreticky až 20 : 1) a rychlou dekompresi v reálném čase. Pro vědecká data má metoda problém se ztrátovostí s zašuměním dat.

Nejhůře v testech dopadl oktalový strom. Tento způsob komprese není vhodný pro následné vizualizace průsvitných dat. V jejich případě dojde k opačnému efektu a data jsou místo zmenšení velikosti naopak zvětšena. To je způsobeno potřebou uložit datovou strukturu stromu včetně vnitřních uzlů, které zabírají většinu prostoru. Kromě vlastního nárůstu velikosti se zvýší i potřebný čas dekomprese, což je způsobeno procházením stromu a velkým počtem čtení z pomalé globální paměti.

Metoda Marching Cubes není určena pro poloprůhledná data. Zde dochází pouze k extrakci povrchu z dat na základě uživatelského kritéria. Výhodou metody je její snadná paralelizace, a proto je tento algoritmus vhodný pro běh na grafické kartě. Narozdíl od oktalového stromu není potřeba extrahovaný povrch přepočítávat v každém snímku, ale pouze při změně dat. Metoda oktalového stromu a Marching Cubes lze snadno zkombinovat a zahrnout tak výhody datové komprese, což přináší další zvýšení výkonu.

Vlastní vizualizace byla testována pod rozhraním CUDA a výpočetními shadery. V případě cílení metod na vizualizace poskytují vyšší výkon výpočetní shadery, které jsou přímo svázány s grafickým rozhraním (v našem případě DirectX 11). V případě použití rozhraní CUDA byl vykreslovací výkon omezen. Vzhledem k uzavřeným zdrojovým kódům nelze přesně určit důvod omezení výkonu.

Pro zobrazování dat byl testován Bresenhamův algoritmus, ale vzhledem k jeho způsobu průchodu daty způsoboval problikávání modelu. Na základě tohoto pozorování byl ke všem vizualizacím použit klasický algoritmu ray-castingu. Rozdíl mezi výkonem těchto dvou algoritmů byl velmi malý, obrazová kvalita byla ovšem v případě ray-castingu mnohonásobně vyšší.

Pracujeme-li s algoritmy přímo na grafické kartě, uvažujeme jako velkou výhodu možnost paralelního běhu. Je nutno uvážit, zda má cenu za každou cenu paralelizovat algoritmy a snažit se je prosadit v GPGPU podobě jen proto, že je to v současné době módní. Paralelizace s sebou přináší také negativní hlediska v podobě zvýšené režie. Data musíme přenášet mezi CPU a GPU. Vlastní algoritmus může přinést oproti klasické verzi urychlení, ovšem se započítáním časů přenosů a režie se dostaneme na stejné, ne-li horší časy.

8 Časově proměnná data

V minulé kapitole 7 jsme se zaměřili na jednotlivé snímky, nyní budeme uvažovat časově proměnná data. Tento pojem v praktickém pojetí představuje vlastně obyčejnou videosekvenci. V rámci ní můžeme pracovat s různými typy dat a jejich různým obsahem, přes opravdu klasické animace typu poskakující míček, až po fyzikální simulace proudění v kapalinách.

Stejně jako u statických dat, tak i zde narážíme na problém velikosti dat. Bohužel v tomto případě je problém daleko závažnější. Pokud by se do grafické paměti vešel jeden celý snímek, v případě animace další snímky do paměti uložit již nelze. Komprese je zde zásadní pro veškeré vizualizace. Vlastní vizualizační algoritmus pracuje stejným způsobem, jako by se jednalo o statická data.

V této kapitole jsou nejdříve shrnuty některé existující metody, následně je diskutován navržený přístup. Ten je podrobně rozebrán a otestován v další části kapitoly na uvedených datových množinách. V závěrečném zhodnocení jsou shrnuty jednotlivé aspekty navrženého přístupu a jeho efektivita.

8.1 Některé dostupné metody

V současné době lze nalézt mnoho algoritmů zabývajících se časově proměnnými objemovými daty. Většina z těchto postupů kombinuje, podobně jako u klasických videí, přístupy získané ze statických dat s algoritmy sloužícími pro animace.

S jedním z možných algoritmů přišel Fout v [8]. Ten využívá různé metody kvantizace, tudíž se jedná o ztrátový algoritmus. Výhodou, plynoucí z využití kvantizace, je rychlá dekomprese dat. Jako v případě jiných algoritmů využívajících kvantizace, i zde jsou data dělena na bloky. Uvnitř jednoho bloku je pak uloženo více kvantizovaných časových kroků.

Využít lze také komprimační algoritmy založené na podobném principu jako MPEG video, kdy využíváme I snímky a P snímky[4]. Tato metoda je rozebírána v [14]. Jednotlivé snímky jsou komprimovány pomocí 3D waveletové transformace. Postupně vzniká oktalový strom. Pro P snímky porovnáváme uzly oktalového stromu oproti předchozímu I snímku a kódujeme rozdíl v těchto uzlech. Jednotlivé uzly jsou obdobou makrobloků z video komprese MPEG.

Podobná metoda, založená opět na I, P snímcích a waveletech je také rozebírána v [34]. Zde se využívá vypouštění bloků, které mají "malý přínos" na změnu dat. Autor se ovšem nijak nevěnuje rozebrání problému, co znamená "malý přínos" a o jaké hodnoty se řádově jedná. Jako sekundární komprese je následně použit algoritmus gzip (pracující s LZ kompresemi).

Kombinace využití waveletů a kvantizace je využívána v [9]. Tato metoda byla již zmíněna pro komprese statických snímků, v článku je také následně diskutováno využití pro animace. Zde se opět uplatní komprese MPEG, kde se ovšem využívá plná trojice

[4] I snímky jsou uloženy jako celá data a lze je přímo zobrazit. P snímky jsou závislé na I snímcích a obsahují rozdílová data a pohybové vektory. Pro jejich zobrazení je potřeba jejich rekonstrukce na základě předchozího I snímku.

kódovaných snímků. K dvojici I a P je přidán ještě B snímek. Ten pracuje s minulými i následujícími I snímky.

Algoritmy založené na principu MPEG video kompresí lze využít také v kombinaci s vektorovou kvantizací ([12]) a jinými metodami, kde nejsme pevně svázáni s využitím pouze waveletových transformací. Možností je například [19]. Metoda kombinuje klasické slovníkové komprese s JPEG kompresí. Autor udává jako důvod složitost dekompresního algoritmu pro waveletové transformace.

Slovníkové algoritmy můžeme nalézt také v metodě [26]. Zde jsou kombinovány s algoritmy pro komprese textur založené na DXT. Na straně CPU se provádí při vykreslování dekomprese LZO, zatímco na GPU se dekomprimuje následně textura.

Pro vědecká data je vhodné uvažovat také bezeztrátové komprese. Až doposud se jednalo o metody založené na určité ztrátě informace. V článku [22] lze nalézt bezeztrátovou kompresi založenou na LZ algoritmech. Data jsou rozdělena na jednotlivé bloky a tyto bloky jsou následně komprimovány pomocí LZ komprese. Dekomprese se provádí na CPU a na grafickou kartu jsou pak posílána dekomprimovaná data. Urychlení je zde v případě shody po sobě jdoucích bloků, kdy není třeba blok z dalšího snímku na kartu posílat znovu, ale využije se již existující blok z aktuálně zpracovaného snímku.

8.2 Navržený přístup ke kompresi

V navrženém postupu se vycházelo z klasických kompresních algoritmů uvedených v kapitole 4. Tyto algoritmy jsou bezeztrátové a univerzální. Z jejich univerzálnosti vycházejí i jejich limity. Nedosahují kompresních poměrů specializovaných algoritmů pro konkrétní typ dat. Vzhledem ke snaze o univerzální využití algoritmů jak pro vědecká, tak pro data získaná např. ze 3D scannerů, kde jsou očekávány převážně povrchy a velké prázdné množiny, jsou zvoleny právě tyto komprese. Cílem navrženého přístupu bylo otestovat jiný pohled na problematiku, než jaký nabízí většina dostupných řešení. V těch vychází sekundární komprese z časového průběhu dat, což ovšem neumožňuje plně náhodný přístup k datům.

Jako první testovací krok ve fázi návrhu použitého přístupu, otestovány komerční implementace WinZip, WinRar pro prostudování efektivity komprese. Kompresní poměry jsou závislé na typu dat. Pro data s velkou hodnotou entropie jsou tyto algoritmy nevhodné. Na testovaných souborech s průměrnou hodnotou entropie 4 bity / byte bylo dosaženo kompresních poměrů přes 50% a bylo uznáno za vhodné se univerzálními bezeztrátovými algoritmy dále zabývat, přenést a otestovat jejich implementaci na GPU. Je nutné si uvědomit, že tento přístup není univerzální a míra komprese záleží na vstupních datech.

Použití těchto algoritmů pro volumetrická data bylo již v několika článcích diskutováno (např. [19], [22]). V žádném z prostudovaných článků nebyla posuzována možnost spouštět tyto algoritmy přímo na grafické kartě. Vzhledem ke zmíněné bezeztrátovosti je lze použít pro kompresi na druhé úrovni. K tomuto účelu byly algoritmy využity i ve zmíněných článcích. Jednotlivé snímky můžeme komprimovat libovolným algoritmem, popř. je ponechat ve své původní raw podobě. Na takto uložené snímky lze aplikovat univerzální bezeztrátový komprimační algoritmus. Tyto přístupy jsou známé již z MPEG kompresí, kde každý I snímek je komprimován určitým algoritmem a druhý

stupeň komprese je zaveden v čase, mezi snímky.

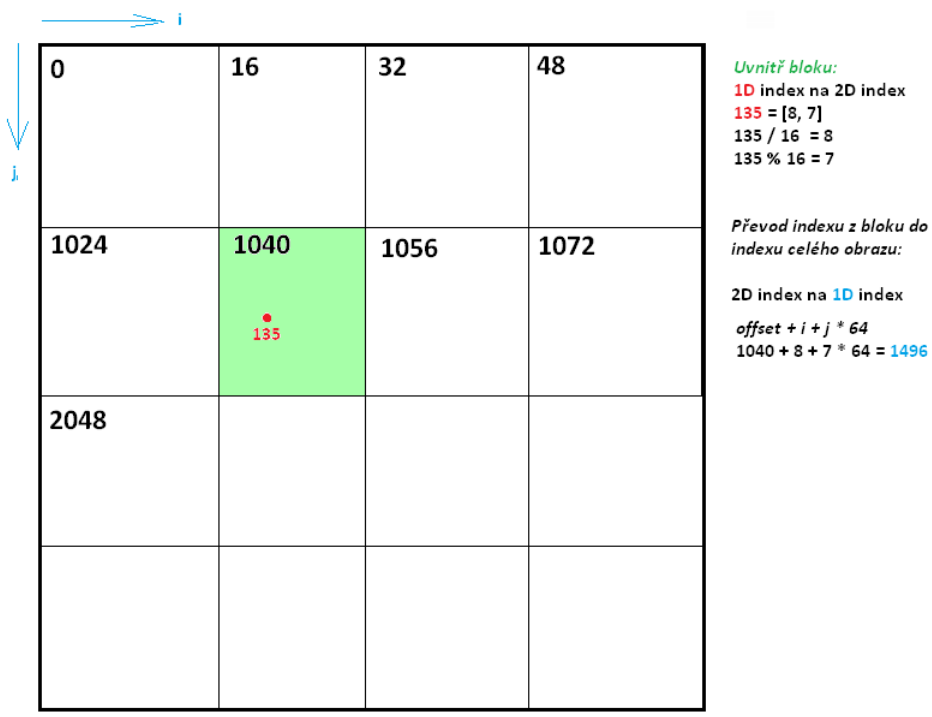
8.2.1 Přístup k datům

Použité univerzální kompresní algoritmy jsou ve většině případů sériové, což v případě zapojení GPU není žádoucí. Algoritmy lze částečně do paralelní podoby upravit. Data se rozdělí na jednotlivé celky, které se následně komprimují a dekomprimují samostatně, každý ve vlastním vláknu. Tyto celky mohou být poskládány různým způsobem. Pro základní otestování vhodnosti algoritmu pro paralelizaci na GPU bylo využito pouze klasického lineárního přístupu.

Zde byly zvoleny dva základní přístupy. První pracoval s daty na lineární úrovni. Na data bylo nahlíženo jako na souvislý blok v paměti. V případě dat, kde očekáváme podobné hodnoty v určitém okolí, není tento přístup plně vhodný, nicméně jako základní testovací metodika výkonu postačuje.

Druhý přístup rozděluje data na jednotlivé bloky a komprimuje každý blok samostatně. V rámci jednoho bloku je opět použit lineární průchod daty. Tato změna reprezentace má vliv pouze na slovníkové algoritmy, pro statistické nedochází z globálního pohledu k žádné změně.

Vzhledem k lineárnímu přístupu je třeba přepočítat index z rozsahu bloku na index v rozsahu nekomprimovaných dat. Tím určíme pozici, kam se zapíší dekomprimované hodnoty. Na obrázku 23 je ukázán výpočet pro vzorovou hodnotu. Pro větší názornost je vyobrazena pouze 2D verze.

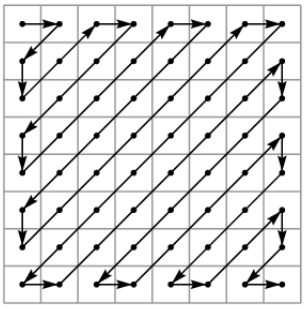


Obrázek 23: Rozložení dat na bloky a přepočet indexů. Jednotlivé bloky mají rozměr 16x16. Celkový obrázek má rozměr 64x64. Čísla v levém horním rohu každého bloku určují offset počátečního indexu tohoto bloku od začátku dat.

