Univerzita Hradec Králové

Fakulta informatiky a managementu

Katedra informatiky a kvantitativních metod

Zobrazení 3D scény metodou raytracing se zaměřením na urychlující datové struktury

Diplomová práce

Autor práce: Bc. VOJTĚCH ŘEHÁK
Studijní obor: Aplikovaná Informatika

Vedoucí práce: Ing. BRUNO JEŽEK, Ph.D.

Hradec Králové 2023

Prohlášení

Prohlašuji, že jsem bakalářskou/diplomovou práci zpracoval/zpracovala samostatně a s použitím uvedené literatury

.....

Bc.Vojtěch Řehák

13. srpna 2023

Poděkování

Děkuji svému vedoucímu práce za cenné rady a podporu. Děkuji také své manželce za její podporu a trpělivost.

Anotace

Přehled urychlujících struktur pro metodu sledování paprsku, sloužící k vykreslení 3D scény. Zvláštní pozornost je věnována struktuře hierarchie intervalů, implementované na grafické kartě. Paralelizace je provedena pomocí Mortonova rozkladu. Naměřené výsledky jsou porovnány s původní implementací a ostatními urychlujícími strukturami jakou jsou Bounding volume hierarchy (BVH) nebo KD-tree. Naimplementovaná datová struktura prokazuje zásadní zrychlení při budování struktury. Naopak při procházení struktury během vykreslování je velmi pomalá kvůli vláknové a datové divergenci.

Anotation

Title: Rendering of 3D scene using raytracing method with focus on accelerating data structures

Overview of accelerating data structures for rendering method – raytracing. Special attention is paid to the bounding interval hierarchy structure implemented on the graphics card. The parallelization is done by exploiting the Z-Curve. The measured results are compared with the original implementation of BIH and other accelerating structures such as Bounding volume hierarchy or KD-Tree. The implemented data structure demonstrates a significant speedup in building the structure. In contrast, the traversal of the structure is very slow during rendering. It is caused by thread and data divergence.

Klíčová slova

sledování paprsku, hierarchie intervalů, BIH, počítačová grafika, hierarchie obálek, kd-strom, CUDA, mortonův rozklad, z-křivka, openGL

Keywords

raytracing, bounding interval hierarchy, BIH, computer graphics, bounding volume hierarchy, kd-tree, CUDA, morton code, z-order curve, openGL

Obsah

Seznam obrázků

Seznam tabulek

Seznam ukázkových kódů

1 Úvod

1.1 Rendering

Rendering je odvětví počítačové grafiky, které se zabývá syntézou obrazu. Je to proces, kdy počítačový program, tzv. „renderer“, přijímá na vstupu soubor dat a parametrů a na jejich základě vygeneruje fotorealistický či nefotorealistický obraz. Data na vstupu bývají souřadnice vrcholů v prostoru, které zpravidla tvoří trojúhelníky. Tyto trojúhelníky jsou dále uskupeny do složitějších útvarů, které ve finálním stavu tvoří 3D model, který je poté vykreslený v prostoru světa. S každým vrcholem či trojúhelníkem jsou spjaty ještě další informace jako třeba barva, normálový vektor, textura apod.

Ještě v nedaleké historii byl výpočetní výkon počítačů relativně slabý a na počítačích nešlo vykreslovat rozsáhlé scény, natož simulovat reálné šíření světla a s ním spojené světelné efekty. S růstem výpočetního výkonu se fotorealistický rendering stával stále dostupnější a čím dál více vyžadován v různých oblastech vizualizace dat.

V dnešní době je fotorealistický rendering či vykreslování velkého množství dat předmětem mnoha odvětví. Ať už se jedná o počítačové hry, design, architektonické návrhy, film nebo také například vizualizace lékařských dat ze zdravotnických zařízení jako je výpočetní tomografie.

1.2 Techniky renderingu

Pro vykreslování geometrických primitiv na obrazovku existují v dnešní době dvě základní metody. Metoda rasterizace a metoda sledování paprsku (dále „Raytracing“).

1.2.1 Rasterizace

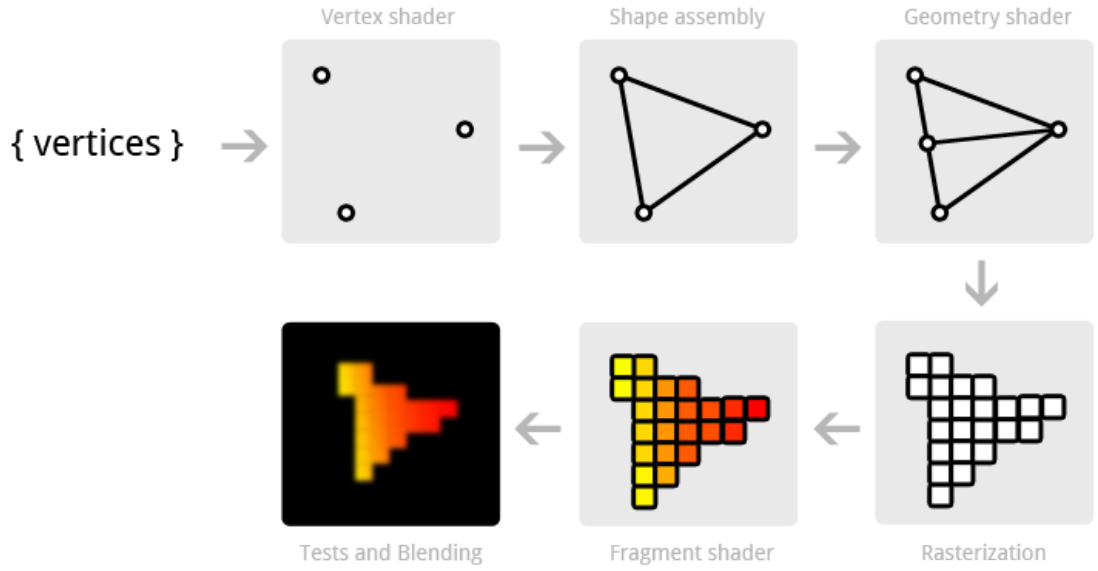
Rasterizace je produktem tzv. „vykreslovacího řetězce“. Jedná se o projekci 3D světa do dvourozměrného světa naší obrazovky. Tento řetězec vznikl historickým vývojem, kdy první grafické karty měly speciálně navržené procesory pro výpočty s geometrickými primitivy.[3] U prvních verzí těchto karet byl tento vykreslovací řetězec, složený z několika modulů, pevně daný a nebylo možné ho jakkoli programovat. To neplatí o dnešních moderních kartách. Fixní kroky byly málo flexibilní a tím i omezující při vytváření různých grafických efektů. Postupem času se tedy většina kroků (i těch volitelných, které byly záměrně vynechány, protože nejsou pro tuto práci příliš podstatné), až na pár výjimek, stala programovatelnými. Tyto moduly se nazývají „shadery“ a jsou řízeny shaderovými programy.

Průběh vykreslovacího řetězce(Obr. 1)
Na začátku řetězce se nachází tzv. „vertex shader“, což je program, který se paralelně spustí pro každý vrchol na vstupu. Vrchol zpracovaný či modifikovaný tímto programem je předán na výstup. Z vrcholů na výstupu se poté sestavují geometrická primitiva (body, čáry, trojúhelníky...).

Primitiva se následně ořezávají. To znamená, že pokud nějaká část primitiva přesahuje přes okraj „frusta“(„frustum“ - komolý jehlan, který ohraničuje prostor, ve kterém se primitiva vykreslují), je tato část oříznuta a zahozena.

Dále probíhá fáze tzv. „face culling“. Je to proces, kdy se zahodí primitiva, která jsou k pozorovateli odvrácena. Pro uzavřené povrchy jsou tato primitiva zakryta primitivy přivrácenými k pozorovateli a není tedy potřeba je vykreslovat.

Nyní nastává proces samotné rasterizace. Při tomto procesu vzniknou tzv. „fragmenty“, což jsou jakýsi potenciální kandidáti na finální barvu pixelu na obrazovce.



Obrázek 1: Vykreslovací řetězec

Na konci zobrazovacího řetězce se zpracují všechny vzniklé fragmenty v tzv. „fragment shaderu“, nad kterými se ještě provede několik testů. Tyto testy mohou způsobit, že se fragment nedostane do výsledného obrazu. Výstupem z fragment shaderu je finální barva, která se zapíše do framebufferu, což je část paměti reprezentující pixely, která se následně promítne na zobrazovacím zařízení.[4]

1.2.2 Raytracing

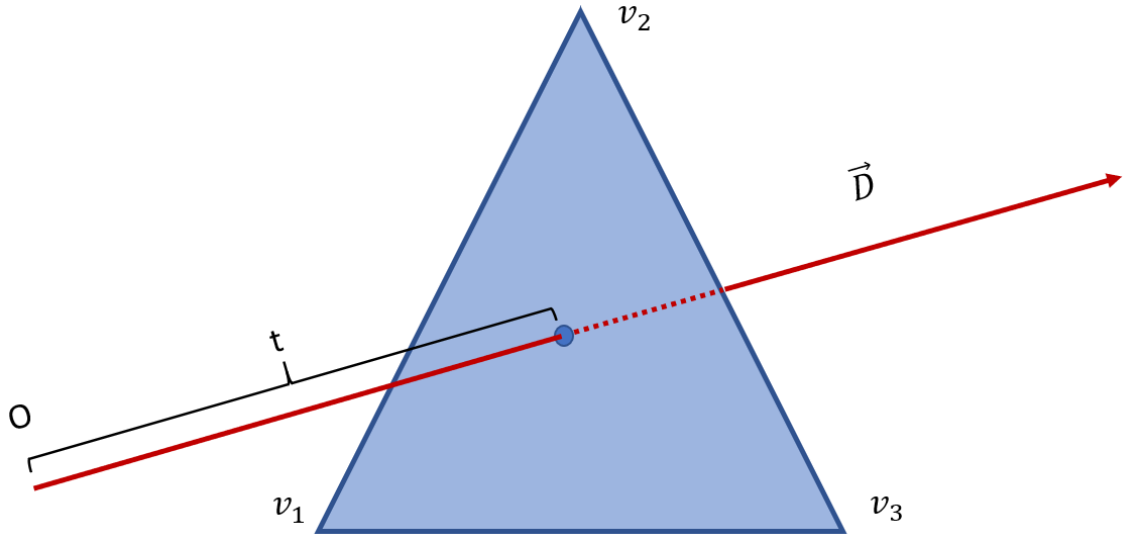
Když paprsky světla dopadají na nějaký povrch, některé paprsky povrch pohltí a jiné odrazí. Odražené paprsky mají nějakou vlnovou délku, která určuje barvu tohoto světla. Dále tyto paprsky mohou dopadnout na sítnici lidského oka, a tak si mozek vytvoří obraz okolního světa.

A na podobném principu funguje metoda raytracing. Akorát trochu opačně. Raytracing nevysílá paprsky ze zdroje světla, nýbrž sleduje z oka pozorovatele. Metoda vyšle paprsek skrz každý pixel obrazovky do scény a kontroluje, zda paprsek neprotíná nějaké geometrické primitivum (zpravidla trojúhelník)(Obr. 2).

Pokud paprsek neprotíná žádný objekt, vybarví pixel barvou pozadí. Pokud paprsek nějaké primitivum protne, spočte přesný nejbližší průsečík. Výsledný obraz poté záleží na způsobu stínovaní (tzv. „shading“). Může pixel rovnou vybarvit nějakou barvou, případně může z tohoto průsečíku vyslat další paprsky a spočíst odrazy, stíny nebo při pokročilejších metodách simulovat šíření světla a konvergovat k přesnému množství dopadajícího světla.

Algoritmus pro výpočet průsečíku paprsku a trojúhelníku

Pro výpočet průsečíku paprsku s trojúhelníkem existuje vícero algoritmů.[5] Asi nejznámější je Möller-–Trumbore algoritmus pojmenovaný po svých autorech. Protože výpočet průsečíku paprsku s trojúhelníkem je velmi důležitá součást raytracingu, bude algoritmus níže podrobně popsán.



Obrázek 2: Průsečík paprsku a trojúhelníku [1]

Ještě před popisem samotného algoritmu je třeba zmínit **Barycentrické souřadnice**, které algoritmus pro výpočet průsečíku používá. Barycentrické souřadnice jsou klíčovou součástí algoritmu, neboť pomocí nich lze popsat jakýkoliv bod v trojúhelníku. Včetně vrcholů a hran. Rovnice bodu v trojúhelníku popsaného barycentrickými souřadnicemi vypadá takto:

$$P = wA + uB + vC \tag{1}$$

Kde P je požadovaný bod, A, B a C jsou vrcholy trojúhelníku a skaláry u, v, a w jsou barycentrické souřadnice. Souřadnice splňují podmínku $u + v + w = 1$. Z podmínky plyne, že ze dvou souřadnic lze odvodit třetí $w = 1 - u - v$. Bod je uvnitř trojúhelníku, jestliže platí nerovnost $0 \leq u, v, w \leq 1$.