U blokového přístupu lze dále velmi snadno pracovat s daty podobným způsobem, jako je uveden v [22]. Při kompresi dat ukládáme pouze změněné bloky. Při dekompresi pak bloky, které zůstaly od předchozího snímku neměnné, ponecháme ve své původní podobě, zatímco ostatní bloky nahradíme jejich dekomprimovanými aktuálními verzemi. V tomto případě ovšem ztrácíme možnost náhodného přístupu k libovolnému snímku a animace musí běžet kontinuálně, nelze přeskakovat jednotlivé snímky bez nutnosti rozbalit posloupnost od posledního tzv. I snímku.

8.2.2 Možné vylepšení

Lepších kompresních poměrů by mohlo být dosaženo s využitím blokových schémat, kde by jednotlivé bloky mohly být procházeny metodou založenou na zig-zag průchodu. Ukázka průchodu daty je na obrázku 24. V případě nelineárního přístupu bychom ovšem museli brát v úvahu úbytek výkonu spojený s nelineárními přístupy do paměti v případě dekomprese, kde je čas kritický. Nelineární přístup by nám deaktivoval používání cache paměti pro zápisy. Jako první myšlenka byla proto otestovat rychlosti v případě lineárního přístupu a v případě časů umožňující další zatížení algoritmu využít i další možné postupy. Zde je třeba si ovšem uvědomit, že zig-zag přístup nezlepší kompresi nad míru entropie dat. Pokud se tedy lineární přístup této hodnotě přiblíží, je zbytečné uvažovat změnu průchodu daty.



Obrázek 24: Ukázka průchodu dat metodou zig-zag

8.2.3 Shrnutí

Kromě vlastního využití v uložení snímků animace se uplatnění pro tyto algoritmy nalezne i při nahrávání obecných dat na grafickou kartu pro využití v GPGPU výpočtech. Je rychlejší přenést menší množství dat a ty následně rozbalit, než data rozbalit na CPU a pak teprve inicializovat přenos. V případě časově kritických operací je rozdíl zásadní a umožňuje provádět některé operace v reálném čase.

Vzhledem k omezeným možnostem grafických čipů je potřeba pracovat s jednoduchými algoritmy. Z tohoto důvodu byly zvoleny jednoduché statistické a slovníkové LZ metody. V závěru části kapitoly 5 věnované bezeztrátovým kompresím byly zmíněny také další možné algoritmy, jejich použití v rámci grafických karet ovšem není v rámci reálného času prakticky možné. Jejich dekomprimační algoritmy pracují se stromy, dynamicky alokovanými strukturami a využívají extenzivně paměť. Všechny tyto operace nejsou pro grafické karty vhodné, přihlédneme-li navíc k sériové povaze algoritmů, vyjdou nám jako nevyhovující.

8.3 Návrhy algoritmů

Ve všech uvedených algoritmech se využívají definované konstanty. Pro větší přehlednost je uveden jejich seznam a hodnoty v tabulce 8. Hodnota 0xFFFFFFFF odpovídá maximálnímu zaplnění 32bitového celého čísla.

Konstanta	Hodnota
SEARCH_BUFFER_SIZE	511
LOOK_AHEAD_BUFFER_SIZE	127
SEARCH_BIT_SIZE	9
LOOK_AHEAD_BIT_SIZE	7
CODE_INFO_SIZE	16
CODE_INFO_MASK	(0xFFFFFFFF >> (32 - CODE_INFO_SIZE))
POS_MASK	(0xFFFFFFFF >> (32 - SEARCH_BIT_SIZE))
LENGTH_MASK	(0xFFFFFFFF >> (32 - LOOK_AHEAD_BIT_SIZE))
ZNAK_MASK	0xFF
CODED_MASK	0x1
BITS_IN_BYTE	8
BITS_IN_UINT	32

Tabulka 8: Tabulka používaných konstant a jejich hodnot

8.3.1 LZ77

Algoritmus LZ77 byl zvolen jako základní verze vzhledem k jednoduchosti jeho dekomprese. Kompresní algoritmus není nijak paralelizován, je použita jeho sériová verze běžící čistě na CPU. Zde není potřeba zabývat se příliš optimalizací komprese, jelikož kompresi nad daty provádíme ve většině případů pouze jednou a následně nás zajímá během vykreslování hlavně čas dekomprese. V současné době jsou grafické karty dostatečně výkonné, aby zvládaly dekompresi dat v reálném čase. Otázkou pouze zůstává hraniční velikost dat, kdy se dostaneme na limit reálného času. Výpočetní výkon grafické karty není ovšem limitující. Limity představují operace s pamětí, které jsou potřeba v případě slovníkových algoritmů ve značném množství.

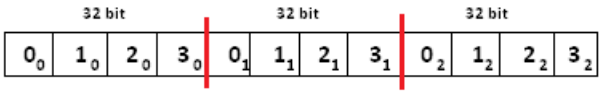
Vlastní dekomprese začíná u datové reprezentace komprimovaných dat. Každé slovo je kódováno stejným způsobem, jako samostatná trojice. Pro připomenutí je uvedena již jednou zmíněná ukázka komprese vstupních dat.

```
Vstup: abracadabra
Výstup: [0,0,a][0,0,b][0,0,r][3,1,c][2,1,a][7,3,a]
```

Každý zakódovaný znak je tím pádem trojice [pozice, délka, následující znak]. Následující znak je vždycky kódován pomocí 8bitů. Pro zbylé dvě hodnoty je bitová velikost klíčová, neboť určuje rozsahy slovníku. Problém je, že GPU v rámci výpočetních shaderů pracuje pouze s celočíselným datovým typem o velikosti 32bit. Zakódujeme-li jednu trojici jako 32bitovou hodnotu, získáme na slovníky velikost 24bit. To je bohužel příliš velký rozsah a ve většině případů také zbytečný. Rozdíl v komprimované velikosti dat při použití slovníků o velikostech 2048 vs. 511 bytů je v řádu jednotek nebo desetin procent.

Použijeme-li jakýkoliv nižší rozsah, je třeba upravit načítání s využitím bitových operací. Nejlépe se pracuje s násobky 8bitů, proto bylo na velikost slovníku vyhrazeno 16bitů. To nám umožňuje vytvořit slovníky o velikostech 511 (9 bitů) a 127 (7 bitů) znaků.

Na obrázku 25 je vyobrazena reprezentace dat uložení v paměti na grafické kartě a jejich načítání pro operace dekomprese v případě uložení trojice na 24bitech.



Krok	Last	Act	Val	Výpočet Val
1	$0_0 1_0 2_0 2_0$	$0_0 1_0 2_0 3_0$	$0_0 1_0 2_0 \vert 3_0$	Last
2	$0_0 1_0 2_0 2_0$	$0_1 1_1 2_1 3_1$	$3_0 0_1 1_1 \vert 2_1$	Last << 3 | Act >> 1
3	$0_1 1_1 2_1 2_1$	$0_2 1_2 2_2 3_2$	$2_1 3_1 0_2 \vert 1_2$	Last << 2 | Act >> 2
4	$0_2 1_2 2_2 2_2$	$0_3 1_3 2_3 3_3$	$1_2 2_2 3_2 \vert 0_3$	Last << 1 | Act >> 3
5	$0_3 1_3 2_3 2_3$	$0_3 1_3 2_3 3_3$	$0_3 1_3 2_3 \vert 3_3$	Last
6	$0_3 1_3 2_3 2_3$	$0_4 1_4 2_4 3_4$	$3_4 0_4 0_4 \vert 2_4$	Last << 3 | Act >> 1

Obrázek 25: Buffer pro načítání hodnot v rámci výpočetního shaderu při dekompresi LZ77

Jednotlivá čísla určují pozici v rámci 32bitového čísla, jejich spodní indexy pak určují globální pozici s ohledem na uložení v paměti. Tím vznikne jednoznačné určení pro každou hodnotu. Proměnná *Last* udržuje poslední načtenou hodnotu, *Act* je pak načtená v aktuální iteraci. Proměnná *Val* obsahuje požadovaný 24bitový buffer pro aktuální znak. Jak je vidět z tabulky na obrázku 25, jsou hodnoty *Val* vypočteny podle vzorce v posledním sloupci. Vzhledem k tomu, že se sekvence opakuje vždy po čtyřech čteních, není nutné v tabulce dále pokračovat. Na základě pozorování posledního sloupce jo možné vytvořit univerzální vzorec pro složení bufferu *Val*.

```
val = (act  << (shiftAct)) | (last >> (32 - shiftAct))
```

V rámci výše uvedeného vzorce je dobré si povšimnout obráceného pořadí operací oproti tabulce. To je dáno použitým endianem na grafické kartě. V tabulce uvedené operace jsou pro Big endian, který je snadněji čitelný a vizualizovatelný, protože odpovídá čtení zleva doprava. Výpočetní shadery pak pracují, stejně jako operační systém Windows, s Little endianem. Čísla jsou poskládána opačným způsobem, zprava doleva. Chceme-li otestovat naší grafickou kartu, stačí uložit do bufferu na kartu libovolné číslo (jeho bitová reprezentace nesmí být ale symetrická), provést nad ním bitové operace a následně ho vrátit zpět na CPU. Stejné operace nad stejným číslem provedeme i zde. Pokud se výsledky shodují, jsou endiany obou systémů shodné.

Rozdělení načteného bufferu *Val* na trojici hodnot je realizováno pomocí bitových operací a masek, jak je uvedeno v následující ukázce.

```
pozice =  val & 0x1ff;
délka = (val & 0xfe00  ) >>  SEARCH_BIT_SIZE;
následující znak = (val & 0xff0000) >> (SEARCH_BIT_SIZE + LOOK_AHEAD_BIT_SIZE);
```

Ukládání dekomprimovaných hodnot opět využívá buffer. Načtené hodnoty se kumulují do 32bitového bufferu a k zápisu do globální paměti dochází pouze po jeho naplnění. Důvod je stejný jako v případě čtení, GPU v shaderech neumí pracovat se znaky.

Kompletní základ algoritmu pro dekompresi bez některých urychlení je uveden v algoritmu 6. Inicializace proměnných a polí nejsou pro větší přehlednost uvedeny. Uvedená verze pracuje pouze s jedním vláknem, jedná se tedy o sériovou verzi upravenou pro běh ve výpočetním shaderu.

Algoritmus 6 Výpočetní shader pro dekompresi LZ77. Pro přehlednost kódu nejsou zahrnuty optimalizace.

```
#define CYCLIC_BUFFER(a, b) ((a + b) % SEARCH_BUFFER_SIZE))
valAct = Compressed[1];

do
{
        valLast = valAct;

        uint shiftAct = (iterIndex % 4) * BITS_IN_BYTE;
        if (shiftAct == 0)
        {
                val = valLast;
                valAct = valLast;
        }
        else
        {
                valAct = Compressed[blockIndex];
                val = (valAct  << (shiftAct)) | (valLast >> (32 - shiftAct));
                blockIndex++;
        }
        iterIndex++;

        pos    =  val & 0x1ff;
        length = (val & 0xfe00  ) >>  SEARCH_BIT_SIZE;
        znak  = (val & 0xff0000) >> (SEARCH_BIT_SIZE + LOOK_AHEAD_BIT_SIZE);

        tmpIndex = 0;
        tmpVal = 0;

        for (i = SEARCH_BUFFER_SIZE - pos; i < SEARCH_BUFFER_SIZE - pos + length; i++)
        {
                if (i >= SEARCH_BUFFER_SIZE)
                {
                        tmpVal = lookAheadBuffer[i - SEARCH_BUFFER_SIZE];
                }
                else
                {
                        tmpVal = searchBuffer[CYCLIC_BUFFER(searchIndex, i)];
                }
                WriteDecompressed(tmpVal, storeBuffer, index);

                lookAheadBuffer[tmpIndex] = tmpVal;
                tmpIndex++;
        }

        WriteDecompressed(znak, storeBuffer, index);

        lookAheadBuffer[tmpIndex] = znak;

        searchIndex += length + 1;
        tmpIndex = 0;
        for (i = SEARCH_BUFFER_SIZE - (length + 1); i < SEARCH_BUFFER_SIZE; i++)
        {
                tmpVal = lookAheadBuffer[tmpIndex];
                tmpIndex++;
                searchBuffer[CYCLIC_BUFFER(searchIndex, i)] = tmpVal;
        }
} while (index < decompSize);
```

V případě paralelizace je potřeba do shaderu předat informace o jednotlivých blocích. Důležitá je počáteční pozice (index) komprimovaných dat a počáteční pozice (index) pro zápis dekomprimovaných dat v rámci jednoho vlákna. Předat tyto dvě hodnoty

lze například pomocí strukturovaného bufferu. Je zbytečné používat texturu, protože každé vlákno načte hodnotu pouze jednou při inicializaci.

Buffery použité při dekompresi (*searchBuffer* a *lookAheadBuffer*) jsou uloženy ve sdílené paměti jako 32bitové pole. Zde není potřeba řešit převod mezi zapsanými hodnotami (8bit) a vlastním datovým typem (32bit). Vzhledem k bitovým operacím při zápisu výsledných znaků se tato nesrovnalost ruší. Samozřejmě, pokud bychom chtěli pracovat s větší hodnotou slovníků, mohli bychom velikost bufferů zvětšit a ukládat do jedné hodnoty čtyři znaky. Pro čtení a zápis bychom použili upravené operace pro přístup k prvkům tohoto pseudo-pole. Bohužel, použití těchto operací znatelně zpomalilo dekompresi.

```
#define POSITION_GET(a, b) ((a[(b) >> 2] >> (8 * ((b) & 3))) & 0xFF)
#define POSITION_SET(a, b, c)  (a[(b) >> 2] = (a[(b) >> 2] & (~(0xFF << (8 * ((b) & 3)
    )))) | ((c) << (8 * ((b) & 3))))
```

Vzhledem k potřebě unikátních bufferů pro každé vlákno se nám snižuje maximální počet vláken dostupných pro dekompresi. Současná implementace pracuje s více skupinami vláken, kdy v rámci každé skupiny běží vždy jedno vlákno. Tato situace není optimální, ideální stav by byl v každé skupině spustit více vláken. Maximální počet vláken na skupinu je pak limitovaná velikostí sdílené paměti. Součet velikostí všech bufferů v rámci skupiny musí být maximálně do velikosti kapacity paměti dostupné konkrétní grafickou kartou. Na základě testů bylo zjištěno, že více vláken uvnitř jedné skupiny dekompresi ještě zpomalilo. To může být dáno limitním stropem, kterého se dosáhlo. Aplikace byla s velkou pravděpodobností již limitována operacemi nad globální pamětí, nikoliv vlastním algoritmem LZ77.

8.3.2 LZSS

Nevýhoda algoritmu LZ77 je jeho kompresní poměr. V praxi komprimujeme i znaky, které se vyskytují samostatně. Může se tak velmi snadno stát, že soubor místo zmenšení svoji velikost naopak zvětší. Tuto nevýhodu ve většině případů algoritmus LZSS odstraňuje. Pokud máme již naprogramovanou verzi LZ77, její převedení na LZSS je velmi jednoduchá záležitost. Pro připomenutí je opět uvedena již jednou zmíněná ukázka komprese.

```
Vstup: abracadabra
Výstup: abracad[7,4]
```

Pro každý znak potřebujeme navíc jeden bit, který určí jeho typ. V případě nulového bitu je následující znak reprezentován přímo ve své podobě, v opačném případě se provádí dekomprese (na ukázce dvojice [7, 4]).

V případě komprese je vhodné zvolit shlukování těchto informačních bitů z důvodu lepšího cachování. Pracovat se znaky o velikosti 9bit není příliš pohodlné. Informační bity shlukuji do bloků o velikosti 16bitů. Následujících 16 hodnot potom odpovídá nastavení informačních bitů. Na následující ukázce je vidět základní myšlenka.

```
0110010...||8|A|A|8|8|A|8|....||
```

Počáteční posloupnost určuje informace, které začínají za dvojitou čárou. Velikost 8bit odpovídá ASCII znaku, písmeno A pak označuje komprimovanou sekvenci. Velikost této

sekvence je opět dána, stejně jako v případě LZ77, velikostí slovníků. Již z minulé sekce se nám osvědčila velikost vyhrazená pro slovníky jako 16bitové číslo. Opět použijeme slovníky o maximálních velikostech 511 (9 bitů) a 127 (7 bitů) znaků.

Systém načítání je v tomto případě ovšem třeba přizpůsobit, protože již nepracujeme s trojicemi. Nyní načítáme buď jeden znak, nebo dvojici. Vzhledem k této nesymetrii nelze aplikovat stejný načítací přístup, jaký byl uveden na obrázku 25. Upravený postup je uveden v algoritmech 7 a 8. Pracujeme zde s několika proměnnými, které jsou uvedeny v tabulce nad vlastním algoritmem.

Algoritmus 7 Načítání dat do bufferu

Proměnná	Význam
bitIndex	index v bitovém streamu
tmpIndex	dočasný index v rámci aktuálního bufferu
lastShift	velikost posledního zpracovávaného typu (8 nebo 16)
val	aktuální obsah bufferu
v2	poslední načtená hodnota z komprimovaných dat
v2Shift	postupně se posunující okno s polední načtenou hodnotou

```
if (((bitIndex % BITS_IN_UINT) == 0)||(tmpIndex >= BITS_IN_UINT))
{
        lastShift = bitIndex % BITS_IN_UINT;
        tmpIndex = 0;
        val = v2;
        v2 = Compressed[(bitIndex - lastShift) / BITS_IN_UINT];
        v2Shift = v2;
}

val = (val >> lastShift);

if (lastShift)
{
        val |= v2Shift << (BITS_IN_UINT - lastShift);
        v2Shift = v2Shift >> lastShift;
}
```

Paralelizace kódu se provádí stejným způsobem jako v případě verze LZ77. Je potřeba také předat informace o jednotlivých blocích.

Algoritmus 8 Výpočetní shader pro dekompresi LZSS. Pro přehlednost kódu nejsou zahrnuty optimalizace.

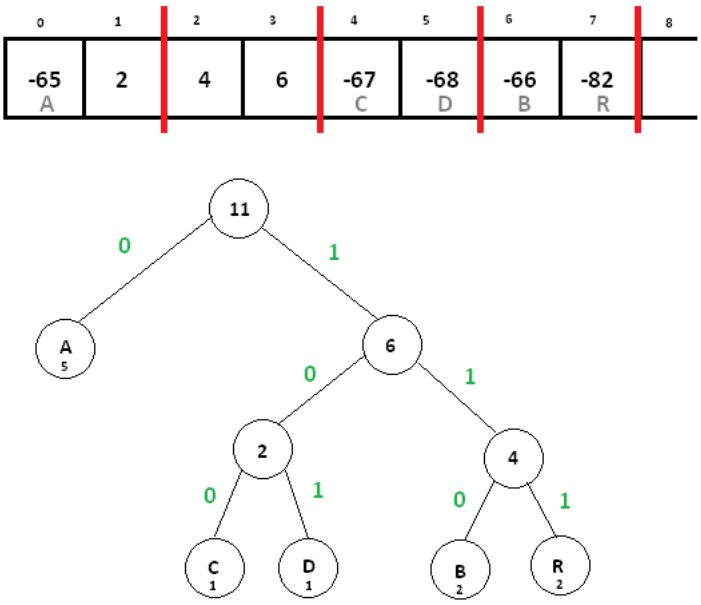
```
#define CYCLIC_BUFFER(a, b) ((a + b) % SEARCH_BUFFER_SIZE))
tmp = CODE_INFO_SIZE;
do {
      /* Kód z algoritmu 8*/

      if (tmp == CODE_INFO_SIZE)
      {
            codedInfo = val & CODE_INFO_MASK;
            lastShift = CODE_INFO_SIZE;
            bitIndex += CODE_INFO_SIZE;
            tmpIndex += CODE_INFO_SIZE;
            tmp = 0;
      }
      else
      {
            if (codedInfo & CODED_MASK)
            {
                  lastShift = SEARCH_BIT_SIZE + LOOK_AHEAD_BIT_SIZE;
                  bitIndex += SEARCH_BIT_SIZE + LOOK_AHEAD_BIT_SIZE;
                  tmpIndex += SEARCH_BIT_SIZE + LOOK_AHEAD_BIT_SIZE;
                  pos = val & POS_MASK;
                  length = (val >> SEARCH_BIT_SIZE) & LENGTH_MASK;

                  for (i = SEARCH_BUFFER_SIZE - pos; i < SEARCH_BUFFER_SIZE - pos +
                      length; i++)
                  {
                        if (i >= SEARCH_BUFFER_SIZE)
                        {
                              tmpVal = lookAheadBuffer[i- SEARCH_BUFFER_SIZE];
                        }
                        else
                        {
                              tmpVal = searchBuffer[CYCLIC_BUFFER(searchIndex, i)];
                        }

                        WriteDecompressed(tmpVal, storeBuffer, index);

                        lookAheadBuffer[i - (SEARCH_BUFFER_SIZE - pos)] =  tmpVal;
                  }

                  searchIndex += length;

                  for (i = 0; i < length; i++)
                  {
                        tmpVal = lookAheadBuffer[i];
                        sbIndex = SEARCH_BUFFER_SIZE - length + i;
                        searchBuffer[CYCLIC_BUFFER(searchIndex, sbIndex) = tmpVal;
                  }
            }
            else
            {
                  lastShift = BITS_IN_BYTE;
                  bitIndex += BITS_IN_BYTE;
                  tmpIndex += BITS_IN_BYTE;

                  tmpVal = val & ZNAK_MASK;
                  WriteDecompressed(tmpVal, storeBuffer, index);
                  lookAheadBuffer[0] = tmpVal;

                  searchIndex++;
                  sbIndex = SEARCH_BUFFER_SIZE - 1;
                  searchBuffer[CYCLIC_BUFFER(searchIndex, sbIndex) = tmpVal;
            }
            codedInfo >>= 1;
            tmp++;
      }
} while (index < decompSize);
```

Vlastní algoritmus dekomprese je složitější, takže na první pohled bychom očekávali pokles výkonu. Musíme si ovšem uvědomit, že se jedná o efektivnější kompresi a tudíž vstupní komprimovaná data jsou menší. Vzhledem k tomu, že verze LZ77 byla limitována operacemi nad globální pamětí, dojde zde ke snížení jejich počtu. Ve výsledku je tak očekáván vyšší výkon, než v minulém případě.

8.3.3 Huffmanovo kódování

Slovníkové algoritmy potřebují dodatečný prostor v rámci každého vlákna pro uložení slovníků. V případě Huffmanova kódování potřebujeme pouze přístup ke stromu, který ovšem může být pro všechna vlákna stejný a je navíc určen pouze ke čtení. To nám umožňuje větší paralelizaci než v případě slovníkových kompresí. Bohužel větší rychlost dekomprese je ve většině případů vykoupena nižšími kompresními poměry.

Již jednou byla zmíněna problematika přenosu stromových struktur z CPU na GPU přímo. Huffmanův strom lze dobře linearizovat a uložil ho jako 1D pole. Hodnoty v poli jsou indexy potomků. Pokud se jedná o list, je uložená hodnota záporná nebo nula. Reprezentace je ukázána na obrázku 26. Chceme-li například lokalizovat posloupnost 110 (znak B), budeme postupovat následovně. Načteme první bit posloupnosti, tzn. 1. Hodnota 1 určuje pravý podstrom. Zvolíme uzel na pozici *[1]*. Jeho hodnota je číslo 2, což nám udává index potomka. Následující načtená hodnota ze vstupní posloupnosti je opět 1. V potomkovi se posuneme na pozici *[2 + 1]*, na které leží hodnota 6. V dalším kroku načteme ze vstupu bit 0. V poli vybereme hodnotu na indexu *[6 + 0]*, což nám dává negativní hodnotu -66. Hodnota je záporná, dosáhli jsme listu. Dekódovaný znak odpovídá absolutní hodnotě, tudíž 66. To je naše hledaná hodnota B. Tento přístup k průchodu stromem je velmi rychlý a nevyžaduje žádná větvení ani podmínky. Pouze postupujeme v cyklu, dokud nenarazíme na zápornou (nebo nulovou) hodnotu.



Obrázek 26: Uložení Huffmanova stromu jako 1D pole

Pro větší rychlost operací je strom uložen buď v textuře, kam je ovšem třeba přistupovat po celočíselných indexech, nebo lépe ve sdílené paměti. Uložení stromu v globální paměti a následné operace čtení jsou příliš pomalé a brzdí celý dekompresní algoritmus 9.

Algoritmus 9 Výpočetní shader pro dekompresi Huffmanova kódování

```
do
{
        val = Compressed[(bitIndex / BITS_IN_UINT)] >> (bitIndex % BITS_IN_UINT);
        if ((bitIndex % BITS_IN_UINT) != 0)
        {
                val = val | (Compressed[(bitIndex / BITS_IN_UINT) + 1] <<  (
                    BITS_IN_UINT - (bitIndex % BITS_IN_UINT)));
        }

        do
        {
                bit = val & 1;
                val >>= 1;
                tmp = hufTree[tmp + bit];
                bitIndex++;

        } while(tmp > 0);

        tmp *= -1;
        WriteDecompressed(tmp, storeBuffer, index);

} while (index < decompEndIndex);
```

8.4 Testy

Testování je rozděleno na dvě části. V první části se zabýváme kompresním poměrem, v druhé pak rychlostí dekomprese. Obě části lze považovat za stejně důležité, jelikož spolu úzce souvisí. Rychlost dekomprese může dosahovat vynikajících časů, ovšem kompresní poměr může být nízký. To samé platí samozřejmě i naopak. Důležité je najít vhodný kompromis. K výpočtu kompresního poměru je použitý vztah 17.

$$kompresni\_poměr = velikost_{komprimovano}/velikost_{vstup} \cdot 100\% \tag{17}$$

8.4.1 Testovací data

V rámci testovaných dat byly zvoleny jak statické snímky pro ověření účinnosti navržených kompresí, tak i časově proměnná data (animace). Některé volumetrické animace lze nalézt v [18].

V rámci statických dat jsem testoval účinnost komprese na modelu *bloku motoru* o rozlišení 256x256x256. Model obsahuje velké množství prázdného prostoru, který lze dobře eliminovat. Je očekáván relativně velký kompresní poměr, o čemž vypovídá i velmi nízká hodnota entropie 0.01 bitu / byte. Podobný případ je očekáván i v případě modelu *Syn64* o rozlišení 64x64x64, kde jsou uložena homogenní data s velkým množstvím prázdného prostoru. Entropie tohoto souboru je 0.4 bitu / byte

Z množiny vědeckých dat bylo zvoleno několik *statických snímků z animace raketového motoru* o rozlišení 128x128x128 a 4byty na jeden voxel. Vzhledem k celkovému zaplnění prostoru je očekáván nižší kompresní poměr. Snímky byly vybrány tak, aby některé z nich obsahovali data s velkou mírou entropie (5.2 bitu / byte), jiné naopak s malou entropií (4.1 bitu / byte), kde je očekáván velký kompresní poměr.

Jako časově proměnná data bylo použito *200 snímků animace raketového motoru*, každý snímek o rozlišení 128x128x128 a 4byty na voxel. Rozsah hodnot v rámci dat se pohybuje v průměru od 0.99 do 1.5. Průměrná entropie datového setu je 4.6 bitu / byte. Na základě doporučení autora dat nebyl brán každý krok animace, ale pracovalo se pouze s každým desátým snímkem celkového datového setu.