Z předchozích rovnic lze dosadit:

$$P = (1 - u - v)A + uB + vC \tag{2}$$

Po roznásobení a vytknutí:

$$P = A - uA - vA + uB + vC \tag{3}$$
$$P = A + u(B - A) + v(C - A) \tag{4}$$

Stojí za povšimnutí, že $(B-A)$ a $(C-A)$ jsou hrany AB a AC trojúhelníku ABC. Bod P lze také vyjádřit parametrickou rovnicí paprsku:

$$P = O + tD \tag{5}$$

Kde O je počátek paprsku a D jeho směrový vektor. Parametr t je vzdálenost bodu od počátku. Po dosazení do původní rovnice:

$$O + tD = A + u(B - A) + v(C - A) \tag{6}$$
$$O - A = -tD + u(B - A) + v(C - A) \tag{7}$$

Na pravé straně rovnice se nyní vyskytují tři neznámé t, u, a v vynásobené třemi známými. Rovnici lze přepsat na maticové násobení:

$$\begin{pmatrix} -D_x & (B-A)_x & (C-A)_x \\ -D_y & (B-A)_y & (C-A)_y \\ -D_z & (B-A)_z & (C-A)_z \end{pmatrix} \times \begin{pmatrix} t \\ u \\ v \end{pmatrix} = \begin{pmatrix} (O-A)_x \\ (O-A)_y \\ (O-A)_z \end{pmatrix} \tag{8}$$

Nyní je třeba nalézt řešení vzniklé soustavy rovnic. Algoritmus k tomu využívá Cramerovo pravidlo.[6] Pravidlo je založeno na výpočtu diskriminantu matice. Pro matici $\mathbf{A}$ a vektor pravých stran B platí, že matice $\mathbf{A}_i$ vznikne nahrazením i-tého sloupce matice vektorem B. Jednotlivé složky řešení $X = (x_1, ..., x_n)^T$ jsou dány vztahem:

$$x_i = \frac{\det \mathbf{A}_i}{\det \mathbf{A}} \tag{9}$$

V n-rozměrném prostoru je tato metoda výpočetně náročná. Ve třech rozměrech se tato metoda dá uplatnit díky **smíšenému součinu** tří vektorů A, B a C, který definujeme jako:

$$A \cdot (B \times C) \tag{10}$$

Přeznačením $E_1 = B - A$, $E_2 = C - A$ a $T = O - A$ řešení předchozí soustavy je:

$$t = \frac{\begin{vmatrix} T_x & E_{1_x} & E_{2_x} \\ T_y & E_{1_y} & E_{2_y} \\ T_z & E_{1_z} & E_{2_z} \end{vmatrix}}{\begin{vmatrix} -D_x & E_{1_x} & E_{2_x} \\ -D_y & E_{1_y} & E_{2_y} \\ -D_z & E_{1_z} & E_{2_z} \end{vmatrix}} \tag{11}$$

$$u = \frac{\begin{vmatrix} -D_x & T_x & E_{2_x} \\ -D_y & T_y & E_{2_y} \\ -D_z & T_z & E_{2_z} \end{vmatrix}}{\begin{vmatrix} -D_x & E_{1_x} & E_{2_x} \\ -D_y & E_{1_y} & E_{2_y} \\ -D_z & E_{1_z} & E_{2_z} \end{vmatrix}} \tag{12}$$

$$v = \frac{\begin{vmatrix} -D_x & E_{1_x} & T_x \\ -D_y & E_{1_y} & T_y \\ -D_z & E_{1_z} & T_z \end{vmatrix}}{\begin{vmatrix} -D_x & E_{1_x} & E_{2_x} \\ -D_y & E_{1_y} & E_{2_y} \\ -D_z & E_{1_z} & E_{2_z} \end{vmatrix}} \tag{13}$$

$$\tag{14}$$

$$\tag{15}$$

U smíšeného součinu lze zaměnit operátory. Prohození operandů mění znaménko:

$$A \cdot (B \times C) = -(A \times C) \cdot B = -(C \times B) \cdot A \tag{16}$$

Substitucí $M = (D \times E_2)$ a $N = (T \times E_1)$ z toho plyne:

$$t = \frac{N \cdot E_2}{M \cdot E_1} \tag{17}$$

$$u = \frac{M \cdot T}{M \cdot E_1} \tag{18}$$

$$v = \frac{D \cdot N}{M \cdot E_1} \tag{19}$$

Pokud $0 \leq u \leq 1$ a zároveň $0 \leq v \leq 1$, potom paprsek protíná trojúhelník. Lze takto spočítat průsečík trojúhelníku a paprsku prakticky jen pomocí skalárního a vektorového součinu.[7]

1.3 Rasterizace versus Raytracing

Obrovskou předností rasterizace je rychlost. Obecný proces rasterizace rychlý není. Metoda rasterizace je rychlá díky tomu, že probíhá paralelně na grafickém akcelerátoru a je založena na jednoduchých geometrických operacích, které jsou navíc prováděny na specializovaném hardwaru uzpůsobeném právě pro tyto výpočty. Díky tomu má tato metoda využití hlavně v aplikacích běžících v reálném čase, kdy na aplikaci klademe nárok, aby se obraz obnovoval frekvencí alespoň 60 snímků za sekundu. Díky tomu oko vnímá obraz plynule a zároveň i ovládání je dostatečně responzivní.

Způsob, jakým je obraz touto metodou generován s sebou přináší i různé problémy. Tato metoda má například problém vykreslit různé světelné efekty fotorealistickým způsobem. Existují algoritmy, které se snaží fotorealistické zobrazení napodobit, nicméně nedosahují tak dobrých výsledků jako simulace světla pomocí raytracingu.

Raytracing má v tomto případě přesně opačný problém. Metoda dokáže scénu zobrazit velmi fotorealisticky. Nicméně aby nebyl obraz zrnitý, musí se do scény poslat velké množství paprsků. Zároveň scéna může obsahovat miliony trojúhelníků a s každým jednotlivým trojúhelníkem se musí počítat průsečík s paprskem a najít ten nejbližší, což je obrovská výpočetní zátěž.

1.3.1 Stíny

Rasterizace si příliš neumí poradit s měkkými stíny, které vrhají objekty osvětlované plošným zdrojem světla. Existují různé algoritmy, které navodí jakousi iluzi měkkého stínu, nicméně do reálného měkkého stínu to má stále daleko.

Asi nejpopulárnějším rasterizačním algoritmem pro výpočet stínů je metoda vykreslování ve dvou krocích pomocí tzv. „shadow mapy“. Nejprve se scéna vykreslí z pozice zdroje světla. Vše, co zdroj světla vidí se považuje za osvětlené. Vzdálenost těchto osvětlených bodů se uloží do jednotlivých pixelů výše zmíněné shadow mapy. Poté se scéna klasicky vykreslí z pohledu kamery. Scéna se následně rasterizuje, tedy každému pixelu náleží nějaký bod ve scéně. Transformačními maticemi lze poté spočítat, jakému pixelu v shadow mapě tento bod náleží. Poté se porovná vzdálenost bodu s hodnotou uloženou v shadow mapě, a tím se zjistí, zda se bod nachází ve stínu či nikoliv. Tento způsob vykreslování stínu s sebou nese ještě různé vizuální artefakty, které se musí dodatečně korigovat.

Raytracing vykresluje stíny tak, že se vyšle do scény paprsek a hledá se průsečík s objektem. Jakmile je průsečík nalezen, vyšle se z něj další paprsek směrem ke zdroji světla. Pokud paprsek zdroj světla zasáhne, znamená to, že průsečík je osvětlen. Pokud

mezi průsečíkem a zdrojem světla nalezne další průsečík, znamená to, že mezí aktuální pozicí a zdrojem světla je překážka a tento bod se nachází ve stínu.

Výše popsaný algoritmus by sám o sobě produkoval ostré (tvrdé) stíny stejně jako rasterizace. Pokud je však zdroj světla plošný, můžeme z průsečíku poslat paprsků více s náhodným rozptylem a spočítat, kolik paprsků zdroj světla zasáhne a kolik ne. Počet neúspěšných paprsků pak určuje intenzitu stínu a výsledný obraz bude obsahovat měkké stíny.

1.3.2 Globální osvětlení

Pojem globální osvětlení obvykle označuje množinu algoritmů, které nejenže zahrnují postupy přímého osvětlení scény (nasvětlení scény vzniká pouze z přímého dopadu světla), ale také osvětlení nepřímého (nasvětlení scény bere v úvahu nejen paprsky přímo ze světelného zdroje, nýbrž i světelné paprsky odražené od ostatních povrchů).

Raytracing s vykreslováním nepřímého osvětlení nemá problém již z podstaty fungování samotného algoritmu. Po vystřelení primárního paprsku do scény se nalezne průsečík s objektem a z tohoto průsečíku se vysílají další paprsky do různých směrů. Na základě materiálů jednotlivých objektů lze pak nepřímé osvětlení dopočítat. V závislosti na počtu objektů a počtu paprsků pak ale tato metoda vyžaduje velký výpočetní výkon.

Rasterizace na tom je poněkud hůř. Samotný název rasterizace dává napovědět, že prvotní záměr tohoto algoritmu zřejmě nebyl fyzikální simulace světla v prostoru, ale projekce 3D světa na 2D plátno.

Z tohoto důvodu vznikla spousta algoritmů, jak se co nejvíce vizuálně přiblížit fotorealistickému ideálu, ale stále to není tak dobré jako samotný raytracing.

Jednou z metod je například tzv. „lightmapping“. Tato metoda předpočítá nasvětlení scény (třeba právě pomocí raytracingu) dopředu a uloží do textury. Tato textura se poté aplikuje na scénu. Protože se textura počítá offline, nevýhodou této metody je, že ji lze aplikovat pouze na statické scény. Nelze v reálném čase měnit zdroje světla nebo geometrii.

Z předchozích řádků lze nahlédnout, že je-li potřeba vykreslit fotorealistický obraz, je raytracing vhodnější než rasterizace. Nicméně, jak už bylo několikrát řečeno, tato metoda je velice pomalá a vyžaduje značný výpočetní výkon. Aby vykreslování mohlo probíhat v reálném čase, je potřeba tuto metodu nějakým způsobem optimalizovat.

Časově nejdražší operace při raytracingu je počítání průsečíků. Čím více objektů, tím více průsečíků musí algoritmus spočítat. Proto první optimalizací, která se nabízí, je redukce počtu této operace. K tomu slouží akcelerační struktury, které buďto dělí prostor, nebo seskupují jednotlivá geometrická primitiva do větších celků. V závislosti na použité struktuře pak algoritmus buďto osekává prostor dokud se nedostane k jednotlivým primitivům, nebo se zanořuje do hierarchie seskupených celků.

Další „optimalizací“, nebo lépe řečeno urychlení výpočtu, se nabízí využití moderního výpočetního hardwaru. Jak se výkon grafických karet zvyšoval a technologie modernizovala, začaly se grafické akcelerátory využívat i k obecnějším výpočtům než jen k pevně daným geometrickým manipulacím grafického řetězce.

2 CUDA

V roce 2006 společnost NVIDIA představila framework zvaný CUDA.[8] Tato softwarová vrstva rozšiřuje stávající programovací jazyky jako je Python, Fortran nebo C++ a umožňuje spouštět programy napsané v těchto jazycích na GPU. Výhodou tohoto frameworku je fakt, že není závislý na žádném dalším grafickém API jako je třeba Direct3D nebo OpenGL.

2.1 Kernel

Základním konceptem tohoto programovacího modelu je rozšíření jazyka C++ způsobem, že programátor může definovat funkci, nazývanou „kernel“, která v případě volání není vykonána jednou na CPU jako je tomu normálně, ale vykoná se paralelně N-krát na N různých CUDA vláknech.

Kernel je definován klíčovým slovem `__global__`. Při volání kernelu potřebujeme určit na kolika vláknech a v kolika blocích se spustí. Pro to přidala CUDA do jazyka C++ novou syntaxi, která vypadá takto: `<<<pocetBlokuVMrizce,pocetVlaken>>>`.

```
// Definice kernelu
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Volani kernelu s N vlakny v jednom bloku
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Kód 1: Ukázka definice kernelu a jeho volání

2.2 Hierarchie vláken

Každému spuštěnému vláknu je přiřazeno unikátní identifikační číslo, ke kterému lze přistoupit uvnitř kernelu pomocí vnitřní proměnné `threadIdx`. Ve skutečnosti proměnná `threadIdx` je třísložkový vektor, což dovoluje identifikovat konkrétní vlákno ve skupině vláken uspořádaných do jednoho, dvou nebo tří rozměrů. Taková skupina vláken se

nazývá „thread block“(Dále „blok“). Díky tomu můžeme přirozeně spouštět výpočty přes vícerozměrné objekty jako jsou třeba matice, vícerozměrná pole apod.

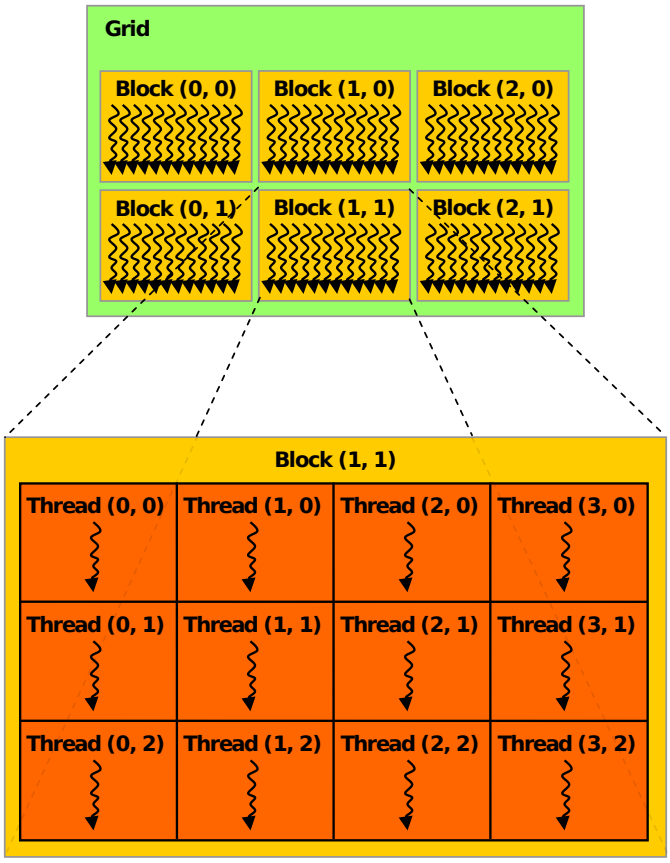
```
// Definice kernelu
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Volani kernelu s jednim blokem vlaken
    // zarovnanych do dvou rozmeru o velikosti N * N
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Kód 2: Ukázka kernelu s dvojrozměrným zarovnáním vláken