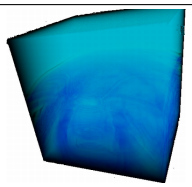
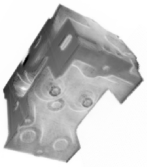
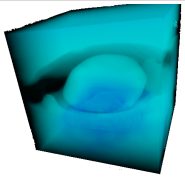
8.4.2 Testy kompresních poměrů

Mezi bezeztrátovými kompresemi jednoznačně zvítězila komerční implementace WinRar. Moje implementace kompresních algoritmů se svojí velikostí přiblížila programu WinZip. Je to logické, protože i ten kombinuje LZSS a Huffmanovo kódování. WinRar je také založen na LZSS, ale v dalších krocích již nepracuje s Huffmanem. V rámci LZSS byly použity dva různé rozsahy slovníků. První pracoval s velikostí 255 (8 bitů) a 63 (6 bitů). Druhá verze pak ukazuje doporučenou velikost 511 (9 bitů) a 127 (7 bitů). Rozdíl ve velikosti komprese není zanedbatelný. Zvyšujeme-li dále velikost slovníku, postupně se rozdíl v kompresích snižuje, což je dáno potřebou více bitů pro slovník.

Pro porovnání lze na data pohlížet jako na soubor 2D obrázků uložených v jednom velkém obrazu. Na základě tohoto přístupu můžeme použít klasické komprimační technologie známé z obrazových formátů. Algoritmus JPEG selhal s chybovou hláškou, označující rozměry obrázku jako příliš veliké. Kompresní algoritmus JPEG2000, hojně

používaný pro volumetrická data, má v bezeztrátové podobě značně špatný kompresní poměr. Ve výsledku tak klasické bezeztrátové kompresní algoritmy dosahují lepších výsledků. Vezmeme-li v potaz i dekompresní algoritmus, nevyplatí se příliš používat JPEG2000 ve své bezeztrátové variantě. Zde je ideální kandidát PNG, který je víceméně založen na stejném principu jako Zip. V případě ztrátové komprese je pak JPEG2000 velmi ovlivněn typem dat. Pokud jsou data různorodá, dosahuje jeho ztrátová verze v porovnání s neobrazovými algoritmy velmi špatných kompresních poměrů. Na druhou stranu musíme brát v úvahu, že na rozdíl od nich nabízí například náhodný přístup k datům. V našem případě toto ovšem není potřeba. Je lepší data nejprve dekomprimovat a pak teprve zahájit vlastní vykreslování.

Kompletní výsledky jsou shrnuty v tabulkách 9 a 10.

Soubor #1	Soubor #2	Soubor #3
		
snímek animace s malou entropií	blok motoru	snímek animace s velkou entropií
128x128x128	256x256x256	128x128x128
float	byte	float

Typ komprese	Soubor #1	Soubor #2	Soubor #3	Poměr #1	Poměr #2	Poměr #3
-	8 388 608	16 777 216	8 388 608	-	-	-
LZ77	7 487 872	6 393 487	8 376 583	89.26	38.11	99.85
Huffman	7 197 078	3 936 345	5 855 054	85.80	23.46	69.80
LZSS (255, 63)	6 144 662	4 978 245	7 490 998	73.25	29.67	89.30
LZSS (511, 127)	5 765 515	4 493 111	7 051 464	68.73	26.78	84.06
LZSS + Huffman	4 486 355	3 748 575	5 299 225	53.48	22.34	63.17
WinZip	4 128 319	3 177 702	4 899 516	49.21	18.95	58.41
7-Zip (LZMA)	3 057 929	2 547 821	3 242 415	36.45	15.19	38.65
WinRar	2 468 847	2 754 712	2 993 166	29.43	19.42	35.68

Tabulka 9: Porovnání kompresí různých statických snímků, dle první části tabulky. Uvedené velikosti jsou v bytech. Kompresní poměry jsou uvedeny v procentech.

Typ komprese	Soubor #1	Soubor #2	Soubor #3	Poměr #1	Poměr #2	Poměr #3
-	8 388 608	16 777 216	8 388 608	-	-	-
JPEG2000 (losless)	6 511 996	2 209 498	8 071 533	77.63	13.17	96.22
JPEG2000 (80%)	4 758 958	537 697	5 881 817	56.73	3.21	70.12
PNG	4 204 741	3 193 525	4 979 928	50.12	19.03	59.37
JPEG2000 (40%)	2 947 205	221 303	3 614 835	35.13	1.31	43.10

Tabulka 10: Porovnání kompresí jednoho statického snímku, na který je pohlíženo jako na obrázek. Uvedené velikosti jsou v bytech. Kompresní poměry jsou uvedeny v procentech.

Kvalita komprese se více projeví na posloupnosti snímků, kde jsou různě proměnlivé hustoty dat. Jako omezující faktor, který je ovšem logický, uvažujeme všechny snímky v posloupnosti stejně rozměrné. V opačném případě by nefungovala výše navržená metoda pro dekompresi příliš efektivně.

V případě animace se již vyplatí uvažovat i ztrátové komprese, kdy je využit dříve probíraný algoritmus L3VQ. Pomocí tohoto algoritmu komprimujeme každý snímek zvlášť. Následně lze takto komprimovaná data ještě dále komprimovat bezeztrátovými algoritmy. Testováno bylo prvních 25 snímků *animace raketového motoru*. Shrnutí měření je uvedeno v tabulce 11.

Typ komprese	Bezeztrátová animace	L3VQ animace	Poměr - Bezeztrátová animace	Poměr - L3VQ animace
-	209 717	5 127	-	-
LZ77	81 819	3 265	39.01	63.68
Huffman	100 527	3 753	47.93	73.20
LZSS	70 125	3 662	33.44	71.42
LZSS + Huffman	62 326	3 020	29.72	58.90
WinZip	50 098	2 269	23.89	44.26
WinRar	31 014	2 258	17.79	44.05

Tabulka 11: Porovnání kompresí sekvence *25 snímků animace raketového motoru*. Uvedené velikosti jsou v KB. Kompresní poměry jsou uvedeny v procentech.

Testován byl také blokový přístup k rozložení snímku. Každý blok byl komprimován samostatně pomocí algoritmu LZSS. V tomto případě byl očekáván větší kompresní poměr. Z naměřených hodnot ovšem vzešla průměrná úspora velmi malá - pouze v řádu jednotek procent.

V případě dat v plovoucí řádové čárce jsou ve většině případů data uložena ve formátu 32-bitového floatu. Vzhledem k typu dat ovšem ve většině případů není využit celý rozsah možné přesnosti. Některé datové sety si tak vystačí s využitím zkráceného formátu plovoucí řádové čárky, tzv. half (někdy také označován jako 16-bitový float). Tento datový formát jsem otestoval na datovém setu raketového motoru. Přesnost dat tento přístup umožňuje. V tomto případě se ovšem vzdáváme bezeztrátové reprezentace

dat, protože převod mezi reprezentacemi v plovoucí řádové čárce není jednoznačný. Chybu během vizualizace úspěšně maskuje použitá Gaussova přenosová funkce [13]. Vizuální vjem je tak na úrovni originálních dat.

Pro využití 16-bitového floatu na vstupní data *animace raketového motoru* bylo naměřeno průměrné *PSNR* o hodnotě 75.23. Další aplikované bezeztrátové komprese již nijak dále *PSNR* neovlivňují.

Druhá možnost je upravit interval hodnot datového setu na rozsah <0, 1>. Na základě této reprezentace pak uložíme desetinnou část v podobě 16-bitového celočíselného typu. Tento přístup nabízí vyšší přesnost desetinné části oproti použití typu half. Určitá ztrátovost dat je zde zanesena také, vlivem přemapování dat na požadovaný interval. Problém této reprezentace je zpomalení vizualizace, pokud chceme data převádět zpět do plovoucí řádové čárky. Účinnost sekundární komprese v je tomto případě nižší.

Testy pro uložení dat v zachování plovoucí řádové čárky, ale omezení rozsahu na 16-bitů jsou uvedeny v tabulce 12.

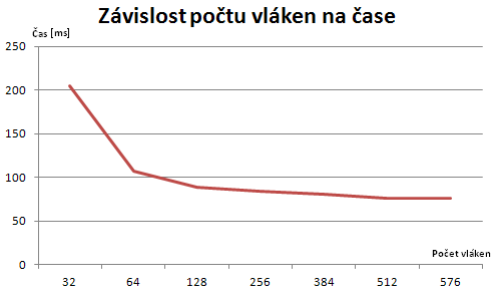
Typ komprese	Velikost dat	Poměr
-	1 258	-
float16	629	50.00
float16 + Huffman	513	40.78
float16 + LZ77	79	6.28
float16 + LZSS	75	5.96

Tabulka 12: Porovnání kompresí sekvence *150 snímků animace raketového motoru*. Využita je změna datové reprezentace z 32-bitového floatu na nižší datový typ. Velikost je uvedena v MB, kompresní poměr v procentech

8.4.3 Testy časů dekomprese

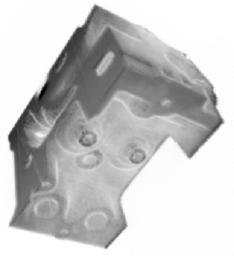
Jak již bylo jednou uvedeno, kompresní poměry a velikosti nejsou v našem případě jediné rozhodující kritérium o kvalitě komprese. Druhá, důležitější veličina, je dekompresní čas, kterého lze dosáhnout na grafické kartě. Z algoritmů testovaných v první části byl testován LZ77, LZSS a Huffmanovo kódování. Kombinace Huffmanova kódování a LZSS nebyla testována.

U slovníkových (LZ) algoritmů se rychlost dekomprese od použití jednoho vlákna zvyšovala až do použití zhruba 512 vláken. Zvyšování rychlosti pro různý počet vláken je uvedeno v grafu na obrázku 27.



Obrázek 27: Vliv počtu vláken u LZ algoritmů na dekompresním čase.

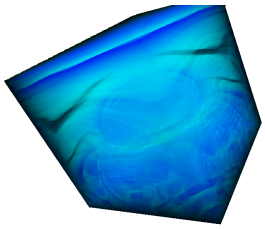
První test rychlostí dekomprese pracoval se souborem *bloku motoru*. Provedené testy pracovaly v sériové verzi s jedním vláknem a pak v plném vytížení všech 512 vláken. Jak je vidět z tabulky 13, sériová dekomprese s jedním vláknem je příliš pomalá pro běh v reálném čase. V případě paralelního běhu již dostáváme použitelné hodnoty. Nejvyššího výkonu dosahuje Huffmanovo kódování. To je dáno tím, že nejsou potřeba pomocné slovníky a lze využít plně všechna paralelní vlákna. V případě LZ algoritmů je skutečný výkon limitován počtem skupin vláken, které podporuje grafická karta.



Komprimovaná velikost	Poměr	Čas dekomprese	Počet vláken	Typ komprese
4 978 252	29.67	5036.55	1	LZSS (255, 63)
4 493 116	26.78	4907.13	1	LZSS (511, 127)
4 987 146	29.73	66.68	512	LZSS (255, 63)
4 505 130	26.85	64.22	512	LZSS (511, 64)
5 393 492	32.15	5404.73	1	LZ77
5 585 542	33.30	80.83	512	LZ77
5 382 684	32.09	$>$ 10000.00	1	Huffman
5 386 166	32.10	49.52	512	Huffman

Tabulka 13: Porovnání kompresní velikosti a času dekomprese u souboru *bloku motoru*. Časy uvedeny v ms, velikosti v bytech. Kompresní poměry jsou uvedeny v procentech.

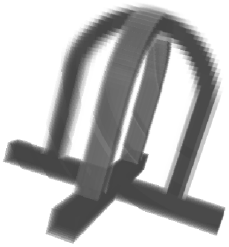
V druhém testu byly měřen časy pro špatně komprimovatelný soubor s malou redundancí dat. Jednalo se jeden *snímek animace raketového motoru*. Naměřené hodnoty se nacházejí v tabulce 14.



Komprimovaná velikost	Poměr	Čas dekomprese	Počet vláken	Typ komprese
8 115 704	96.75	3760.44	1	LZSS (255, 63)
7 540 380	89.89	2907.13	1	LZSS (511, 127)
8 125 170	96.86	43.18	512	LZSS (255, 63)
7 676 010	91.51	40.27	512	LZSS (511, 64)
8 814 252	105.01	4024.01	1	LZ77
8 866 026	105.70	45.82	512	LZ77
6 396 632	76.25	$>$ 10000.00	1	Huffman
6 400 654	76.30	23.13	512	Huffman

Tabulka 14: Porovnání kompresní velikosti a času dekomprese u *jednoho snímku animace raketového motoru*. Časy uvedeny v ms, velikosti v bytech. Kompresní poměry jsou uvedeny v procentech.

Pro třetí test byl použit soubor *Syn64*. Výsledky testů uvádím v tabulce 15.



Komprimovaná velikost	Poměr	Čas dekomprese	Počet vláken	Typ komprese
15 420	6.11	65.34	1	LZSS (255, 63)
10 840	4.30	87.37	1	LZSS (511, 127)
23 486	9.31	1.40	512	LZSS (255, 63)
19 570	7.76	0.89	512	LZSS (511, 64)
15 756	6.25	81.56	1	LZ77
25 434	10.09	1.63	512	LZ77
37 840	15.00	95.85	1	Huffman
42 410	16.82	0.60	512	Huffman

Tabulka 15: Porovnání kompresní velikosti a času dekomprese u modelu *Syn64*. Časy uvedeny v ms, velikosti v bytech. Kompresní poměry jsou uvedeny v procentech.

Na základě měření je možno posoudit, kolik času bude potřeba a jaký výkon dosáhneme s využitím uvedených algoritmů při aplikačním nasazení v reálném čase.

Pracujeme-li s animací, je pro vnímání plynulého pohybu potřeba alespoň 25 snímků za vteřinu. Jako tolerovatelnou mez lze brát i menší snímkovou frekvenci, zde pak ovšem záleží na konkrétním pozorovateli, zda vnímá pohyb přerušovaně, či nikoliv. Budeme-li se držet klasické snímkové frekvence 25, je čas vyhrazený na jeden snímek cca. 40 ms. V tomto čase je započtený čas dekomprese i vlastní vizualizace. Hodnoty naměřené ve výše uvedených tabulkách jsou při skutečném použití o několik milisekund menší, což souvisí s tím, že neztrácíme čas nutností měřit statistiky potřebné pro data uvedená v tabulkách. Pokud aplikaci krokujeme ručně, tzn. zobrazujeme si snímky postupně a nenecháváme běžet animační smyčku automaticky, prodlev v době dekomprese si nevšimneme a aplikace běží plynule.

8.4.4 Rozdíly mezi bloky

Pro kompresi časově proměnných dat lze dobře využít blokové schéma. Jak již bylo uvedeno, objemová data jsou rozdělena na jednotlivé bloky. Každý blok je následně komprimován samostatně.

V časově proměnných datech jsou v po sobě jdoucích snímcích očekávány podobné hodnoty. Některé bloky se tak v po sobě jdoucích snímcích opakují a je zbytečné je komprimovat pro každý snímek samostatně. Při dekompresi nejsou pak tyto stejné bloky přepsány novými daty a jsou využity uložené hodnoty z minulého snímku.

Tento přístup není bohužel univerzální, pokud chceme zachovat bezeztrátovost v datech. Dva po sobě jdoucí bloky jsou jen velmi ojediněle naprosto shodné. Řešení, které umožní vyšší komprese, je povolit mezi bloky určitou ztrátovost. Míra komprese je pak ovlivněna ztrátovostí těchto snímků.

Pro testované hodnoty byla zvolena chyba mezi po sobě jdoucími bloky dle vypočteného *PSNR*. Pokud tato hodnota byla nad určitou zvolenou mez, blok byl označen jako podobný a nebyl nahrazen blokem aktuálního snímku. Nejedná se o přesné řešení, pro primární testy je ovšem postačující. Naměřené hodnoty jsou uvedeny v tabulce 16.

PSNR	Velikost souboru	Poměr
-	1 258	-
70	460	36.56
40	44	3.45

Tabulka 16: Blokové schéma a naměřené hodnoty při ztrátovosti mezi jednotlivými bloky

V případě hodnoty *PSNR* 40 jsou již v datech vidět značné artefakty a data obsahují velké množství šumu.

8.4.5 Rozdílové snímky

Při animaci se předpokládají po sobě jdoucí snímky, proto lze předpokládat, že musí být i jejich rozdíly relativně malé. Nepředpokládá se výskyt situace, kdy by se data najednou mezi dvěma po sobě jdoucími snímky naprosto změnila (obdoba střihu na jinou scénu ve filmu). Této skutečnosti lze využít a data zkomprimovat na další úrovni.

Nejdříve určíme rozdíl aktuálního a následujícího snímku. Tento rozdílový snímek pak bezeztrátově zkomprimujeme.

V rámci rozdílového snímku můžeme zanést určitý stupeň ztrátovosti. Pracujeme-li s hodnotami v plovoucí řádové čárce, vznikají zaokrouhlovací chyby. Tyto nepřesné hodnoty lze nulovat a zvýšit tak kompresní poměr. Interval, ze kterého hodnoty nulujeme, pak ovlivňuje výslednou kvalitu. V rámci testů se osvědčily intervaly v rozsahu <-0.0001, 0.0001> a <-0.001, 0.001>. Mají dobrý poměr ztrátovosti a komprese. Samozřejmě závisí na typu dat a pro některá lze použít vyšší nebo nižší hodnoty mezí intervalu.

Rozdílový snímek teoreticky nemusí být uložen ve stejném datovém typu jako vlastní data. V rozdílovém snímku se předpokládají nižší hodnoty, a proto bychom si měli ve většině případů vystačit s nižším počtem bitů na vzorek. Na CPU bychom toto mohli řešit pomocí bitové masky, kdy určitý počet bitů popisuje bitovou velikost každé rozdílové hodnoty. Na GPU by tento přístup ovšem znamenal dodatečná čtení z paměti a bitové operace, což s sebou přináší zpomalení dekomprese pod hranici reálného času animace.

Další faktor ovlivňující kvalitu je počet takto komprimovaných rozdílových snímků. Vždy máme jeden řídící snímek a následující posloupnost je tvořena postupně uloženými rozdílovými snímky, které se kumulují na výchozí. V takto uspořádaném datovém toku se ovšem nelze snadno vracet ani nelze rychle vybírat náhodné snímky. K jejich zobrazení bychom potřebovali projít celou sekvenci, což je vzhledem k výkonu nemožné v reálném čase.

V tabulce 17 jsou uvedeny komprimované velikosti při použití žádné a nenulové ztrátovosti v rozdílových snímcích. Je vidět, že použití rozdílových snímků značně snižuje kompresní velikost. Kvalita v případě použitého intervalu <-0.0001, 0.0001> je v případě animace pouhým pohledem nerozlišitelná od bezeztrátově komprimovaných dat.

Typ komprese	Velikost souboru	Poměr
-	1 258	-
LZSS	1 016	80.76
LZSS + rozdílové snímky s intervalem <-0.0001, 0.0001>	627	49.84
LZSS + rozdílové snímky s intervalem <-0.001, 0.001>	294	23.37

Tabulka 17: Porovnání komprese animace skládající se z 300 snímků o rozlišení 128x128x128. Velikosti souboru uvedeny v MB.

Využití rozdílových snímků spolu s 16-bitovou reprezentací dat nebylo testováno z důvodu velmi malého ovlivnění výstupní velikosti. Navíc použití rozdílových snímků znemožňuje náhodný přístup k datům, který je v případě velikosti dat pro LZSS a 16-bitové reprezentace možný v reálném čase.

8.4.6 Celkové testy časově proměnných dat

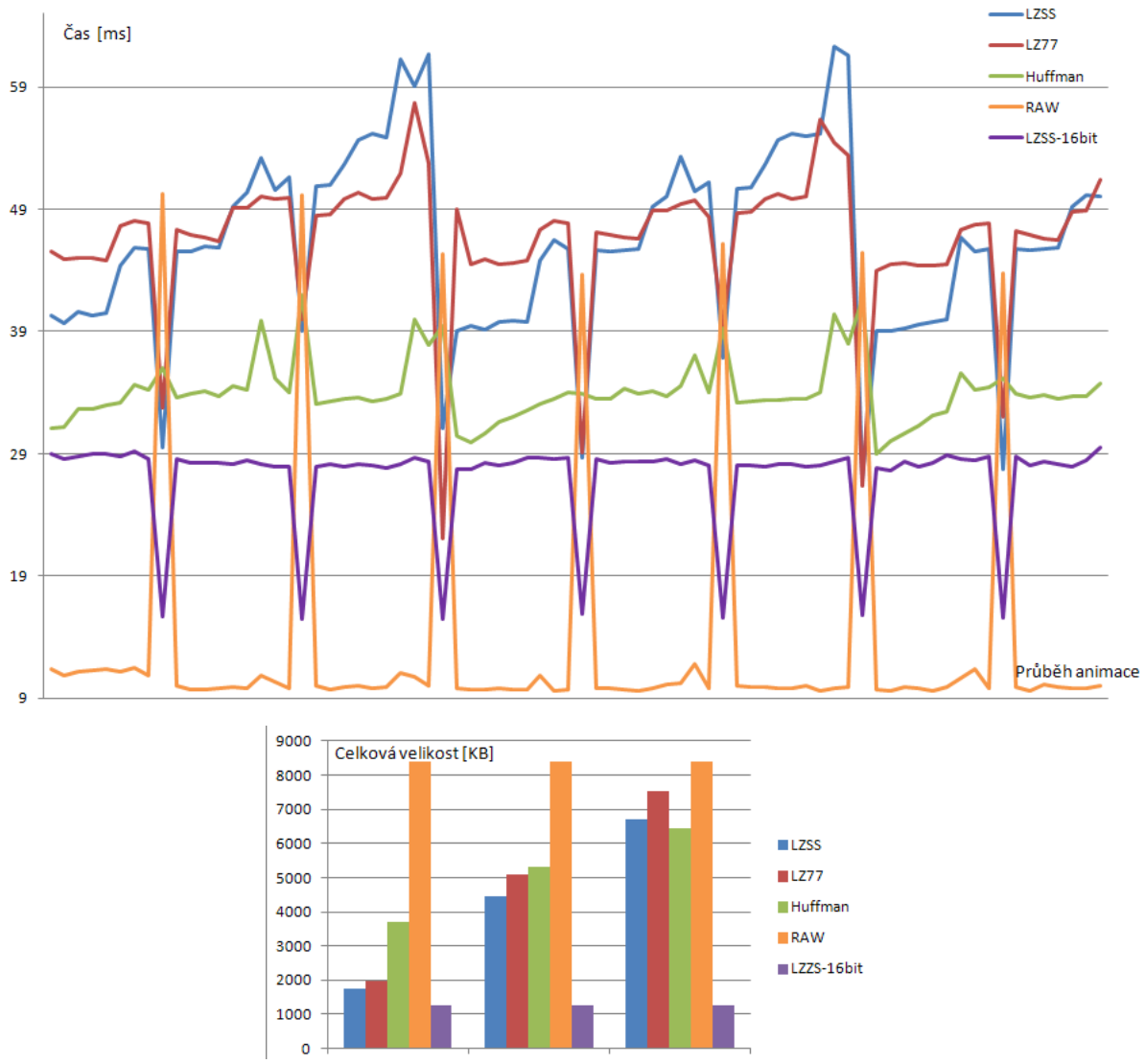
Pro grafické rozhraní DirectX11 je maximální možná velikost dat v rámci jednoho bufferu garantována na 128MB. Jakékoliv větší hodnoty jsou pak již závislé na konkrétní

použité grafické kartě a snižují tak možnou přenositelnost aplikace. Animace lze proto rozdělit do bloků o této velikosti. Ostatní načtené bloky jsou zatím odloženy v hlavní operační paměti počítače. V případě naplnění paměti je možné tyto bloky odložit dále na pevný disk, síť apod. V tomto případě ovšem musíme uvažovat operace s diskem, která je řádově pomalejší, než operace s daty načtenými přímo v paměti.

Na grafu v obrázku 28 jsou uvedeny časy (vodorovná osa), které jsou zapotřebí pro nahrání bloků dat o různé velikosti (svislá osa) na grafickou kartu.

Obrázek 28: Časy nahrávání dat na GPU.

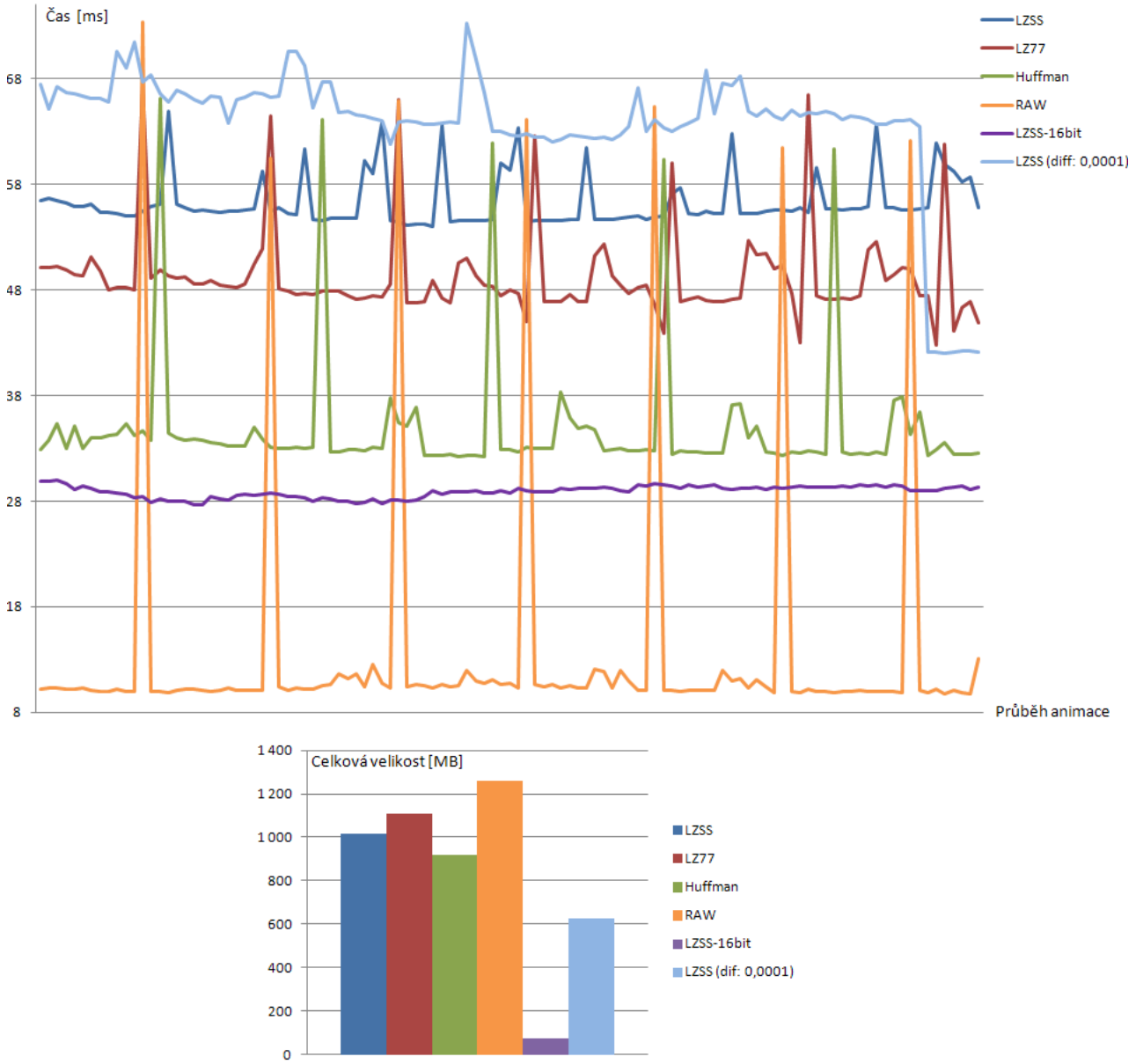
V prvním testu jsou data rozdělena podle počtu snímků. Každý datový blok měl tak jinou velikost, ale obsahoval stejný počet snímků. Nahrávání nového bloku na grafickou kartu tak vždy probíhalo po uplynutí stejné části animace.Výsledky jsou znázorněny v grafech na obrázku 29. Graf v horní části ukazuje čas snímku v ms, spodní sloupcový graf pak ukazuje odpovídající velikosti souborů, které jsou zpracovávány. Nejlepší kompromis dosahuje Huffmanovo kódování. To má sice horší kompresní poměry, ovšem dekomprese je mnohonásobně rychlejší než v případě LZSS. To je způsobeno možností využít k dekompresi více vláken než v případě LZSS, které je limitováno velikostí paměti pro slovníky. V případě, že budeme uvažovat změnu reprezentace dat, je ideální kombinace algoritmus LZSS spolu s 16-bitovou reprezentací.