Z důvodu hardwarových omezení počet vláken v bloku může být pouze omezené množství. Na dnešních moderních grafických kartách je to 1024 vláken. Nicméně kernel může být vykonáván ve vícero blocích o stejné velikosti. Celkový počet běžících vláken je tedy počet vláken v bloku vynásobený počtem bloků. (Rovnice 20)

$$vlaknaCelkove = vlakenPerBlok * pocetBloku \tag{20}$$

Bloky jsou uspořádávány do jedno, dvou nebo třírozměrné „mřížky“(Obr. 3). Počet bloků v mřížce je zpravidla určen velikostí vstupních dat, což typicky přesáhne počet procesorových jader. Každý blok v mřížce, podobně jako vlákno v bloku, lze také identifikovat vnitřní proměnnou - `blockIdx`. Velikost jednoho bloku (tedy počet vláken v bloku) je přístupná přes vnitřní proměnnou `blockDim`. Stejně jako `threadIdx` tak i tyto výše zmíněné proměnné jsou třísložkové vektory. Finální kód jak indexovat konkrétní vlákno v nějakém bloku v nějaké mřížce může vypadat například takto(Kód 3):



Obrázek 3: Uspořádání vláken do bloků a mřížky

```
// Definice kernelu
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
    ...

    int threadIdx = blockIdx.x * blockDim.x + threadIdx.x;

    ...
}
```

Kód 3: Ukázka indexace vlákna

Ideální paralelní algoritmus je takový, že nevyužívá žádné sdílené prostředky a pracuje zcela disjunktně. V reálných podmínkách se nelze vyhnout občasným přístupům do sdílené paměti. Framework CUDA pro tyto případy poskytuje dvě možnosti synchronizace.

První synchronizační pomůcka je na úrovni bloku vláken. V kernelu se dá použít funkce `__syncthreads()`, která funguje jako bariéra. Dokud k této bariéře nedorazí všechny vlákna **v jednom bloku**, výpočet nepokračuje dál. Při použití této funkce je důležité dát si pozor, kam je umístěna. Pokud by byla umístěna do jedné z `if-else` větví, může nastat deadlock[1], protože některé z vláken se může vydat druhou větví a k bariéře se vůbec nedostane.

Druhou synchronizační metodou jsou atomické operace. Znakem těchto operací je tzv. „nedělitelnost“. To znamená, že pokud se na nějakém kusu paměti provádí atomická operace, nelze k této paměti přistupovat z jiného vlákna. Pakliže chtějí dvě atomické operace manipulovat s jedním paměťovým prostorem, je zajištěno, že se operace zařadí za sebe a provedou se sekvenčně. Což v paralelním prostředí, kde ke kusu paměti mohou najednou přistupovat desítky a stovky vláken znamená snížení výkonu.

2.3 Hierarchie pamětí

CUDA vlákno může přistupovat k datům v různých adresních prostorech. Předně každé vlákno má svoji lokální paměť, kterou může číst jenom dané vlákno. Do ní se ukládají například lokální proměnné.

Dále má vlákno přístup do tzv. „sdílené paměti“. Tato paměť je sdílená pouze mezi vlákny v jednom bloku. Což tedy znamená, že vlákno z jednoho bloku nemůže jakkoli číst nebo zapisovat do sdílené paměti jiného bloku. Protože na jedno místo v paměti může najednou přistupovat více vláken, je třeba dbát na to, aby si vlákna nepřepisovala data pod rukama a zajistit výlučný přístup. Sdílenou paměť lze alokovat buďto staticky pomocí klíčového slova `__shared__` uvnitř kernelu, nebo dynamicky zadáním velikosti paměti jako třetí parametr ve volání kernelu.

```
// Definice kernelu
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    ...
    // staticka alokace sdilene pameti
     __shared__ int s[64];
    ...
}

int main()
{
    ...
    // treti parametr mezi trojitymi zavorkami
    // je dynamicka alokace sdilene pameti
    MatAdd<<<numBlocks, threadsPerBlock, DynamicSharedMemorySize>>>(A, B);
    ...
}
```

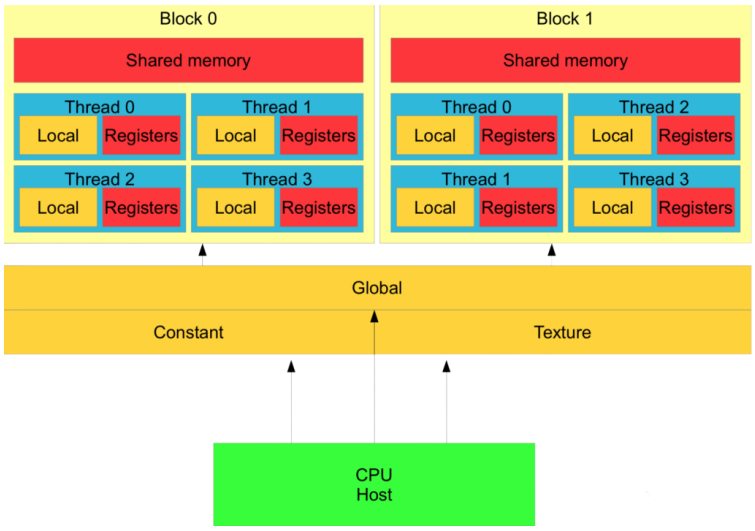
Kód 4: Ukázka statické a dynamické alokace sdílené paměti

[1] zamrznutí programu z důvodu vzájemného čekání N vláken

Třetí paměťový prostor, ke kterému může vlákno přistupovat je „globální paměť“. Do globální paměti mohou přistupovat všechna vlákna bez ohledu na to, do jakého bloku náleží. Globální paměť, stejně jako sdílená, může být alokována staticky, nebo dynamicky. V programovacím jazyku C se dynamická paměť alokuje pomocí funkce `malloc`. V CUDA frameworku je to analogicky a paměť lze dynamicky alokovat funkcí `cudaMalloc`. Staticky se paměť alokuje opět obdobně jako v C/C++ deklarací proměnné v globálním rámci. Pouze se před proměnnou dá klíčové slovo `__device__`.[9]

```
// staticka alokace v globalni pameti
__device__ int globalArray[256];

// neni zde klicove slovo __global__, nejedna se tedy o kernel,
// ale klasickou CPU funkci
void foo()
{
    ...
    // dynamicka alokace v globalni pameti
    int *myDeviceMemory = 0;
    cudaError_t result = cudaMalloc(&myDeviceMemory, 256 * sizeof(int));
    ...
}
```

Kód 5: Ukázka statické a dynamické alokace globální paměti



Obrázek 4: CUDA hierarchie pamětí [2]

2.4 Thrust

Thrust je C++ knihovna šablonových paralelních algoritmů a datových struktur. Do jisté míry je to paralelní kopie standardní C++ knihovny STL. Jedná se o další vrstvu abstrakce, jak programátorovi usnadnit implementaci paralelních algoritmů a částečně odstínit od nízkoúrovňového CUDA kódu. Všechny funkce v knihovně se volají z CPU,

takže programátor ve skutečnosti nemusí napsat ani řádku paralelního GPU kódu. Knihovna obsahuje funkce jako například třídění, hledání unikátní prvků, minima/maxima v poli, manipulace s poli na základě klíčů, transformace dat apod.[10] Některé z těchto funkcí používá i raytracer této práce. Dané funkce jsou blíže popsány v kapitole **Implementace**.

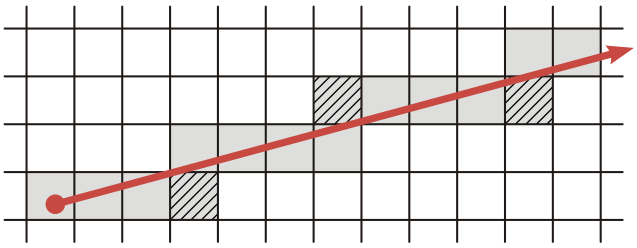
3 Akcelerační struktury pro raytracing

Základním požadavkem syntézy obrazu je fotorealističnost(pokud není umělecký záměr jiný). Což je náročná operace trvající spoustu času. Což při „offline renderingu“ není zas tak palčivý problém, protože u tohoto procesu trvá vykreslit jeden obraz třeba i několik hodin. I přesto je ale dobré tento proces urychlit, protože když se obraz spočítá rychleji, zbývá více času i na ostatní výpočty. Dvojnásob toto platí u realtime aplikací, kde je obvykle požadavek, aby byla snímková frekvence alespoň 60 FPS. Z čehož vyplývá, že na všechny výpočty v jednom snímku má aplikace zhruba 16 milisekund.

Pokud aplikace používá metodu sledování paprsku, klíčový způsob jak urychlit tento proces, je snížit počet průsečíků. A to lze právě akceleračními strukturami.

3.1 Mřížka

Mřížka spadá do kategorie urychlujících struktur dělící prostor. Mřížka rovnoměrně rozděluje prostor na trojrozměrné buňky. Každá buňka v sobě odkazuje všechny objekty, které přesahují do jejího objemu. Díky rozdělení prostoru není třeba počítat průsečíky se všemi objekty ve scéně, ale pouze s buňkami, které paprsek vyslaný do scény protíná. Poté se počítají průsečíky se všemi objekty, které jsou v protnutých buňkách odkazovány. Nevýhodou této struktury je, že jeden objekt může být odkazován z vícero buněk, takže se zbytečně počítá vícekrát průsečík s jedním a tím samým objektem. Také je potřeba vhodně zvolit velikost jednotlivých buněk. Příliš velké buňky budou obsahovat mnoho trojúhelníků a počet průsečíků není příliš zredukován. Naopak příliš malé buňky mohou být menší než referencované objekty, tudíž jeden objekt může být odkazován z velkého množství buněk a bude tím závratně narůstat režie.[11]



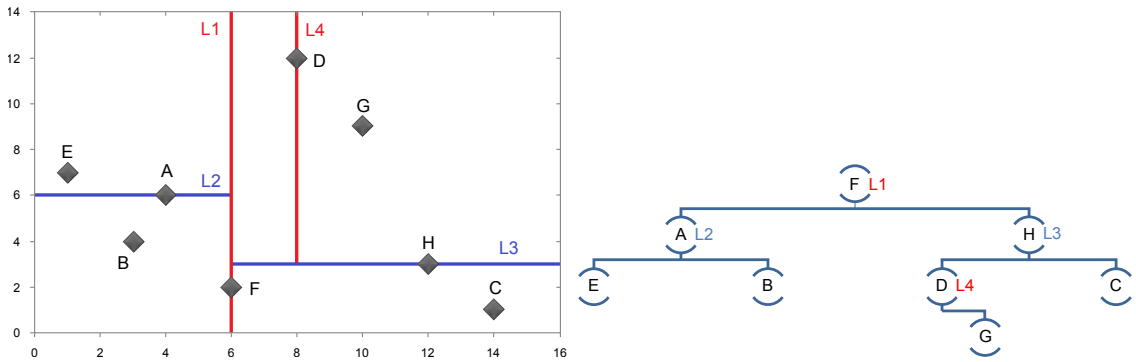
Obrázek 5: Raytracing v mřížce. V každé buňce je reference na objekty, které zasahují do buňky. Paprsek projde všechny buňky, do kterých zasahuje.

3.2 KD-Strom

KD-strom je hierarchická struktura, která rekurzivně dělí prostor rovinami kolmými na osy souřadného systému. Každý vnitřní uzel stromu obsahuje dělicí rovinu, která rozdělí prostor na dvě disjunktní části. Tyto dvě části jsou odkazovány jako levý a pravý potomek rodičovského uzlu. Geometrická primitiva jsou distribuovány mezi oba potomky. Objekty, které leží na dělicí rovině, jsou přiřazeny do obou potomků. Samotná geometrická primitiva jsou odkazovány až z listů vygenerovaného stromu. [12]

Při způsobu, jakým KD-strom funguje, je pro jeho výkon a efektivitu naprosto zásadní pozice dělicích rovin. Známou heuristikou je tzv. SAH (surface area heuristic). V tomto případě se vypočítá plocha potenciálních obalových těles, případně se započítají další kritéria a určí se tak hodnota, která je cenou potenciálního rozdělení. Cílem je umístit dělicí rovinu a rozdělit uzel tak, aby byla cena co nejnižší. [13]

Procházení struktury je pak celkem přímočaré. Při hledání průsečíku paprsku s primitivy v KD-stromě se postupuje rekurzivně. Nejdříve se zkontroluje průsečík s ohraničujícím kvádrem kořene. Ve vnitřním uzlu se zkontroluje průsečík s dělicí rovinou a podle toho se rozhodne, zda paprsek protíná pravého či levého potomka. Pokud protíná oba, navštíví se nejdřív ten, který je blíž počátku paprsku. Pokud procházený uzel je list, zkontroluje se průsečík se všemi primitivy, která k němu patří. [14]



Obrázek 6: KD-Strom

3.3 Hierarchie ohraničujících obálek

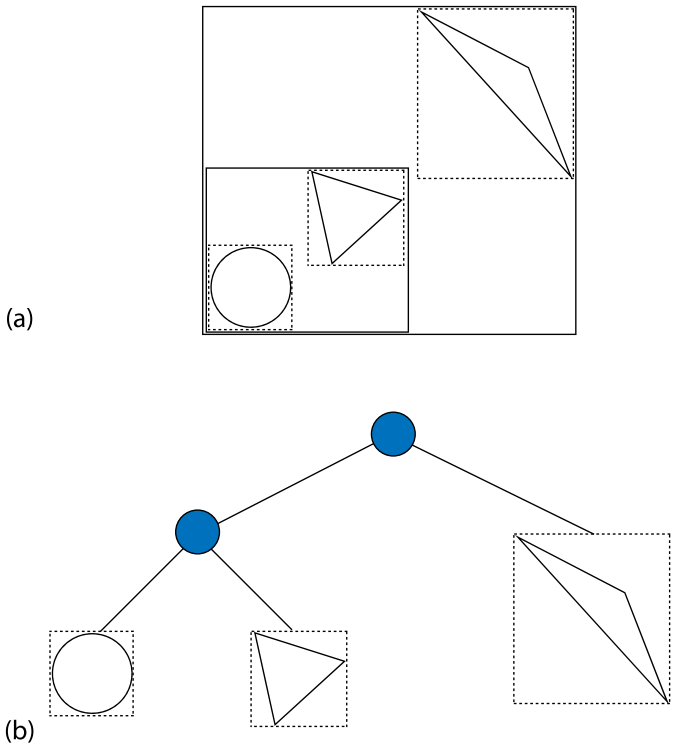
Hierarchie ohraničujících obálek (dále pod anglickou zkratkou „BVH - Bounding volume hierarchy“) tentokrát nerozděluje prostor, ale dělí objekty, případně seskupuje objekty. Záleží, zda se zvolí stavba shora dolů, nebo zdola nahoru.

Stavba shora dolů znamená, že se nejprve spočítá ohraničující obálka(dále pod anglickou zkratkou „AABB - Axis Aligned Bounding Box“), která zahrnuje všechny objekty ve scéně.

Následně se primitiva rozdělí do dvou skupin a každá by měla mít co nejmenší AABB. Takto rekurzivně se pokračuje, dokud potomek nebude obsahovat pouze jeden objekt. Tato metoda je snadnější na implementaci, rychlejší na konstrukci, ale nedává nejlepší možné stromy. Pro optimální dělení se hledá minimum tzv. „účelové funkce“. Metoda shora dolů, přestože může nalézt lokální minima v každém vnitřním uzlu, neznamená to, že celá struktura BVH bude optimální. Stavba zdola nahoru v tomto může podat lepší výsledky.[15][16]

Stavba zdola nahoru postupuje opačným způsobem. Nejprve spočítá AABB pro každé geometrické primitivum ve scéně. Pak postupně seskupuje jednotlivé objekty a zároveň sjednocuje i jejich obálky. Algoritmus sjednocuje tak dlouho, dokud nedorazí do kořene stromu, kde jsou seskupeny všechny objekty a spočtena obálka, která zahrnuje všechny objekty. Tato metoda je náročnější na implementaci, ale dává lepší výsledky než metoda předchozí. [17]

Při hledání průsečíku paprsku se postupuje rekurzivně. Nejdříve se zkontroluje, zda paprsek protíná kvádr kořene. Pokud ano, podívá se, zda protíná kvádry synů. Pokud opět ano, hledá se průsečík stejným způsobem na nižších úrovních. Pokud paprsek protíná list, proiteruje se skrz všechna primitiva, co k němu patří. Pokud se při hledání najde průsečík s primitivem v jednom podstromu nějakého uzlu, stále může existovat průsečík ve druhém podstromu daného uzlu, který je blíž. Proto je obecně potřeba prohledat oba podstromy. [14]

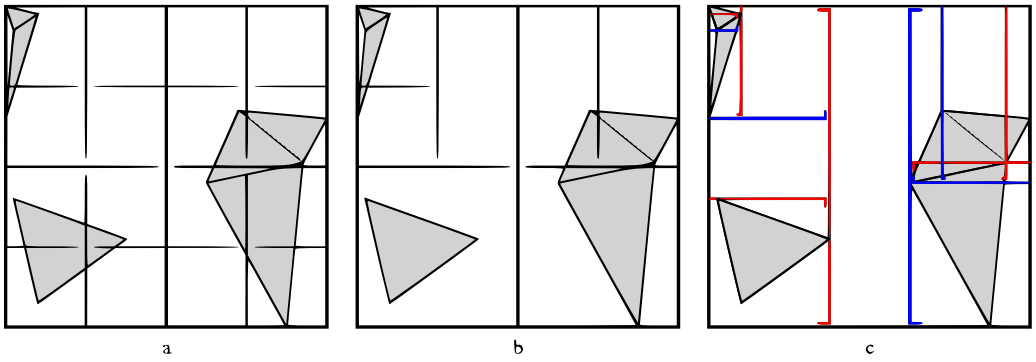


Obrázek 7: Hierarchie ohraničujících obálek (BVH)

3.4 Hierarchie ohraničujících intervalů

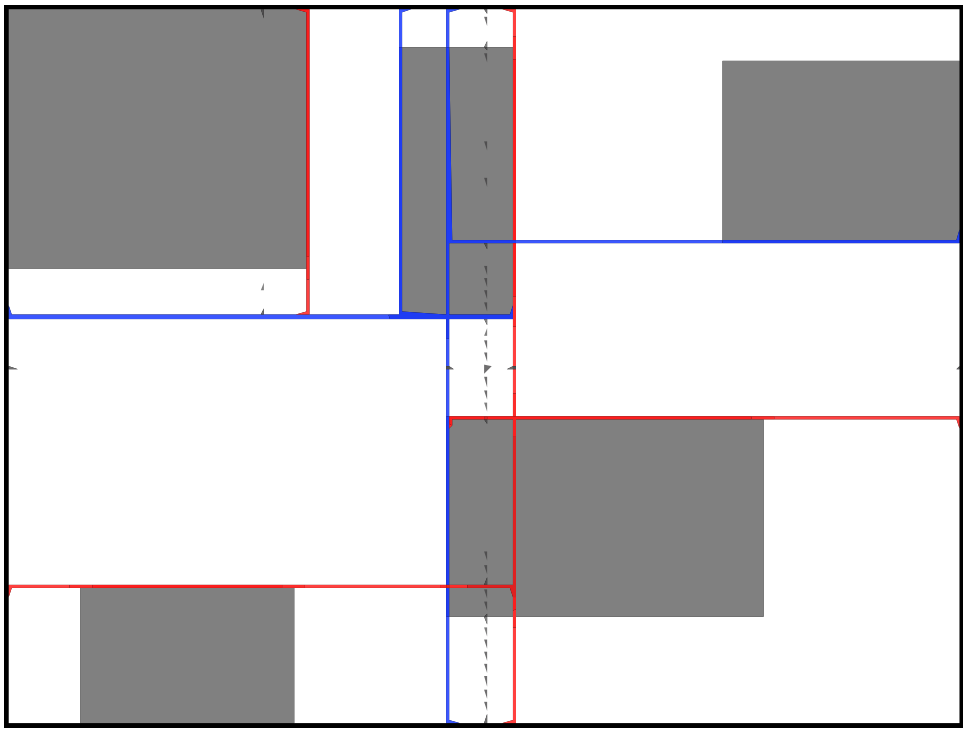
Hierarchie ohraničujících intervalů (dále pod anglickou zkratkou „BIH - Bounding interval hierarchy“) je jakási hybridní struktura, která se snaží využívat výhod obou předchozích přístupů.

Stavba této struktury probíhá rekurzivně podobně jako u KD-stromu. U každého vnitřního uzlu se spočítá délka všech tří os uvnitř AABB daného uzlu a dělicí rovina se umístí doprostřed nejdelší osy. Každý vnitřní uzel má na vstupu množinu geometrických primitiv. Pro tyto primitiva se spočítá AABB a jeho střed. U každého středu se rozhodne, zda leží nalevo či napravo od dělicí osy. Tím se primitiva roztřídí na dvě části dle dělicí roviny kolmé na jednu z os souřadného systému.



Obrázek 8: Průběh výběru dělicí roviny a pak určení finálních dvou. **a)** Kandidáti na zvolení dělicí roviny. **b)** Zvolení dělicí roviny na základě nejdelší osy. **c)** Spočtení finálních rovin levých a pravých potomků.

Poté, co jsou objekty roztříděny, nastává v této struktuře zásadní rozdíl oproti předchozím dvěma strukturám. Místo celého AABB nebo dělicí roviny se do vnitřního uzlu uloží dvě roviny. Jedna ohraničující objekty v levém potomkovi a druhá ohraničující ty v pravém. Levá rovina odpovídá maximální hodnotě souřadnice příslušné osy ze všech objektů v levém potomkovi. Pravá rovina odpovídá minimu objektům v pravém potomkovi. Tento proces se rekurzivně opakuje v každém vnitřním uzlu, dokud nezbude jeden objekt a ten se vloží do listu. Případně lze předem zvolit hloubku stromu a konstrukci ukončit dříve a do listu vložit všechna zbylá primitiva.



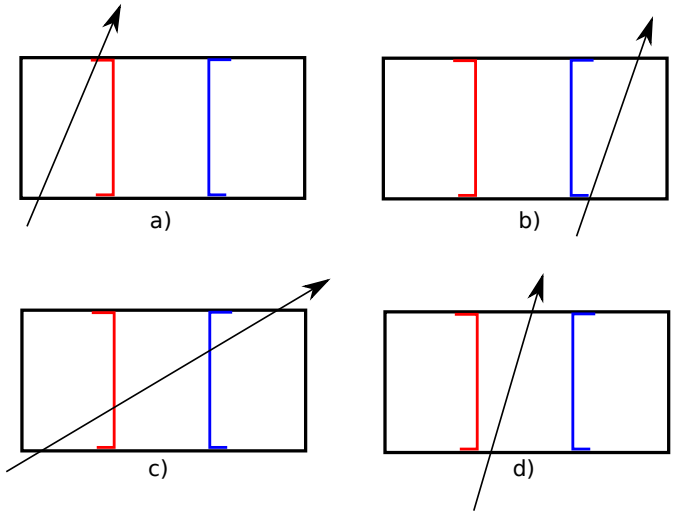
Obrázek 9: Ilustrace hierarchie ohraničujících intervalů (BIH) ve 2D. Barevné přímky znázorňují výsledné dělicí roviny. Šedé útvary symbolizují geometrická primitiva ve scéně. Čárkované linky zobrazují původní dělicí čáry, dle kterých se třídí primitiva.

Samotná konstrukce je dost podobná třídění, jehož struktura je totožná s řadícím algoritmem quicksort. Tudíž složitost konstrukce je průměrně $\mathcal{O}(n \log n)$.

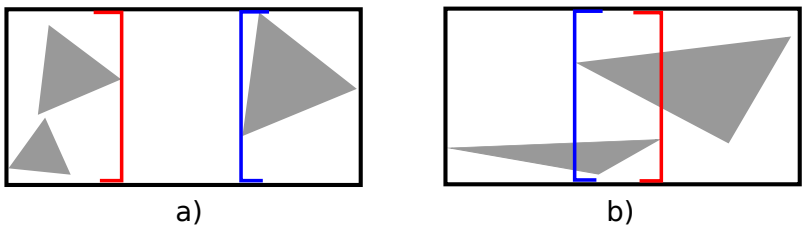
Výhoda dvou dělicích rovin místo jedné, jako je to u KD-stromu je v tom, že se zde nemůže stát, že by nějaký objekt byl vícekrát odkazován z více potomků. Každý objekt je buď vlevo, nebo vpravo.

Průchod strukturou je pak takřka analogický s průchodem KD-stromu až na pár výjimek. Místo průsečíku s jednou rovinou se počítá průsečík s dvěma rovinami. A kromě situací, kdy paprsek protíná buď pouze levého potomka nebo pouze pravého potomka nebo oba najednou, může nastat ještě čtvrtá situace. Tato situace nastane, když se roviny nepřekrývají. Potom mezi nimi vznikne prostor, kudy může paprsek projít a neprotne ani jednoho potomka. Díky tomu lze u řídkých scén přeskočit prázdné uzly a pokračovat dalšími.

Tím, že se v celém algoritmu používají pouze matematické operace minima a maxima, zajišťuje algoritmus numerickou stabilitu. Navíc tím, že v této struktuře se neobjevují vícenásobné reference, lze již dopředu odhadnout paměťovou náročnost struktury. Lze tedy paměť alokovat dopředu a vyhnout se výkonovým ztrátám kvůli fragmentaci paměti způsobené dynamickou alokací.[18]



Obrázek 10: Způsoby protnutí hierarchie paprskem. **a)** Protíná pouze levého potomka. **b)** Protíná pouze pravého potomka. **c)** Protíná oba potomky. **d)** Prochází mezi rovinami a neprotíná žádného potomka.



Obrázek 11: Ilustrace umístění dělicích rovin. Na obrázku **b)** je vidět, že se roviny mohou překrývat.