Obrázek 29: Porovnání časů potřebných pro vykreslení jednoho snímku. Špičky značí nahrávání nového datového bloku na grafickou kartu. V tomto případě jsou zvoleny bloky tak, aby každý blok obsahoval stejný počet snímků. Bloky se tak liší velikostí. Rozdíly mezi velikostmi jsou uvedeny v druhém grafu.

V případě algoritmu LZSS je vidět v grafu nárůst dekompresního času, který přibližně odpovídá nárůstu velikosti komprimovaného souboru. Podíváme-li se na Huffmanův algoritmus, vidíme nižší časy, ačkoliv velikosti komprimovaných souborů pomocí Huffmanova kódování jsou prakticky velmi podobné s LZSS. Z toho důvodu je velmi pravděpodobné zpomalení v důsledku použitého algoritmu a nikoliv z důvodu operací nad globální pamětí. Zde je možno provést zajímavé pozorování. V případě zvýšení počtu vláken ve skupině nedojde u algoritmu LZSS k jeho zrychlení, ale naopak ke zpomalení. Uložení dat v 16-bitovém formátu přineslo spolu s LZSS kompresí téměř konstantní velikosti dat, každý snímek v byl ze své původní velikost 8MB uložen na méně než 0.5MB.

V druhém provedeném testu bylo nahlíženo na problém z opačné strany. Jednotlivé bloky obsahují různý počet snímků, ale velikostně jsou všechny bloky prakticky totožné (velikost 128MB). Přesnou velikost nelze vzhledem k použitým algoritmům zajistit, proto se jednotlivé soubory liší v průměru o 5MB. V případě spuštění animace dochází k nahrávání nových bloků na grafickou kartu po uplynutí různého počtu snímků. Naměřené hodnoty jsou vyobrazeny v grafu na obrázku 30.



Obrázek 30: Porovnání časů potřebných pro vykreslení jednoho snímku. Špičky v grafu značí nahrávání nového datového bloku na grafickou kartu. V tomto případě jsou všechny datové bloky stejně veliké. Liší se počet snímků v rámci jednoho bloku a tím pádem i celkový počet bloků. Druhý graf znázorňuje porovnání celkových velikostí všech bloků.

Nejlepší kompresní poměr při uvažování bezeztrátovosti nabízí Huffmanovo kódování. Nejedná se o velkou úsporu, ale spolu s relativně dobrým časem dekomprese nabízí dobrou alternativu ke klasickému uložení dat v podobě raw snímků.

Povolíme-li určitou míru ztrátovosti, lze kompresní poměry značně vylepšit. V tomto případě nebylo do testů zahrnuto Huffmanovo kódování, vzhledem k výsledkům velmi shodným s bezeztrátovou verzí. Naopak algoritmus LZSS zkomprimoval data téměř na polovinu při nulování intervalu <-0.0001, 0.0001>. Pro menší intervaly jsou již v datech viditelné artefakty.

Jednoznačně nejlepší kompresní poměr a výkon pak nabízí LZSS komprese spolu s daty uloženými v 16-bitové reprezentaci. V tomto případě se průměrný výkon blíží nekomprimovaným datům. Testy pro 16-bitovou reprezentaci a LZ77 algoritmus poskytovali horší časové výsledky, než uvedený LZSS. Huffmanovo kódování se ukázalo jako nevyhovující, vzhledem ke špatnému kompresnímu poměru a rychlosti na úrovni Huffmanova kódování pro data ve své původní reprezentaci.

Na závěr je pro porovnání uveden průměrný čas snímku. Ten byl vypočtený z dvou minut animace a jednotlivé hodnoty lze nalézt v tabulce 18. Je vidět, že v případě LZSS kódování vzhledem k velikosti dat dostáváme velmi dobré časy.

Algoritmus	Průměrný čas
-	14.03
LZSS	52.05
LZ77	47.96
Huffman	34.19
LZSS (diff: 0.0001)	61.41
LZSS (16bit)	27.52

Tabulka 18: Průměrné časy snímku animace. Časy jsou uvedeny v ms.

8.5 Zhodnocení

Vizualizace časově proměnných dat je limitována jejich velikostí. Abychom byli schopni v reálném čase tato data zobrazovat, je potřeba pracovat s určitým typem komprese. Tato komprese může být dvojího typu. Lze komprimovat samostatné statické snímky animace (této problematice se věnovala kapitola 7) a také můžeme komprimovat snímky v rámci časového průběhu.

V žádném z dostupných zdrojů nebyla nenalezena metoda, která by využila paralelního výkonu grafických karet pro slovníkové a statistické dekompresní algoritmy s využitím výpočetních shaderů. Tyto algoritmy umožňují bezeztrátové komprese dat a jsou tak vhodné pro vizualizace vědeckých dat, kde požadujeme přesné hodnoty. Zároveň je možno tyto metody využít jako druhou úroveň komprese pro jiné použité metody.

Pracujeme-li čistě s originální daty a ty komprimujeme, jsou kompresní poměry relativně nízké.

Z testovaných algoritmů dopadl nejhůře LZ77. Algoritmus nabízí špatný kompresní poměr a nemá zaručenou kompresi (v určitých případech může dojít ke zvětšení velikosti dat).

Následovník algoritmu LZ77, verze LZSS, poskytuje lepší kompresní poměry. Z testovaných algoritmů má ovšem nejhorší dekompresní časy. Kompresní poměr algoritmu záleží na typu použitých dat. Pokud jsou data hodně řídká, jsou i vysoké kompresní poměry. Naopak pro málo redundantní data je kompresní poměr velmi malý a nevyplatí se tento algoritmus využívat.

Nejlepší výsledek nabídlo Huffmanovo kódování. Z testovaných algoritmů má nejlepší dekompresní časy a kompresní poměr.

Dobrý kompresní poměr nabízí kombinace LZSS a Huffmanova kódování. Tato varianta nebyla testována na GPU. Složitost algoritmu je vyšší než u LZSS a vzhledem k dosaženým časům pro samostatný LZSS algoritmus by byly časy řádově horší.

Druhá testovaná možnost pracovala s primárním převodem dat do na jinou bitovou reprezentaci. Vstupní 32-bitová desetinná čísla byla převedena na 16-bitové hodnoty. Tato data byla dále komprimována navrženými bezeztrátovými algoritmy. V tomto případě jsou dosažené kompresní poměry již na velmi dobré úrovni. Stále máme navíc umožněn náhodný přístup k datům, kdy jednotlivé snímky nejsou závislé na ostatních.

Změna komprese dat z čistě lineárního přístupu na komprese jednotlivých bloků, kde je očekávána vyšší koherence dat, nepřinesla velké zlepšení v kompresním poměru. Naopak vzhledem k blokovému přístupu je třeba přepočítávat indexy pro zápisy dekomprimovaných hodnot, což zpomalí vlastní dekompresi.

Povolíme-li v datech na úrovni animace ztrátovost, lze dosáhnout lepší kompresní poměry. Jednotlivé snímky nejsou komprimovány samostatně, ale jako po sobě jdoucí posloupnost. Každý následující snímek je komprimován na jako rozdíl, při dekompresi je pak tento rozdíl přičten k aktuálnímu snímku. Tento postup ve většině případů vynuluje velkou část dat a umožní tak efektivní kompresi pomocí algoritmu LZSS.

Data s určitým stupněm ztrátovosti lze využít pro primární vizualizace dat, popř. jako LOD. Pro rychlé prohlédnutí animace takto jsou ztrátová data dostatečná. Pokud si chceme prohlédnou pouze určitou část ve vysoké kvalitě, využijeme poté již bezeztrátové varianty.

Neexistuje univerzální algoritmus vhodný na všechny typy dat. Navržený postup tak nemá problém komprimovat každý snímek jiným způsobem podle kvality komprese. V případě použití různých algoritmů ovšem není zachována plynulost animace, protože se liší časy dekomprese. V tomto případě by bylo potřeba programově např. omezit snímkovou frekvenci na určitou hodnotu.

Dekompresi dat přímo na grafické nemusíme používat pouze pro volumetrická data, jedná se obecný přístup použitelný v širokém spektru algoritmů. Ve většině případů je také urychleno načítání dat, protože je rychlejší data dekomprimovat na GPU, než je dekomprimovat na CPU a následně přenášet tento datový stream. Data tak lze uchovávat i na disku nebo síti v komprimované podobě, tuto komprimovanou podobu načíst na CPU a pak poslat na GPU, kde se teprve provede potřebná dekomprese. V případě přenosu dat tímto způsobem budou zisky metody větší, než v případě použití klasických nekomprimovaných dat.

9 Závěr

V práci jsem se seznámili s některými metodami pro zobrazování volumetrických dat a s programováním algoritmů s přístupem GPGPU. Z prostudovaných metod byly vybráni někteří univerzální kandidáti. Na těchto algoritmech byly provedeny základní testy. Testovací program je přiložen k této práci. Jeho popis je dostupný v sekci Příloha C.

Obecným problémem práce s objemovými daty je jejich velikost. Tuto problematiku můžeme rozdělit na dvě základní skupiny. První potřebuje pracovat s daty v co možná nejpřesnější reprezentaci, možné komprese dat jsou tak omezeny. Do této kategorie bychom mohli řadit lékařství, výzkumná pracoviště apod. Tyto obory mají ovšem dostupný jiný a výkonnější hardware, který problém může částečně eliminovat. Často bývá tento hardware navíc upraven přímo pro konkrétní účely.

Druhou kategorii pak tvoří běžní uživatelé a použití v rámci multimediální zábavy. Zde je potřeba metody cílit tak, aby je bylo možné provozovat na běžně dostupném hardwaru. V tomto využití je možno data komprimovat s většími kompresními poměry, protože případná ztráta kvality není klíčová. Většina uživatelů se spokojí s nižší kvalitou za cenu plynulého chodu aplikace. Přínos volumetrických dat v multimediální sféře je značný. Oproti dnešní reprezentaci virtuálního světa pomocí trojúhelníkových sítí umožňuje jeho snadnější deformace a jednodušší reprezentaci. Můžeme se podívat na některé dostupné prezentace těchto řešení, například [33], [5].

V teoretické části práce byl rozebrán úvod do problematiky objemových dat. Pozornost byla věnována dostupnému hardwaru a přístupům pro možné vizualizace a kompresním algoritmům pro zmenšení velikosti dat.

První část vlastní práce se zaměřila na statická data, která jsou základem všech metod. Je potřeba zajistit zobrazení jednoho konkrétního snímku. Důležitým aspektem v této oblasti je rychlost vykreslování. Pokud chceme s aplikací pracovat v reálném čase, jsou kladené požadavky přísnější, než v případě statického zobrazení jednoho snímku. Nejprve byl proveden přehled dostupných metod. V následující části bylo vybráno několik základních přístupů, které byly otestovány. Hlavní zaměření bylo směrováno na ztrátové techniky komprese. Dobrý poměr kvality a výkonu přinášejí metody založené na vektorové kvantizaci (např. algoritmus L3VQ [32]). O jejich oblibě svědčí i to, že jsou zmiňovány v různých zdrojích, které se zabývají vizualizacemi volumetrických dat. V závěru této části jsou diskutována a porovnávána jednotlivá řešení.

Ve druhé části práce jsou shrnuty poznatky o časově proměnných datech. Zde je, na rozdíl od statických snímků, hlavním problémem velikost dat. Jeden konkrétní snímek s vysokým rozlišením se nám může na základě použití různých kompresí podařit vizualizovat celkem snadno, v případě animace ovšem musíme celkovou velikost uvažovat vynásobenou počtem všech snímků.

Pro testy byly zvoleny bezeztrátové komprese dat, protože je lze aplikovat nad libovolná data. Tyto algoritmy lze vzhledem ke své univerzálnosti využít i jako sekundární kompresní metody. Data nejprve komprimujeme algoritmem vhodným pro objemová data a následně můžeme v některých případech ušetřit další prostor využitím univerzální komprese.

Cílem této práce nebylo přinést nový komprimační algoritmus, ale vyzkoušet neobvyklé kombinace již existujících řešení. Výsledkem je navržené použití algoritmů zalo-

žených na slovníkových kompresích, popř. na statistickém kódování. Současné generace grafických karet umožňují tyto algoritmy počítat v reálném čase, což na dřívějších modelech nebylo možné. Navržený postup se tak ukázal jako možná doplňková cesta pro množinu dalších algoritmů.

Reference

[1] Theory of Data Compression. online, 2000. Dostupné z: http://www.data-compression.com/. [Citováno: Leden 2012].

[2] Volvis, March 2005. Dostupné z: http://www.volvis.org/. [Citováno: Listopad 2011].

[3] ASSOCIATION, N. E. M. DICOM, February 2012. Dostupné z: http://www.nema.org/stds/dicom.cfm. [citováno: Únor 2012].

[4] BRUYLANTS T., A. A. D. R. S. P. M. A. Volumetric image compression with JPEG2000. *SPIE - The International Society for Optical Engineering*. 2007. Dostupné z: http://spie.org/documents/Newsroom/Imported/0779/0779-2007-06-12.pdf.

[5] CRASSIN, C. et al. GigaVoxels: ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, I3D '09, s. 15–22, New York, NY, USA, 2009. ACM. doi: 10.1145/1507149.1507152. Dostupné z: http://doi.acm.org/10.1145/1507149.1507152. ISBN 978-1-60558-429-4.