4 Cíl práce

První zmínka o struktuře BIH se objevila na grafickém symposiu Eurographics v létě roku 2006. Tedy v době, kdy obecné výpočty na grafické kartě se teprve začínaly rozvíjet a ještě ani nebyl oznámen framework CUDA (listopad 2006). Od té doby se mi nepodařilo dohledat, že by se touto urychlovací strukturou někdo zabýval nebo že by se objevila v nějakých odborných publikacích, přestože testovací scény v publikaci vykazují relativně dobré výsledky.

Z výše zmíněných výhod této struktury lze usoudit, že stojí za to prozkoumat, zda by šla tato struktura paralelizovat a implementovat na grafické kartě. Tím využít jak výhody dané struktury, tak masivního paralelního výpočetního výkonu, který za těch necelých dvacet let od publikace této struktury narostl mnohonásobně.

Cílem práce je prozkoumat možnosti implementace urychlující struktury Bounding Interval Hierarchy na grafické kartě. Změřit dobu konstrukce, dobu procházení strukturou a počet vykreslených snímků za sekundu pro potenciální použití v realtime aplikacích. Protože scény v realtime aplikacích zpravidla nebývají statické, kvůli často měnícím se pozicím trojúhelníků se bude muset struktura přebudovat každý snímek. Na rychlost přebudování se tedy klade velký důraz. Naměřené časy porovnat s původní implementací BIH i s ostatními urychlujícími strukturami.

5 Bounding interval hierarchy na GPU

5.1 Paralelizace

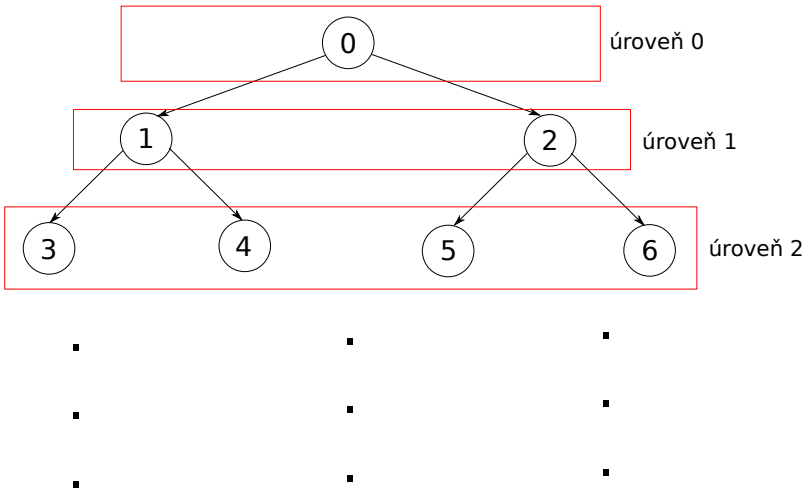
Celý algoritmus se skládá ze dvou částí – konstrukce struktury a její procházení. První věc, která se nabízí k paralelizaci původního algoritmu je procházení strukturou. Procházení stromem do struktury nezapisuje žádná data. Pouze je čte. Což znamená, že nemůže docházet k žádným nežádoucím přepisům během paralelního zpracování, kdy více CUDA vláken přistupuje na stejné místo v paměti.

Zároveň procházení strukturou je při renderování naprosto disjunktní proces a tím ideální pro paralelizaci. Sekvenční přístup funguje tak, že se postupně bere pixel po pixelu a do každého se vystřelí N paprsků. Každý paprsek poté hledá průsečík s urychlující strukturou. Když je nalezen, zanořuje se hlouběji do stromu struktury až dojde k listu a hledá průsečíky s objekty obsažené v listu.

K tomu, aby se mohl vyslat paprsek do následujícího pixelu a procházet strukturou, není potřeba znát žádné informace o předchozím pixelu nebo paprsku. Není mezi nimi žádná datová závislost. Triviálnější část paralelizace algoritmu je tedy na tomto místě.

Procházení už je paralelizováno. Nyní je ještě potřeba paralelizovat konstrukci. Samotná konstrukce BIH není na paralelizaci tak triviální jako procházení. Při sekvenčním algoritmu se struktura buduje od kořene stromu a pokračuje se do spodních úrovní až k listům. Vstupní primitiva u potomků jednotlivých uzlů stromu vznikají právě u rodiče. Je zde datová závislost, kdy bez hotového rodičovského uzlu nelze zpracovávat jeho potomky.

Jedná se ovšem o závislost tzv. „vertikální“ nikoli „horizontální“. Sice nelze zpracovat potomky bez zpracovaného rodiče, nicméně když už je rodič zpracovaný, všechny potomky na stejné stromové hladině zpracovávat paralelně lze.



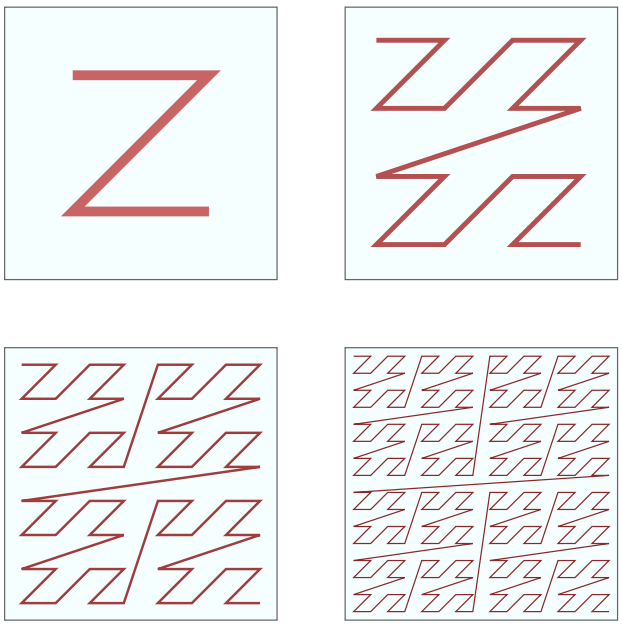
Obrázek 12: Vyznačení jednotlivých stromových hladin, které lze zpracovávat paralelně.

Tento styl paralelizace je technicky možný, ale má jednou zásadní slabinu. Tou je obsazenost a využití CUDA vláken. Vzhledem k tomu, že na grafické kartě jsou k dispozici stovky až tisíce vláken, je omezení paralelizace na malý počet vláken značně neefektivní. V prvních několika hladinách stromu poběží pouze jednotky vláken. Ve chvíli, kdy výpočet začne být efektivní a alespoň částečně se využije výpočetní potenciál grafického akcelerátoru, už je spousta času a výkonu nevyužita v horní části stromu. Toto není efektivní způsob paralelizace.

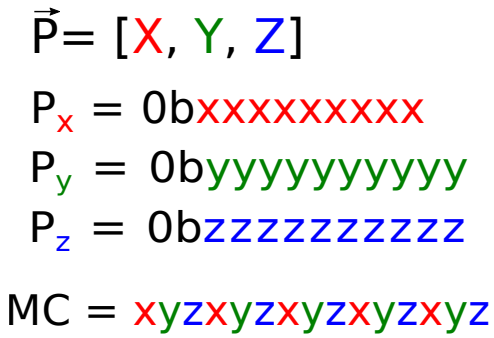
Aby bylo možné konstrukci struktury paralelizovat, je potřeba se zbavit datové závislosti potomků na rodičovském uzlu. Přestože se to na první pohled zdá jako nemožný úkol, existuje metoda jak to provést, a to za pomoci Mortonova rozkladu.

5.2 Mortonův rozklad (Z-křivka)

Mortonův rozklad (též Mortonova Z-křivka nebo pouze Z-křivka) je způsob kódování, který vícerozměrná data mapuje do jednoho rozměru. Lze vytvořit křivku, která vyplní celý vícerozměrný prostor, přičemž zachovává lokalitu jednotlivých dat. To znamená, že data, která byla blízko u sebe, jsou poblíž sebe i na křivce. Samotný kód se pak tvoří tak, že souřadnice datového bodu se převedou na binární reprezentaci a střídavě se prokládají bity jednotlivých souřadnic. Protože má kód nějaké konečné rozlišení, ve skutečnosti rozdělí prostor do pravidelné mřížky. Rozlišení této mřížky je dáno počtem bitů Mortonova kódu.



Obrázek 13: Ukázka čtyř iterací Z-křivky



Obrázek 14: Schéma tvorby Mortonova kódu

5.3 Plně paralelní konstrukce stromu

Aby bylo možné vnitřní uzel považovat za plně zpracovaný, musí algoritmus úspěšně provést tyto operace:

- Určit osu souřadného systému, dle které budeme třídit objekty v daném uzlu a na kterou budou kolmé finální hraniční roviny.
- Roztřídit objekty do levého a pravého potomka.
- Spočítat finální hraniční roviny.

K tomu lze použít právě Mortonův rozklad. Základ algoritmu je v očíslování vnitřních uzlů takovým způsobem, který umožní nalézt interval objektů náležících danému uzlu, aniž by bylo potřeba dalších informací o zbytku stromu. Dále algoritmus využívá vlastnosti binárních stromů, že jakýkoliv binární strom s N listy, má přesně $N-1$ vnitřních uzlů.

Algoritmus alokuje $N-1$ vnitřních uzlů a všechny je zpracuje paralelně. Poté v několika krocích každý vnitřní uzel zjistí, jaká podmnožina geometrických primitiv danému uzlu náleží. Přesný popis algoritmu hledání této podmnožiny bude rozebrán v další kapitole. Následně nalezenou podmnožinu roztřídí na základě Mortonových rozkladů na dvě části. Protože jsou Mortonovy rozklady všech objektů předem setříděny, jsou pro algoritmus při určování levých a pravých potomků ve skutečnosti zajímavé pouze dva kódy. Kódy, které spolu sousedí a liší se v Mortonově rozkladu v bitu na i-té pozici. A protože Mortonův rozklad jsou v principu za sebou poskládané souřadnice x, y a z (viz Obr. 14), jako vedlejší produkt je do vnitřního uzlu uložena požadovaná osa.[19]

Nyní už zbývá pouze spočítat v každém vnitřním uzlu levou a pravou hraniční rovinu. Algoritmus přistoupí ke všem objektům ve scéně a zjistí jejich AABB. Poté projde akcelerační strukturou a všem vnitřním uzlům aktualizuje obě roviny. Přesný postup je popsán v podkapitole 6.3.3.

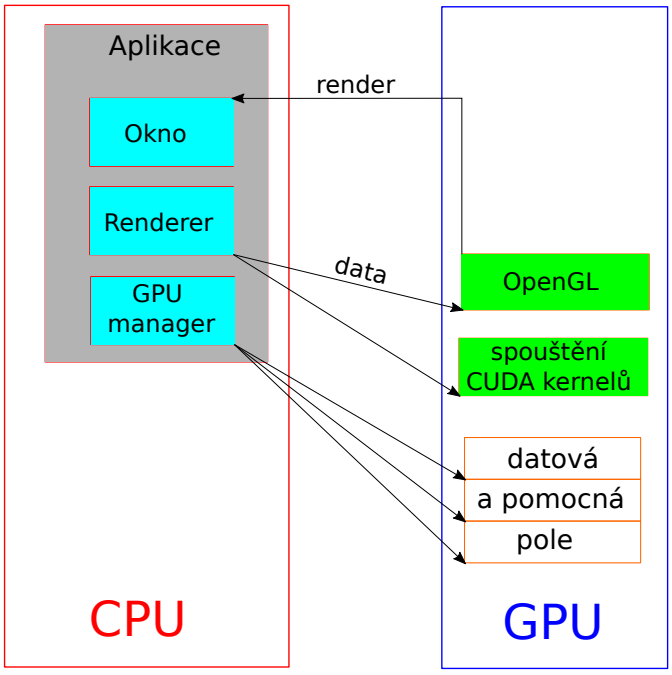
6 Implementace

Následující text obsahuje kompletní rozbor aplikace, ukázky kódu, podrobné vysvětlení všech použitých algoritmů.

6.1 Struktura aplikace

Obrázek (Obr. 15) zobrazuje „high-level“ pohled na aplikaci. Samotná třída aplikace udržuje ukazatele na 3 objekty:

- Manažer datových struktur a polí na GPU
- Renderer
- Okno



Obrázek 15: Schéma aplikace

Manažer datových struktur a polí slouží jako vstupní parametr pro renderer a ten díky němu může manipulovat s pamětí grafického akcelerátoru. Celá scéna je dekomponována na jednotlivé trojúhelníky, které jsou uloženy v globální paměti GPU jako souvislé pole.

Jako souvislé pole jednotlivých uzlů je uložena i urychlující struktura. V urychlující struktuře listy stromu tvoří unikátní Morton kódy. Indexy jednotlivých Morton kódů odpovídají indexům do pole duplicit a pole indexů prvních výskytů. Z dvou naposledy jmenovaných polí se odvozuje index do pomocného pole indexů, přes který se přistupuje ke správným trojúhelníkům na korektní pozici.

Renderer je nejdůležitější třídou v aplikaci. Zajišťuje prakticky veškerou logiku. Navíc spravuje i některé zdroje. Samotný CUDA framework, přes který probíhá celý proces raytracing, nedokáže renderovat přímo na obrazovku. Proto je nutné využít jedno z již existujících API. V aplikaci je použito OpenGL. Proto manažer polí spravuje paměť pro framework CUDA a renderer spravuje zdroje pro OpenGL.

Samotné okno je rovněž vstupním parametrem pro renderer a slouží jako obrazový výstup pro výsledek renderování. Do záhlaví okna ještě aplikace vykresluje aktuální frekvencí vykreslování ve snímcích za sekundu (FPS).

6.2 Inicializace

Jako první se aplikace pokusí nainicializovat framework GLFW, což je open source multiplatformní OpenGL knihovna pro správu a obsluhu oken. Poté se pokusí inicializovat OpenGL a vytvořit okno. Pokud inicializace OpenGL a okna proběhne bez chyby, pokračuje inicializací rendereru.

Inicializace rendereru má několik kroků. Nejprve se vytvoří textura, do které se bude mapovat výsledek raytracingu. Poté kompiluje vertex a fragment shader. Renderer bude vykreslovat pouze dva trojúhelníky se společnou přeponou přes celou obrazovku, na které bude ve fragment shaderu namapovaná výstupní textura představující framebuffer.

Výše zmíněné dva trojúhelníky se také musí za pomoci OpenGL API připravit na grafickou kartu, což je další krok inicializace rendereru. Renderer má v sobě uložené pole souřadnic vrcholů a pole indexů, které popisují trojúhelníky popisují. Během inicializace rendereru se obě pole překopírují na GPU do bufferu.

Další krok je inicializace generátoru náhodných čísel frameworku CUDA pro randomizaci směru paprsků vyslaných do jednoho pixelu.

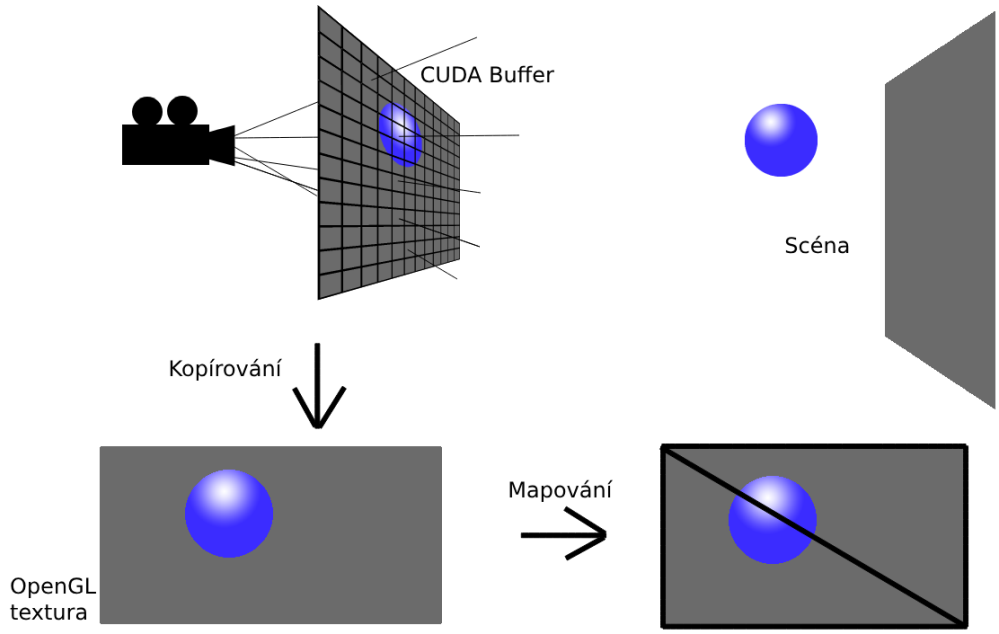
Paprsky se vysílají z virtuální kamery, kterou renderer rovněž inicializuje a ukládá ve své třídě. Inicializace kamery obnáší její umístění do světa a natočení určitým směrem. Poté se vytvoří rastrová průmětna, tj. virtuální výřez roviny, skrze který kamera kouká do světa. Je to projekční rovina, na kterou se bude promítat renderovaná scéna(Obr. 16).

Dále následuje alokace CUDA bufferu, který reprezentuje jednotlivé pixely výše zmíněné roviny. Právě tento buffer bude raytracing naplňovat a poté se namapuje na texturu.

Inicializace pokračuje tím, že se uživatele aplikace zeptá na soubor s 3D modelem. Pokud soubor existuje, aplikace ho otevře a naimportuje z něj data. Což jsou zpravidla trojúhelníky. U všech trojúhelníku poté spočítá ohraničující obálky (AABB) a středy těchto obálek. Následně spočítá AABB celé scény. Z ohraničující obálky celé scény pak normalizuje souřadnice středů obálek trojúhelníků do intervalu $\langle 0, 1 \rangle$. Všechna tato data jsou spočtena na CPU a poté přesunuta na grafickou kartu. Poté se spustí nekonečná renderovací smyčka zajišťující opakované překreslování scény.

```
void App::Run()
{
  float deltaTime=0.0;
  float lastFrame=0.0;
     // nekonecna smycka dokud uzivatel neukonci aplikaci
  while ( !glfwWindowShouldClose( m_window->GetWindow() ) ) {
        // vypocet snimkove frekvence
       float currentFrame = (float)glfwGetTime();
       deltaTime = currentFrame - lastFrame;
       lastFrame = currentFrame;
        // vykresleni snimkove frekvence horni listy okna
       m_window->ShowFPS( 1.0f / deltaTime );
       // renderovaci funkce
       m_renderer->Render( *m_GPUArrayManager );
       glfwSwapBuffers( m_window->GetWindow() );
       glfwPollEvents();
  }
}
```

Kód 6: Renderovací smyčka



Obrázek 16: Mapování obrazových dat

6.3 Proces renderování

Veškeré níže popsané kroky se provádí při vykreslení každého snímku, takže při renderování nového snímku se celá urychlovací struktura znova přebuduje.

Klíčové kroky vykreslovacího procesu jsou:

- Mortonovy rozklady a eliminace duplicit (Podkapitola 6.3.1)
- Konstrukce akcelerační struktury (Podkapitola 6.3.2)
- Spočtení hraničních rovin ve vnitřních uzlech (Podkapitola 6.3.3)
- Render (procházení struktury vystřelenými paprsky) (Podkapitola 6.3.4)
- Vykreslení na obrazovku (Podkapitola 6.3.5)

6.3.1 Mortonovy rozklady a eliminace duplicit

První krok algoritmu je spočtení Mortonových rozkladů jednotlivých objektů. K výpočtu Mortonova rozkladu jsou použity souřadnice středů AABB daných objektů. Dále je k výpočtu použit algoritmus dostupný na [20]. Tento algoritmus spočítá 30-bitový Mortonův kód pro jakýkoliv bod v jednotkovém kvádru (všechny souřadnice jsou v intervalu $\langle 0, 1 \rangle$). Proto byly při inicializaci středy ohraničujících obálek trojúhelníků normalizovány.

```
// Expanduje 10-bitovy integer na 30 bitu
// vlozenim 2 nul po kazdem bitu.
unsigned int expandBits(unsigned int v)
{
    v = (v * 0x00010001u) & 0xFF0000FFu;
    v = (v * 0x00000101u) & 0x0F00F00Fu;
    v = (v * 0x00000011u) & 0xC30C30C3u;
    v = (v * 0x00000005u) & 0x49249249u;
    return v;
}

// Spocita 30-bitovy Mortonuv kod pro jakykoli
// 3D bod nachazejici se v jednotkove kostce [0,1].
unsigned int morton3D(float x, float y, float z)
{
    x = min(max(x * 1024.0f, 0.0f), 1023.0f);
    y = min(max(y * 1024.0f, 0.0f), 1023.0f);
    z = min(max(z * 1024.0f, 0.0f), 1023.0f);
    unsigned int xx = expandBits((unsigned int)x);
    unsigned int yy = expandBits((unsigned int)y);
    unsigned int zz = expandBits((unsigned int)z);
    return xx * 4 + yy * 2 + zz;
}
```

Kód 7: Výpočet Mortonova rozkladu

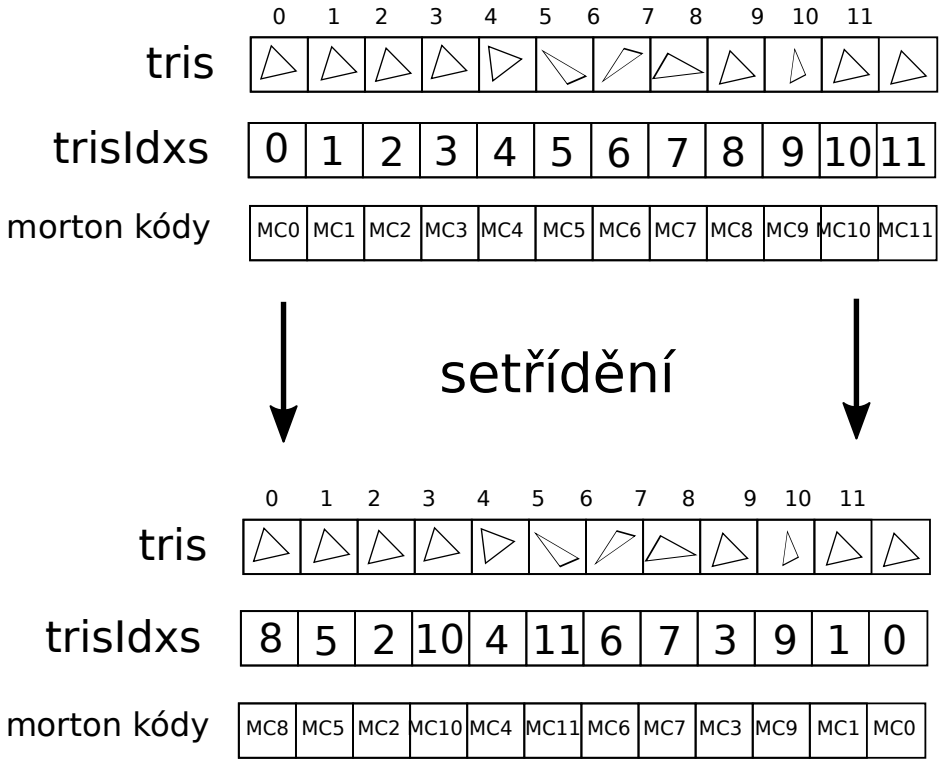
Žádná závislost mezi těmito výpočty neexistuje, tudíž výpočet všech Morton kódů lze provést paralelně s využitím knihovny `Thrust`. Výsledek je ukládán na grafickém akcelerátoru do pomocného pole v globální paměti.

Aby bylo možné najít blízké trojúhelníky, je třeba je seřadit, což je klíčové pro zpracování vnitřních uzlů. Místo řazení samotných trojúhelníků, jsou řazeny jejich Morton

kódy za účelem pozdějšího zpracování během budování struktury. Spolu s Morton kódy je řazeno i pole indexu, protože si algoritmus potřebuje uchovat informaci, který Morton kód náleží odpovídajícímu trojúhelníku. Tato informace je klíčová při procházení struktury. Zároveň je paměťově výhodnější třídit tato dvě pole než pole s trojúhelníky, neboť tímto způsobem je tříděn menší objem dat.

- Trojúhelníky: $3 * float * pocetTrojuhelniku = 3 * 32 * pocetTrojuhelniku = 96 * N$
- Morton kódy + indexy: $2 * int * pocetTrojuhelniku = 2 * 32 * pocetTrojuhelniku = 64 * N$

Ke třídění je opět použita knihovna `Thrust`, která poskytuje optimalizovaný paralelní třídicí algoritmus pro třídění dvou polí. Morton kódy mají funkci klíče. Na základě třídění klíčů se třídí i přidružené pole s indexy.



Obrázek 17: Třídění Mortonových rozkladů

Protože Mortonův rozklad vlastně dělí prostor na pravidelnou mřížku, může se stát, že více trojúhelníků náleží do jedné mřížky a tím vzniknou duplicitní kódy. Těch je potřeba

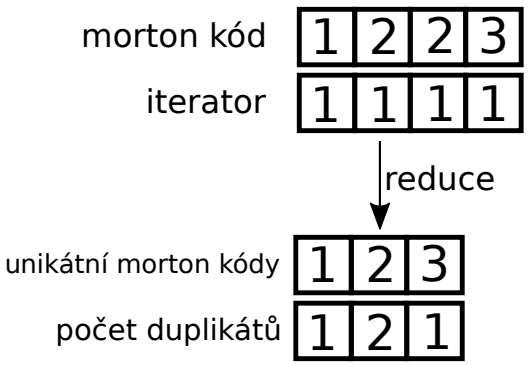
se zbavit, protože algoritmus zpracovávající vnitřní uzly stromu umí pracovat pouze s unikátními kódy. Rovněž si ale algoritmus musí zapamatovat, že danému Mortonově rozkladu náleží více trojúhelníků.

To se zařídí ve dvou krocích opět pomocí optimalizovaných paralelních funkcí v knihovně `Thrust`, které dokážou efektivně pracovat nad dvěma poli. První pole funguje jako klíč, druhé jako hodnota. Ve skutečnosti, aby nebylo nutné zbytečně alokovat paměť, dovoluje knihovna `Thrust` místo druhého pole použít speciální iterátor, který přesně vyhovuje požadavkům na třídění.

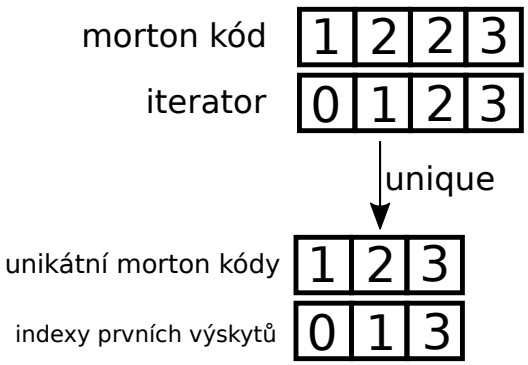
Nejprve je zjištěn počet duplicit. Protože jsou Mortonovy rozklady setříděné, všechny jednotlivé duplicity se nachází za sebou. Mortonovy kódy budou jako první vstupní pole a budou plnit funkci klíčů. Jako pole hodnot bude plnit funkci speciální iterátor, který na každou i-tou pozici dosadí jedničku. Poté se zavolá funkce, která „stlačí“ všechny duplicity až k prvnímu výskytu daného klíče a posčítá všechny stlačené jedničky v poli hodnot. Klíč i součet hodnot poté překopíruje do dvou výstupních polí klíčů a hodnot.