[6] DIETRICH, C. A. et al. Marching Cubes without Skinny Triangles. *Computing in Science and Engg.* March 2009, 11, 2, s. 82–87. ISSN 1521-9615. doi: 10.1109/MCSE.2009.34. Dostupné z: http://dx.doi.org/10.1109/MCSE.2009.34.

[7] ENGEL, K. – KRAUS, M. – ERTL, T. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, HWWS '01, s. 9–16, New York, NY, USA, 2001. ACM. doi: 10.1145/383507.383515. Dostupné z: http://doi.acm.org/10.1145/383507.383515. ISBN 1-58113-407-X.

[8] FOUT, N. – MA, K.-L. – AHRENS, J. Time-varying, multivariate volume data reduction. In *Proceedings of the 2005 ACM symposium on Applied computing*, SAC '05, s. 1224–1230, New York, NY, USA, 2005. ACM. doi: 10.1145/1066677.1066953. Dostupné z: http://doi.acm.org/10.1145/1066677.1066953. ISBN 1-58113-964-0.

[9] GUTHE, S. – STRASSER, W. Real-time decompression and visualization of animated volume data. In *Proceedings of the conference on Visualization '01*, VIS '01, s. 349–356, Washington, DC, USA, 2001. IEEE Computer Society. Dostupné z: http://dl.acm.org/citation.cfm?id=601671.601726. ISBN 0-7803-7200-X.

[10] HAYWARD, K. Volume Rendering: Transfer Functions. online, January 2009. Dostupné z: http://graphicsrunner.blogspot.com/2009/01/volume-rendering-102-transfer-functions.html. [Citováno: Květen 2012].

[11] JAN, P. Methods for iso-surface extraction and scalar data visualization. Master's thesis, Západočeská univerzita v Plzni, 2002.

[12] JENS, S. Kompressions- und Darstellungsmethoden für hochaufgelöste Volumendaten. Master's thesis, Technische Universität München, 2003.

[13] KNISS, J. et al. Gaussian Transfer Functions for Multi-Field Volume Visualization. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, VIS '03, s. 65–, Washington, DC, USA, 2003. IEEE Computer Society. doi: 10.1109/VISUAL.2003.1250412. Dostupné z: http://dx.doi.org/10.1109/VISUAL.2003.1250412. ISBN 0-7695-2030-8.

[14] KO, C.-L. et al. Multi-resolution Volume Rendering of Large Time-Varying Data using Video-based Compression. In *PacificVis*, s. 135–142. IEEE, 2008. Dostupné z: http://dblp.uni-trier.de/db/conf/apvis/pacificvis2008.html.

[15] KRUGER, J. – WESTERMANN, R. Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, VIS '03, s. 38–, Washington, DC, USA, 2003. IEEE Computer Society. doi: 10.1109/VIS.2003.10001. Dostupné z: http://dx.doi.org/10.1109/VIS.2003.10001. ISBN 0-7695-2030-8.

[16] LIPING, Z. – XIANG, Y. An Improved Volumetrix Compression Algorithm Based on Histrogram Information. 2010.

[17] LORENSEN, W. E. – CLINE, H. E. Marching cubes: A high resolution 3D surface construction algorithm. *SIGGRAPH Comput. Graph.* August 1987, 21, 4, s. 163–169. ISSN 0097-8930. doi: 10.1145/37402.37422. Dostupné z: http://doi.acm.org/10.1145/37402.37422.

[18] MA, K.-L. TVDR: Time-Varying Volume Data Repository, September 2008. Dostupné z: http://www.cs.ucdavis.edu/~ma/ITR/tvdr.html. [Citováno: Leden 2012].

[19] MA, K.-L. – CAMP, D. M. High performance visualization of time-varying volume data over a wide-area network status. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '00, Washington, DC, USA, 2000. IEEE Computer Society. Dostupné z: http://dl.acm.org/citation.cfm?id=370049.370399. ISBN 0-7803-9802-5.

[20] MAHOVKSY JEFFREY, W. B. Fast Ray-Axis Aligned Bounding Box Overlap Tests with Plücker Coordinates. Technical report, University of Calgary, 2005. Dostupné z: http://pages.cpsc.ucalgary.ca/~blob/ps/jgt04.pdf.

[21] MATT, M. Data Compression Programs, 2009. Dostupné z: http://mattmahoney.net/dc/. [Citováno: Březen 2012].

[22] MENSMANN, J. – ROPINSKI, T. – HINRICHS, K. A GPU-Supported Lossless Compression Scheme for Rendering Time-Varying Volume Data. In *Volume Graphics'10*, s. 109–116, 2010. Dostupné z: http://scivis.itn.liu.se/publications/2010/MRH10a/vg10-compression.pdf.

[23] MICROSOFT. Texture Block Compression in Direct3D 11, September 2011. Dostupné z: http://msdn.microsoft.com/en-us/library/windows/desktop/bb694531%28v=vs.85%29.aspx. [Citováno: Prosinec 2011].

[24] MICROSOFT. *MSDN (Microsoft Developer Network) - HLSL*. Microsoft, September 2011. Dostupné z: http://msdn.microsoft.com/en-us/library/windows/desktop/bb509561%28v=vs.85%29.aspx. [Citováno: Prosinec 2011].

[25] MIROSLAV, B. Knihovna vizualizačních appletů pro kompresi dat, 2005. Dostupné z: http://www.stringology.org/DataCompression/. [Citováno: Březen 2012].

[26] NAGAYASU, D. – INO, F. – HAGIHARA, K. Two-stage compression for fast volume rendering of time-varying scalar data. In *Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia*, GRAPHITE '06, s. 275–284, New York, NY, USA, 2006. ACM. doi: 10.1145/1174429.1174478. Dostupné z: http://doi.acm.org/10.1145/1174429.1174478. ISBN 1-59593-564-9.

[27] NGUYEN, K. G. – SAUPE, D. Rapid High Quality Compression of Volume Data for Visualization. *Computer Graphics Forum*. 2001, 20, s. 2001.

[28] NVIDIA. *CUDA*. NVidia, April 2012. Dostupné z: http://developer.nvidia.com/.

[29] REVELLES, J. – URENA, C. – LASTRA, M. An Efficient Parametric Algorithm for Octree Traversal. In *Journal of WSCG*, s. 212–219, 2000.

[30] ROETTGER, S. et al. Smart hardware-accelerated volume rendering. In *Proceedings of the symposium on Data visualisation 2003*, VISSYM '03, s. 231–238, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association. Dostupné z: http://dl.acm.org/citation.cfm?id=769922.769948. ISBN 1-58113-698-6.

[31] SALOMON, D. *Data Compression: The Complete Reference*. : , 2007. With contributions by Giovanni Motta and David Bryant. ISBN 1-84628-602-6.

[32] SCHNEIDER, J. – WESTERMANN, R. Compression Domain Volume Rendering. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, VIS '03, s. 39–, Washington, DC, USA, 2003. IEEE Computer Society. doi: 10.1109/VISUAL.2003.1250385. Dostupné z: http://dx.doi.org/10.1109/VISUAL.2003.1250385. ISBN 0-7695-2030-8.

[33] SILES, B. Atomontage Engine, December 2011. Dostupné z: http://www.atomontage.com/.

[34] SOHN, B.-S. – BAJAJ, C. – SIDDAVANAHALLI, V. Feature based volumetric video compression for interactive playback. In *Proceedings of the 2002 IEEE symposium on Volume visualization and graphics*, VVS '02, s. 89–96, Piscataway,

NJ, USA, 2002. IEEE Press. Dostupné z: http://dl.acm.org/citation.cfm?id=584110.584126. ISBN 0-7803-7641-2.

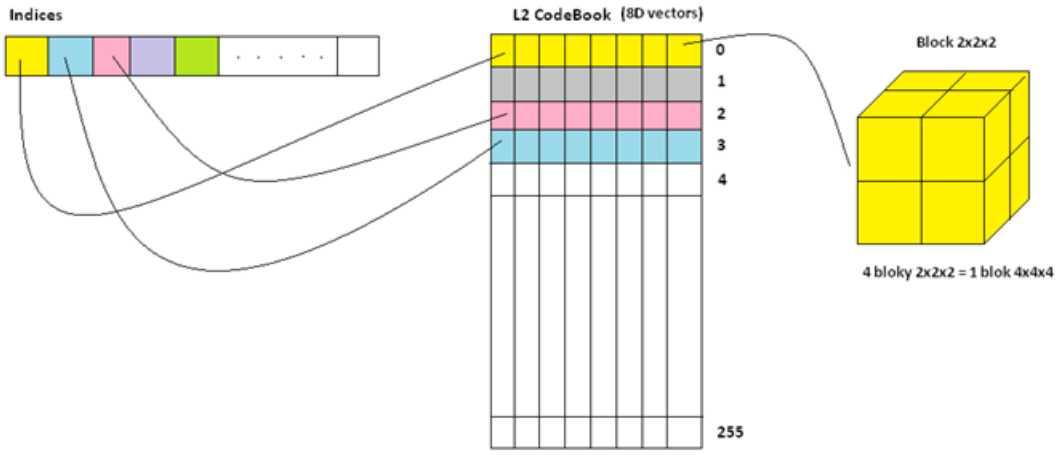
[35] TERO, L. S. K. Efficient Sparse Voxel Octrees. *IEEE Transactions on Visualization and Computer Graphics*. 2011, 17, s. 1048–1059. Dostupné z: http://www.tml.tkk.fi/~samuli/publications/laine2010i3d_paper.pdf.

[36] *Voreen (Volume Rendering Engine)*. University of Münster; Linköping University, 2.6.1 edition, February 2012. Dostupné z: http://www.voreen.org/.

Příloha A

Komprese L3VQ

1. Rozdělit data (šířka x výška x hloubka) na samostatné bloky 4x4x4
2. Pro každý blok 4x4x4:
   (a) Použít box-filtr a data 4x4x4 konvertovat na 1x1x1 (Φ_3) Vzniklou hodnotu kvantizujeme sebe samou ($\Phi_3 = L_3$)
   (b) Vypočítat diferenci mezi původními daty a následně zredukovat na blok 2x2x2
      i. Δ_2 = reduce$_2$(data - expand$_4$(L_3))
      ii. expand$_4$ => rozložení dat 1x1x1 na 4x4x4
      iii. data - expanded => vypočítat rozdíly mezi 4x4x4 buňkami
      iv. reduce$_2$ => aplikace box filtru 2x2x2
      v. Nyní máme Δ_2 jako 8D vektor, který pošleme do kvantizátoru viz. (A)
3. Najdeme scale factor - maximální absolutní hodnota ze složek všech 8D vektoru
   (a) Všechny vektory v kvantizátoru A normalizujeme scale factorem
   (b) Spustit kvantizaci A => výsledkem je blok indexů L_2 a 2D kódovací množina 256x8
   (c) Výsledky kvantizace A normujeme: (CLAMP(kódové slovo) * 127.5 + 127.5) / 127.5



Obrázek 31: Ukázka vztahu mezi indexy a kódovací množinou

4. Pro každý blok 4x4x4:
   (a) Vypočítat diferenci mezi původními daty a následně uložit blok 4x4x4
      i. Δ_1 = data - expand$_2$(decode(L_2)) - expand$_4$(L_3) decode => dekomprese zakódovaných dat z kroku 3)
      ii. expand$_2$ => rozložení dat z 2x2x2 na 4x4x4
      iii. expand$_4$ => rozložení dat z 1x1x1 na 4x4x4
      iv. Nyní máme Δ1 jako 64D vektor, který pošleme do kvantizátoru viz. (B)

5. Najdeme scale factor - maximální absolutní hodnota ze složek všech 64D vektoru
   (a) Všechny vektory v kvantizátoru B normalizujeme scale factorem
   (b) Spustit kvantizaci B => výsledkem je blok indexů L_1 a 2D kódovací množina 256x64
6. L_1, L_2 a L_3 mají stejné rozlišení - uložíme je jako RGB do 3D textury Dále uložíme rozlišení a scale faktory z kroku 3) a 5)

A, B) Kvantizátor

1. Postupně kumulujeme všechny příchozí vektory (v rámci jednoho rozměru - 64D nebo 8D)
2. Inicializace:
   (a) Vygenerujeme první kvantizační buňku V_1 - ta obsahuje veškeré vektory, které jsou v kvantizátoru uloženy
   (b) Určíme první kódové slovo (vektor) Y_1 - střed množiny V_1

```
Y1 = V1[0]
for (j = 1 to počet vektorů v množině) : Y1 += V1[j]
Y1 *= (1 / počet vektorů v množině) //scale
```

   (c) Vypočteme počáteční distorzi $d_1 = \Sigma ||X_i - Y_1||_2$ ($||A - B||_2 = (A - B) * (A - B)$)
   (d) Vytvoříme záznam {d_1, Y_1, množina V_1 } v TODO listu
3. PCA split z TODO listu vybereme záznam s největší d_i
   (a) Spočítat matici auto-kovariance $M = \Sigma (X_i - Y_j) (X_i - Y_j)^T$ .... (i ... pro všechny vektory X_i z množiny V_j)
   (b) Určit vlastní vektor matice M takový, že odpovídá maximálnímu vlastnímu číslu (E_{max}) matice M Využijeme iterativní metodu - konvergence je velmi rychlá, pokud začneme s vektorem, který je blízký maximálnímu Blízký vektor je takový, jehož délka je největší - vybereme proto z matice sloupce / řádku s max. délkou (matice je symetrická)
   (c) Rozdělit množinu V_j na dvě poloviny. Do každé části přiřadíme vektory podle podmínek: (i ... pro všechny vektory X_i z množiny V_j)

```
N = dot(E, Yj)
Levá: { dot((Yj - X), E ) < N }
Pravá: { dot((Yj - X), E ) >= N }
```

   (d) Pro každou polovinu spočítat střed a distorzi (Y a d)
   (e) Dva nově vzniklé záznamy vložíme do TODO listu
   (f) Pokud počet skupin = 2^r KONEC, jinak goto 3) (r = počet bitů slova = např. 8)

Příloha B

Průchod oktalovým stromem bez použití rekurze a zásobníku

Pseudokód předpokládá využití vektorových operací pro násobení vektoru číslem a násobení dvou vektorů mezi sebou po složkách.