Nyní algoritmus má počet duplicit, ale protože se do výstupních polí vkládají po onom „stlačení“, nezná přesný index, kde v původním poli jednotlivé klíče začínají. Na spočtení této informace je použita obdobná funkce, jen s malými změnami. Speciální iterátor nyní nebude na i-tou pozici dosazovat jedničku, ale číslo i. Vytvoří vlastně sekvenci od 0 do $N-1$, kde N je počet trojúhelníků. Funkce potom překopíruje do výstupních polí jen ty prvky, kdy klíč na i-té pozici se nerovná klíči na $i-1$. pozici.

Tím algoritmus získal indexy všech prvních výskytů všech Mortonových rozkladů a počet jejich duplikátů. Budou důležité během procházení strukturou. Když algoritmus bude chtít projít všechny trojúhelníky se stejným rozkladem, bude vědět, že se musí podívat na i-tý index a posunout se o k indexů dál. V paměti jsou tři nová pomocná pole: pole prvních indexů, pole počtu duplikátů a pole unikátních Mortonových kódů. Lze budovat strukturu.



Obrázek 18: Funkce Reduce - spočtení duplikátů



Obrázek 19: Funkce Unique - zjištění indexu prvního výskytu

```
// paralelni spocteni mortonovych rozkladu
thrust::transform(thrust::device,
      gpuArrayManager.GetBBoxArrays().arrCenterNorm.begin(),
      gpuArrayManager.GetBBoxArrays().arrCenterNorm.begin() +
   gpuArrayManager.GetTrisSize(),
      gpuArrayManager.GetMortonCodesVectorRef().begin(),
      morton_functor());

// paralelni trideni mortonovych rozkladu
thrust::stable_sort_by_key(thrust::device,
      gpuArrayManager.GetMortonCodesVectorRef().begin(),
      gpuArrayManager.GetMortonCodesVectorRef().begin() +
   gpuArrayManager.GetTrisSize(),
      gpuArrayManager.GetTrisIndexes().begin()
      );

// paralelni nalezeni poctu duplikatu
auto newEnd = thrust::reduce_by_key(thrust::device,
      gpuArrayManager.GetMortonCodesVectorRef().data(),
      gpuArrayManager.GetMortonCodesVectorRef().data() +
   gpuArrayManager.GetTrisSize(),
      thrust::make_constant_iterator(1),
      gpuArrayManager.GetUniqueMortonCodesVectorRef().data(),
      gpuArrayManager.GetDuplicatesCnts().data()
      );

// paralelni nalezeni indexu prvnich vyskytu
thrust::unique_by_key_copy(thrust::device,
      gpuArrayManager.GetMortonCodesVectorRef().data(),
      gpuArrayManager.GetMortonCodesVectorRef().data() +
   gpuArrayManager.GetTrisSize(),
      thrust::make_counting_iterator(0),
      gpuArrayManager.GetUniqueMortonCodesVectorRef().data(),
      gpuArrayManager.GetFirstIdxs().data()
      );
```

Kód 8: Získání dodatečných informací Mortonových kódů

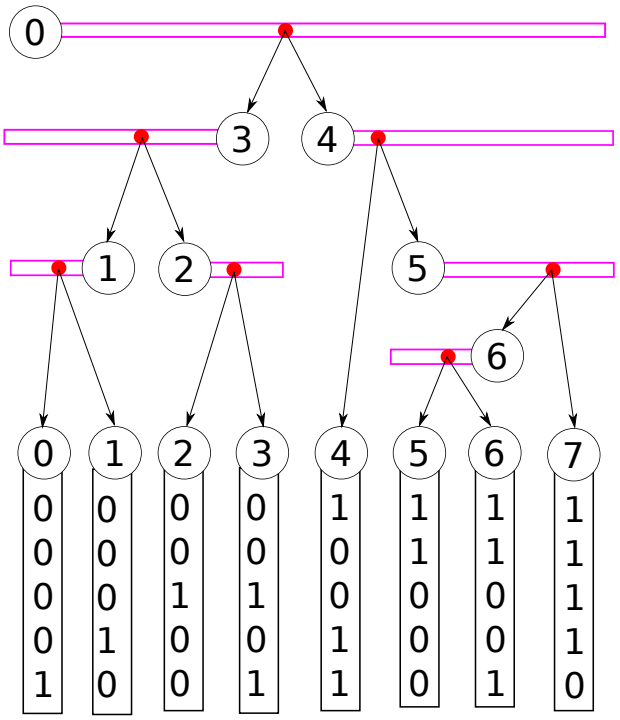
6.3.2 Konstrukce akcelerační struktury

Jak bylo zmíněno v úvodu, binární strom s N listy má $N - 1$ vnitřních uzlů, o kterých nejsou k dispozici žádné informace. Je nutné provést správné číslování vnitřních uzlů, které vytvoří jakési spojení s Morton kódy v listech.

Začíná se v kořenovém uzlu, který má vždy index 0. Kořen zahrnuje všech N listů. Na i-té a $i + 1$. pozici se kořen dělí na své potomky. Aplikací pravidla, že indexy těchto potomků budou právě jejich dělicí pozice. Toto pravidlo se aplikuje na všechny vnitřní uzly ve stromě.

Z toho plyne několik zajímavých důsledků:

- Vnitřní uzel s indexem i zahrnuje alespoň 2 listy, z nichž jeden je právě na i-té pozici.
- Rozsah Mortonových kódů se od i-té pozice může rozšiřovat oběma směry.
- Použité pravidlo číslování, nikdy nevytvoří nějaké číselné díry ani duplikáty. Výsledné indexy vnitřních uzlů jsou unikátní a pokud by byly setříděny, vytvořily by sekvenci od 0 do $N - 1$.
- Potomci společného rodičovského uzlu mají po sobě jdoucí indexy (tj. i a $i + 1$).
- Index všech vnitřních uzlů vždy koresponduje buďto s prvním, nebo posledním Mortonovým kódem v rozsahu.



Obrázek 20: Číslování vnitřních uzlů

Nyní je potřeba zjistit, které všechny Morton kódy vnitřní uzel zahrnuje a na kterém místě je rozdělit na levého a pravého potomka. Algoritmus, který se spustí paralelně na všech vnitřních uzlech by se dal shrnout do těchto 4 kroků:

1. Zjistit „směr intervalu“[2].
2. Rozšiřovat interval, dokud to jde.
3. Nalézt přesné místo dělení intervalu na levého a pravého potomka.
4. Určit počet prvků v jednotlivých potomcích.

6.3.2.1 ad 1)

Algoritmus začíná pouze s množinou Morton kódů na vstupu a indexem i, což je ID CUDA vlákna. Z předchozích důsledků je známo, že vnitřní uzel s indexem i zahrnuje Morton kód na i-té pozici. Další informace z předchozích důsledků je, že vnitřní uzel zahrnuje alespoň ještě jeden další list. Ale není známo, jestli to je list na pozici $i + 1$, nebo $i - 1$. To bylo myšleno tím „směrem intervalu“. Poslední informací, která plyne z předchozích důsledků je, že jeden ze sousedních listů patří do tohoto vnitřního uzlu, zatímco ten druhý zahrnuje uzel sourozenecký.[3]

Směr rozšiřování intervalu algoritmus zjistí dvojím porovnáním Mortonvých kódů. Respektive v jejich binární reprezentaci bude zjišťovat počet společných bitů od nejvýznamnějšího bitu. Počet společných bitů zjistí tak, že mezi dvěma Morton kódy provede operaci `XOR`, která vrátí nulu, pokud jsou bity stejné, nebo jedničku, pokud se liší. Výsledný kód se vloží do CUDA funkce `__clz`, která vrací počet nul od nejvýznamnějšího bitu. Což je přesně číslo, které je potřeba.

První operace se provede s kódy na pozici i a $i + 1$. Druhá operace se provede s kódy na pozici i a $i - 1$. Tím je získána informace, se kterým sousedním Morton kódem má kód na i-té pozici více společných bitů. Do tohoto vnitřního uzlu bude spadat ten, se kterým má více společných bitů. To dává smysl relativně intuitivně, že více společných bitů budou mít kódy, které spadají pod jeden vnitřní uzel, než kdyby každý spadal do jiného. Hodnotu méně společných bitů je ještě uchována pro další krok algoritmu.

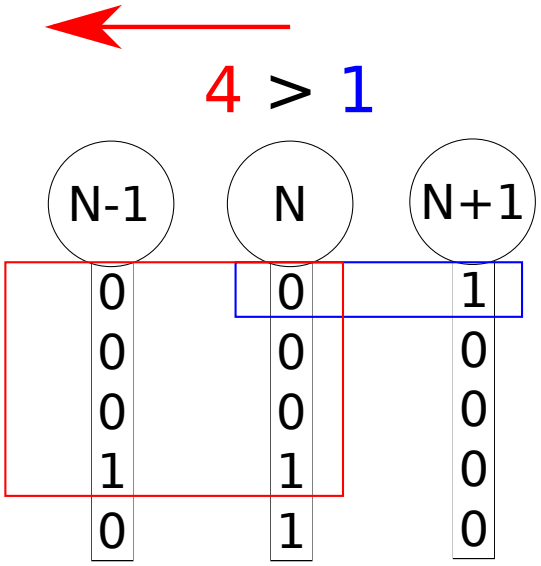
6.3.2.2 ad 2)

Nyní je potřeba spočítat, kolik vlastně Morton kódů vnitřní uzel zahrnuje. Jeden konec už je známý (Morton kód n i-té pozici). Teď algoritmus musí najít konec druhý. Délka intervalu bude násobena dvěma, dokud se nenalezne Morton kód, u kterého společný počet bitů bude menší než aktuální nalezené minimum. Což byl kód vzdálený jeden krok na opačnou stranu rozšiřování intervalu.

Při tomto způsobu zvětšování intervalu se může stát, že se algoritmus dostane mimo validní indexy, tzn. dostane se mimo pole Morton kódů. V takovém případě klade natvrdo společný počet bitů roven -1. Což automaticky splní ukončovací podmínku a ukončuje se rozšiřování intervalu. Binárním půlením nyní nalezne index druhého konce intervalu. Nyní se spočítá společný počet bitů Morton kódů na obou koncích intervalu.

[2] výraz „směr intervalu“ nedává příliš smysl. Bude vysvětleno v dalším odstavci

[3] Sourozenecký uzel je uzel na stejné hladině ve stromě, pod stejným rodičovským uzlem.



Obrázek 21: Výpočet směru rozšiřování množiny

6.3.2.3 ad 3)

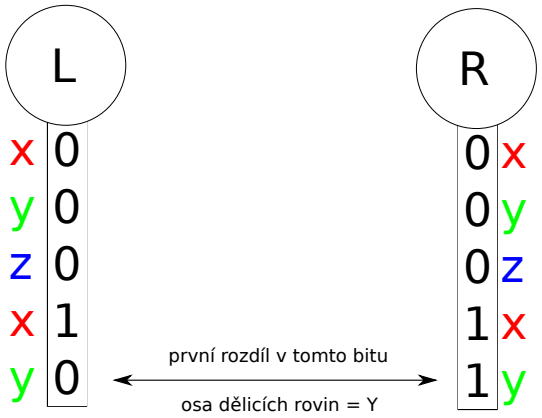
Podobně jako se hledal směr rozšiřování intervalu, bude se teď hledat index, na kterém se interval rozdělí na levého a pravého potomka. Bude se hledat přesně ten samý efekt. Jestliže algoritmus spočítal, že společný počet bitů na obou koncích intervalu je k, tak nyní bude hledat kódy, které mají společný počet bitů alespoň $k+1$. Jakmile narazí na kód, který má s i-tým kódem společný počet bitů k, našel dělicí index. Dělicí index se opět hledá binárním půlením.

6.3.2.4 ad 4)

Z původního vstupního indexu, z dělicího indexu a z indexu druhého konce intervalu se jednoduše dopočítá počet prvků v levém a pravém potomku. Pakliže jich je více než jeden, zapíší se do vnitřního uzlu pouze indexy. V opačném případě (potomek obsahuje pouze jeden prvek) se nastaví u příhodného potomka příznak, že je listem.

Protože byli právě spočítány indexy pro levého a pravého potomka aktuálního uzlu, může se jim nastavit odkaz na jejich rodiče, což je právě aktuální uzel. Tím aktuální vlákno vstupuje do uzlu, se kterým manipulují i jiná vlákna. Protože s odkazem na rodiče, u všech uzlů manipuluje právě jenom rodič, je manipulace bezpečná. Nedochází k žádnému datovému konfliktu ani přepisování paměti.