Ukázka 1: Reprezentace uzlu

```
struct NodeInfo:
float3 box - střed uzlu
float3 halfSize - polovina velikosti uzlu
float3 change - poslední změna velikosti
char caseIndex - aktuální typ uzlu
char depth - aktuální hlubka
char lastDepth - poslední hlubka
```

Ukázka 2: Výpočet informací o uzlu

```
//tabulka posunů na sousedy
char nodeOffsetTable; //x = šířka; y = výška; z = hloubka
switch(node.caseIndex)
{
        case 0:
                nodeOffsetTable = (-1, -1, -1)
                break;
        case 1:
                nodeOffsetTable = (-1, -1, +1)
                break;
        case 2:
                nodeOffsetTable = (-1, +1, -1)
                break;
        case 3:
                nodeOffsetTable = (-1, +1, +1)
                break;
        case 4:
                nodeOffsetTable = (+1, -1, -1)
                break;
        case 5:
                nodeOffsetTable = (+1, -1, +1)
                break;
        case 6:
                nodeOffsetTable = (+1, +1, -1)
                break;
        case 7:
                nodeOffsetTable = (+1, +1, +1)
                break;
        default:
                break;
}

        if (node.lastDepth < node.depth)
        {
                //posun o úroveň dolů
                node.change = nodeOffsetTable * node.halfSize
                node.box += node.change
        }

        else if (node.lastDepth > node.depth)
        {
                //posun o úroveň nahoru
                node.box += node.change
                node.change = nodeOffsetTable * node.halfSize
        }
```

```
        else
        {
                //pohyb po stejné hlubce

                //"posuneme se na rodiče"
                node.box += node.change;

                //vypočítáme nové offsety pro aktuální hlubku a uzel
                node.change = nodeOffsetTable * node.halfSize

                //aktualizovat aktuální uzel
                node.box += node.change
        }

        node.change *= (-1, -1, -1)
```

Ukázka 3: Hlavní smyčka

```
inicializace výchozího uzlu (kořene) node
        node.box = dataSize * 0.5f
        node.halfSize = dataSize * 0.5f
        node.change = 0
        node.depth = 1
        node.lastDepth = 1
        node.caseIndex = 1
        actualIndex = 1 //jednotliví potomci jsou číslování 1 - 9

while(true)
{
        //jeden uzel má 8 potomků + 1 rodiče => velikost 9
        node.caseIndex = (actualIndex % 9) - 1

        while(node.caseIndex == -1)
        {
                if (actualIndex - 9 <= 0)
                {
                        //dosáhli jsme kořene - konec
                        return
                }

                //posuneme se o úroveň výše na další uzel
                actualIndex = strom[actualIndex - 9]
                actualIndex++

                //cyklický průchod potomků uvnitř jednoho uzlu
                node.caseIndex = (actualIndex % 9) - 1
                if (node.caseIndex == -1)node.caseIndex = 7
                else node.caseIndex--

                //vypočteme informace o aktuálním uzlu
                node.lastDepth = node.depth
                node.depth--

                halfInvPower2 = 1.0 / (2 * (node.depth ^ 2))
                node.halfWidth = dataSize * halfInvPower2

                CalcNewCoord(&node);
                node.lastDepth = node.depth;

                //aktualizace stavu
                node.caseIndex = (actualIndex % 9) - 1
        }

        halfInvPower2 = 1.0 / (2 * (node.depth ^ 2))

        node.halfWidth = dataSize.x * halfInvPower2
```

```
        CalcNewCoord(&node);

        //testujeme, zda paprsek protnul uzel
        if (!RayHitBox(ray, node))
        {
                //paprsek neprotnul uzel
                //posuneme se na další uzel na stejné hloubce
                actualIndex++;
                node.lastDepth = node.depth;
                continue;
        }

        if (strom[actualIndex] <= 0)
        {
                        //protnuly jsme uzel
                        if (strom[actualIndex] != EMPTY)
                        {
                                //uzel není prádný - pracuj s hodnotou
                        }

                        //posuneme se na další uzel na stejné hloubce
                        actualIndex++
                        node.lastDepth = node.depth
                        continue
        }

        //posun o level níž

        actualIndex = strom[actualIndex]
        actualIndex++   //přeskočení rodičovského "ukazatele"
        node.lastDepth = node.depth
        node.depth++
}
```

Výše popsané algoritmy nejsou optimalizované verze. Některé operace a přístupy do paměti by šlo nahradit a využívat pomocných proměnných pro snížení opakovaných čtení. Předpočítané tabulky hodnot by teoreticky mohli také vykonávání kódu urychlit. V případě cyklů záleží na implementaci překladače, zda je efektivnější použít while, do-while, popř. for cyklus.

Příloha C

Informace o programu

Pro vývoj aplikace bylo použito prostředí Microsoft Visual Studio 2010 Professional. Aplikace není přeložitelná v jiném překladači, než od firmy Microsoft. Důvodem jsou použitá proprietární řešení, jako například *Properties* v C++. Pro vlastní vykreslování bylo použito knihovny DirectX11 a shadery ve verzi 4/5. K překladu je tak potřeba nainstalovat také vývojové SDK knihoven DirectX11 ve verzi minimálně June 2010.

Aplikace se ve výchozím nastavení překládá jako dynamická knihovna. Primárně tak ovšem není psaná, pokud změníme typ projektu na spustitelnou aplikaci, dojde k překladu a spuštění.

Distribuován je celý projekt aplikace Visual Studio. V něm je třeba změnit cestu k aktuální instalaci DirectX. Výchozí hodnota očekává instalaci ve složce *C:\Program Files (x86)\Microsoft DirectX SDK (June 2010)*. Pokud je Vaše instalace v jiném adresáři, je potřeba změnit cestu pro *Include Directories* a *Library Directories*. Nastavení je dostupné z hlavního menu Visual Studia pod záložkou *Project* → *Properties*. Otevře se dialogové okno, zde vybereme záložku *Configuration Properties* → *VC++ Directories* → změníme *Include Directories* a *Library Directories* na námi požadovanou cestu. Důležité je toto nastavení změnit jak pro Debug mód, tak pro Release mód. Ostatní knihovny potřebné k běhu aplikace jsou distribuovány spolu se zdrojovými kódy a jsou umístěné v hlavním projektu ve složce *Libraries*.

Součástí aplikace je také grafické uživatelské rozhraní, které bylo napsáno v jazyce C#. Propojení s C++ aplikací je na úrovni dynamické knihovny, využit je přístup založený na COM rozhraní.

Vzhledem k nutnosti překladu shaderů během načítání programu není možné program spouštět přímo z DVD. Je potřeba program zkopírovat do složky, kam bude umožněno programu zapisovat.

CUDA verze

Pro spuštění programu ve verzi s využitým CUDA rozhraním je potřeba grafická karta firmy NVidia. Dále je třeba mít nainstalované správné ovladače, které podporují běh aplikací napsaných s využitím CUDY. Všechny další informace a dostupné ovladače lze najít na oficiálních stránkách [28].

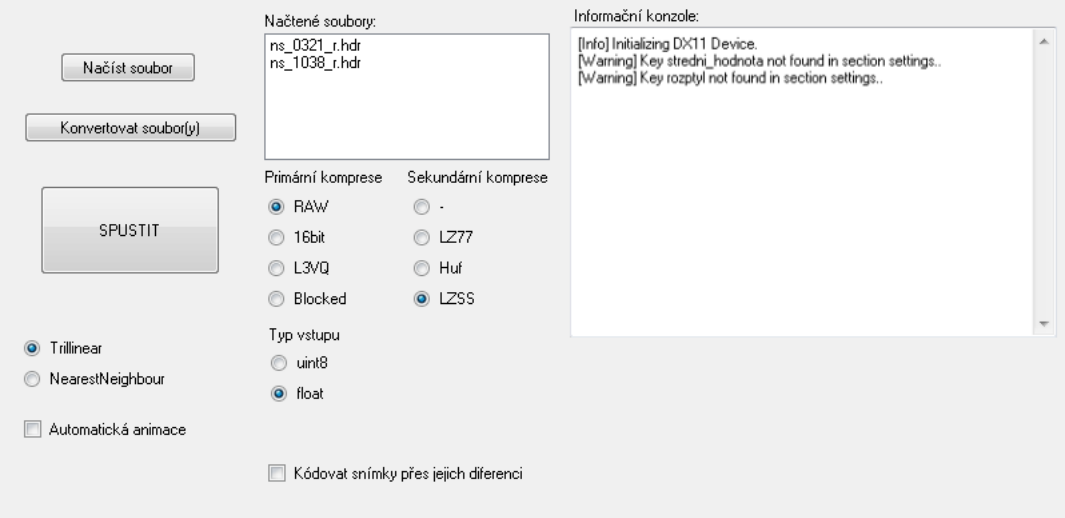
Vizualizační Engine Limitless

Vizualizační jádro aplikace využívá můj vlastní systém. Z tohoto důvodu jsou součástí aplikace také další knihovny, které ovšem nejsou samotným kódem přímo využity. Jedná se například o fyzikální systém Bullet, knihovny pro výpočet Fourierovy transformace. Odstranění těchto knihoven by vzhledem k jejich provázání se systémem mělo za následek potřebu přepsání značné části kódu, proto jsou ponechány v systému.

Ovládání programu

Verze s výpočetními shadery

Pro hlavní program bylo napsáno velmi jednoduché uživatelské rozhraní. Pro jednoduchost je rozhraní pouze jednorázové, tzn. pro opakované spuštění aplikace je třeba vše vypnout a načíst znovu.



Obrázek 32: Ilustrační ukázka jednoduchého uživatelského rozhraní

Aplikace umožňuje konvertovat soubory do vlastního formátu. Při konverzi lze zvolit různé parametry. Na obrázku 32 jsou uvedena všechna možná nastavení.

Spuštěnou vizualizační aplikaci lze ovládat pomocí myši, která rotuje objektem. Pro pohyb kamery je využito kombinace kláves WSAD. V rámci uživatelského rozhraní aplikace lze přepínat mezi trilineární a nearest neighbour filtrací. V případě načtené animace lze zapnout automatický průběh zaškrtnutím příslušné volby v aplikaci. Pokud uživatel chce krokovat animaci ručně, lze toto provést pomocí klávesy N (dopředu) a M (dozadu). Použití klávesy M bude v případě použití rozdílových snímků produkovat chybné výsledky, což je způsobeno dekompresí dat na základě pouze aktuálního snímku (neproběhne tak celá potřebná sekvence od předchozího plného snímku).

V případě prvního spuštění programu s unikátním nastavením (typ komprese, počet snímků videa apod.) je potřeba přeložit shadery. Tato operace může v závislosti na výkonu počítače trvat až desítky minut. Při dalším spuštění stejného souboru jsou pak již přeložené shadery uloženy a načítání je tak prakticky okamžité.

CUDA verze

Vzhledem k okrajovému využití CUDY ve vlastním programu je i její část aplikace velmi jednoduchá. Aplikace se ovládá z příkazového řádku. Příklad spuštění aplikace nad ukázkovými daty je uveden v následující ukázce

```
LimitlessEngine.exe Syn64_RAW.ddat
```

Pro konverzi dat není dostupná žádná programová podpora. Lze načítat data zkonvertovaná aplikací pro výpočetní shadery, podporována jsou pak pouze raw data bez žádných dodatečných kompresí a data uložená jako oktalový strom. K aplikace jsou přiloženy testovací data.

Spuštěnou vizualizační aplikaci lze ovládat pomocí myši, která rotuje objektem. Pro pohyb kamery je využito kombinace kláves WSAD.

Datová reprezentace

Pro snažší správu dat byl navržen jednoduchý formát pro volumetrická data. Jedná se komprimovaný balíček obsahující vlastní data, jejich popis a další potřebné informace závislé na vlastních datech. Přípona tohoto souboru je nastavena jako **.ddat*.

Veškeré načítání tohoto souboru je řízeno pomocí informačního podsouboru info.dat (ten je součástí balíčku). Základní ukázka pro raw data je vypsána níže. Pokud je tento soubor poškozen nebo chybí, data se nenačtou.

Ukázka 4: Informační soubor

```
[settings]
width = 256
height = 256
depth = 256
frames-count = 1
compression = RAW
secondary-compression = -
data-type = UINT8
difference-frames = false
min-corners = vector3(0.000000, 0.000000, 0.000000)
max-corners = vector3(256.000000, 256.000000, 256.000000)
```

Aplikace umí libovolný datový soubor ve formátu raw dat na tento formát převést. Pro převáděný soubor je potřeba dodat také popisný soubor s formátem ukázaným v následujícím výpisu

Ukázka 5: Popisný soubor pro raw data, která chceme převést na interní formát aplikace

```
header length: <0, N>
width: <0, W>
height: <0, H>
images: <0, D>
bits/voxel: 8 / 32
endian: B / L
file name: <file name>
```

Použité symboly značí: N - délka hlavičky, [W, H, D] - rozměry [šířka, výška, hloubka], B / L - malý / velký endian

Testovací sestava

- Systém: Microsoft Windows 7 x64 Professional
- Procesor: Intel Core 2 Duo, E6400 @ 2.14GHz

- Paměť: 5 GB RAM
- Grafická karta: NVidia GeForce 560Ti, 1 GB RAM
- Rozlišení testů: 640x480, 32bit-ová barevná hloubka

Příloha D

Obsah přiloženého DVD

Součástí této práce je DVD s doprovodnými testovacími programy a některými vzorovými daty.

- Text práce ve formátu PDF - dostupné z kořenového adresáře
- Testovací program v jazyce CUDA s přiloženými daty - složka *CUDA*
- Některé testované datové množiny - složka *DataSets*
- Hlavní testovací program - složka *MainProgram*
- Zdrojové kódy programu a knihovny - složka *src*