Jako vedlejší produkt dělení intervalu na levého a pravého potomka byla získána osa dělení. Mortonův rozklad vznikl jako prokládání bitů jednotlivých souřadnic bodu v prostoru. Počet společných bitů lze geometricky interpretovat jako společnou náležitost do stejné prostorové buňky v daném rozlišení. Čím hlouběji je algoritmus zanořen ve stromě, tím se rozlišení zvětšuje (tzn. více společných bitů a zmenšující se prostorové buňky). Zákonitě tedy nastane stav, kdy původní množina objektů v rodičovském uzlu náleží do jedné buňky, ale při zanoření do potomků se tato buňka rozdělí dle nějaké osy, čímž se rozdělí i množina objektů. Osa je uložena do vnitřního uzlu.[19]



Obrázek 22: Určení osy

6.3.3 Spočtení hraničních rovin ve vnitřních uzlech

Po dokončení konstrukce vnitřních uzlů se spustí nový kernel. Každé vlákno obsadí jeden list stromu a z pomocného pole předem spočtených ohraničujících obálek všech objektů si zjistí AABB příslušného objektu, který listu náleží. V každém uzlu struktury byl při její konstrukci uložen odkaz na rodičovský uzel. Po těchto odkazech bude každé vlákno stoupat strukturou nahoru až ke kořeni. V každém vnitřním uzlu se zjistí, zda se přišlo z levého potomka, nebo z pravého.

Pokud se přišlo z levého, porovná levou hraniční rovinu uloženou v uzlu s maximální hodnotou AABB v ose určené též vnitřním uzlem. Pokud je hodnota AABB větší, než hraniční rovina uložená v uzlu, hraniční rovina se touto hodnotou přepíše.

Pokud vlákno přišlo z potomka pravého, probíhá analogický proces, pouze se nepočítá maximum, ale minimum. Tímto způsobem je docíleno, že levá hraniční rovina bude na pozici geometrického primitiva nejvíce vpravo v levých potomcích a pravá rovina bude na pozici primitiva nejvíce vlevo v pravých potomcích. Výše zmíněné operace však musejí být prováděny atomicky, neboť zde k hraničním rovinám může v jednu chvíli přistupovat více vláken.

6.3.4 Render (procházení struktury vystřelenými paprsky)

Obrázek, který je renderován, má rozlišení $X * Y$`px`, kde X je šířka a Y výška obrázku. Vykreslovací kernel je spuštěn pro každy jednotlivý pixel. Pomocí dvourozměrné vnitřní proměnné `threadIdx` je identifikován vykreslovaný pixel. Do něj se následně sekvenčně ve smyčce vyšle N paprsků. Směr paprsků má náhodný rozptyl za účelem anti-aliasingu[4].

Následně se počítá průsečík s ohraničující obálkou celé scény. Pokud paprsek neprotne tento AABB, vrátí barvu pozadí. Pokud paprsek prochází skrz AABB scény, jsou spočteny dva parametry `tMin` a `tMax`. Parametr `tMin` udává vzdálenost kamery od bodu, kde paprsek vstupuje do AABB. Analogicky parametr `tMax` udává vzdálenost, kde paprsek ohraničující obálku opouští. Vzdálenost mezi těmito body je validní úsek paprsku, kde se bude hledat nejbližší průsečík s objektem.

[4]Anti-aliasing je proces vyhlazování hran. Popsaná metoda je supersampling - algoritmus vezme více vzorků a barvu zprůměruje.

Nyní si algoritmus nadefinuje pomocnou datovou strukturu **zásobník**. Na tento zásobník ukládá soubor dat:

- ukazatel na uzel struktury
- spodní hranici validního rozsahu paprsku
- horní hranici validního rozsahu paprsku

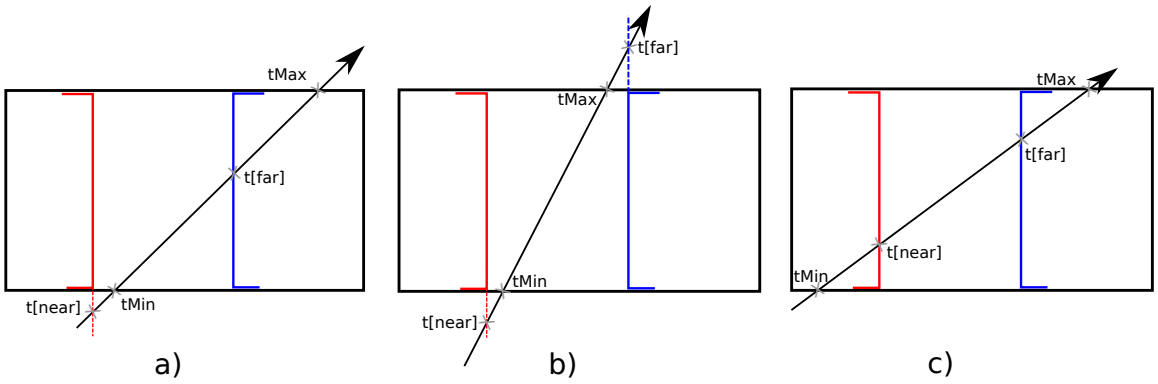
Dále je definován ukazatel na aktuálně zpracovávaný uzel, do něhož je vložen kořen stromu. Dokud je tento ukazatel nenulový (`pCurrNode != nullptr`), provádí se následující postup:

Z aktuálního uzlu se zjistí osa, na které budou probíhat všechny následující operace. Poté se spočítají průsečíky s oběma dělicími rovinami aktuálního uzlu. Dle znaménka u dané souřadnice směrového vektoru paprsku algoritmus určí, která z dělicích rovin je bližší a která vzdálenější. Vzdálenosti průsečíků s dělicími rovinami jsou uložené v dvouprvkovém poli `t[2]`. Na základě výše zmíněného znaménka jsou vytvořeny dvě proměnné (`near` a `far`), které zajistí invarianci ke směru paprsku. Ať bude směr paprsku jakýkoliv, indexy `t[near]` a `t[far]` budou vždy indexovat správné vzdálenosti. Teď je potřeba vyhodnotit několik podmínek.

```
while( pCurrNode != nullptr ){

    ...

    bool tMinLessThanNear = ( tMin < t[near] );
    bool tMaxLessThanFar = ( tMax < t[far] );

    bool noIntersection = ( !tMinLessThanNear && tMaxLessThanFar );
    bool nearIntersection = ( tMinLessThanNear && tMaxLessThanFar );
    bool farIntersection = ( !tMinLessThanNear && !tMaxLessThanFar );
    bool bothIntersection = ( tMinLessThanNear && !tMaxLessThanFar );

    ...
}
```

Kód 9: Vyhodnocení podmínek a určení typu průsečíku se strukturou

První podmínka vyhodnocuje, zda průsečík při vstupu do AABB scény (toto platí pouze pro kořen, protože `tMin`/`tMax` se bude postupně ořezávat na polohy dělicích rovin) je blíže ke kameře než průsečík s bližší dělicí rovinou. Analogicky druhá podmínka vyhodnocuje, zda průsečík při opuštění AABB scény je blíže než průsečík se vzdálenější dělicí rovinou. Pravdivostní kombinace těchto dvou podmínek určuje, jaký scénář nastal. U každého scénáře algoritmus kontroluje, zda některý z potomků není list.



Obrázek 23: Pořadí průsečíků na paprsku určuje, kterými potomky paprsek prochází. **a)** Paprsek prochází pouze vzdáleným potomkem. **b)** Paprsek neprotíná žádného potomka. **c)** Paprsek protíná oba potomky.

6.3.4.1 No intersect

Pokud není nalezen žádný průsečík, znamená to, že dělicí roviny se nepřekrývají a paprsek prošel mezi nimi. V tomto případě se vyzvedne uzel z pomocného zásobníku a procházení struktury pokračuje tímto uzlem. Pokud je zásobník prázdný, procházení končí.

```
if ( noIntersection ) {
        stackIdx--;
        currNode = stack[stackIdx].t_node;
        tMin = stack[stackIdx].t_tMin;
        tMax = stack[stackIdx].t_tMax;
 }
```

Kód 10: Vyhodnocení uzlu bez průsečíku

6.3.4.2 Near intersect – No leaf

Pokud se nalezne pouze průsečík s bližší dělicí rovinou a bližší potomek není list, pokračuje procházení struktury tímto potomkem. Zde se projeví síla dělicí roviny, protože zkrátí validní úsek paprsku. Hodnotu `tMax` přepíše hodnotou `t[near]`. Poněvadž žádný objekt, který spadá do následně procházeného potomka není dál, než dělicí rovina. Je tedy zbytečné v tomto úseku paprsku počítat jakékoliv průsečíky.

6.3.4.3 Near intersect – Leaf

Pokud je list pouze bližší potomek, iteruje se přes všechny geometrické objekty bližšího potomka a hledá se průsečík s nejbližším objektem. Nejkratší nalezená vzdálenost se zapamatuje. Pakliže už algoritmus nějaký list navštívil, vzdálenosti se porovnají a zapamatuje se ta menší z nich. Po skončení iterace se vyzvedne uzel z pomocného zásobníku a procházení struktury pokračuje tímto uzlem. Pokud je zásobník prázdný, procházení končí.

```
if ( nearIntersection ) {
     if ( currNode->isLeaf[near] )
     {
         FindNearestTriangle(firstIdxs, duplicatesCnts,
                             triangles, triangleIdxs, r,
                             currNode->children[near],
                             outRecord);
         stackIdx--;
         currNode = stack[stackIdx].t_node;
         tMin = stack[stackIdx].t_tMin;
         tMax = stack[stackIdx].t_tMax;
     }
     else
     {
         currNode = &( BIHTree[currNode->children[near]] );
         tMax = t[near];
     }
 }
```

Kód 11: Vyhodnocení uzlu s průsečíkem bližší dělicí roviny

6.3.4.4 Far intersect – No leaf

Postup je zde obdobný jako u bližší dělicí roviny. Jediný rozdíl je, že se nepokračuje bližším potomkem, ale vzdálenějším a nepřepisuje se `tMax` hodnotou `t[near]`, ale přepisuje se `tMin` hodnotou `t[far]`.

6.3.4.5 Far intersect – Leaf

Opět obdobný postup jako u průsečíku s bližší dělicí rovinou. Avšak neiteruje se přes objekty bližšího potomka, ale iteruje se přes potomky vzdálenějšího potomka.

6.3.4.6 Both intersect – No leaf

Pokud algoritmus narazí na uzel, u kterého protíná obě dělicí roviny, postupuje se následujícím způsobem. Na zásobník uloží ukazatel na vzdálenějšího potomka, horní hranici nezměněnou a spodní hranici zkrácenou stejným způsobem jako u **Far intersect – No leaf**.

6.3.4.7 Both intersect – Near leaf

U uzlu, kde paprsek protíná obě dělicí roviny a bližší z potomků je list, se nejprve iteruje

přes objekty listu a hledá se nejbližší průsečík. Potom se zkrátí validní úsek paprsku (`tMin` se přepíše hodnotou `t[far]`). Pokračuje se procházením struktury vzdálenějším potomkem.

6.3.4.8 Both intersect – Far leaf

Analogicky jako o odstavec výše. Pouze se projdou objekty vzdálenějšího potomka a úsek paprsku se zkrátí z opačné strany (`tMax` je nahrazen `t[near]`).

6.3.4.9 Both intersect – Both leaf

V případě, že paprsek protne obě dělicí roviny uzlu a oba potomci jsou listy se projdou objekty obou potomků. Nejprve bližšího, ale protože v něm nemusí být průsečík nalezen, prochází se i vzdálenější. Po zpracování potomků se pokračuje do dalšího vnitřního uzlu vyzvednutého ze zásobníku. Pokud je zásobník prázdný, končí procházení struktury.

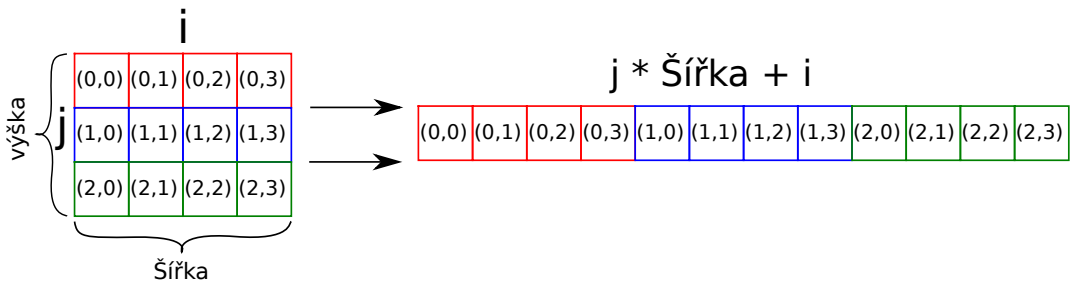
```
if ( bothIntersection ) {
  if ( currNode->isLeaf[near] && currNode->isLeaf[far] )
  {
      FindNearestTriangle(firstIdxs, duplicatesCnts,
                          triangles, triangleIdxs, r,
                          currNode->children[near], outRecord);
      FindNearestTriangle(firstIdxs, duplicatesCnts,
                          triangles, triangleIdxs, r,
                          currNode->children[far], outRecord);
      stackIdx--;
      currNode = stack[stackIdx].t_node;
      tMin = stack[stackIdx].t_tMin;
      tMax = stack[stackIdx].t_tMax;
  }
  else if( !currNode->isLeaf[near] && currNode->isLeaf[far] )
  {
      FindNearestTriangle(firstIdxs, duplicatesCnts,
                          triangles, triangleIdxs, r,
                          currNode->children[far], outRecord);
      currNode = &( BIHTree[currNode->children[near]] );
      tMax = t[near];
  }
  else if ( currNode->isLeaf[near] && !currNode->isLeaf[far] )
  {
      FindNearestTriangle(firstIdxs, duplicatesCnts,
                          triangles, triangleIdxs, r,
                          currNode->children[near], outRecord);
      currNode = &( BIHTree[currNode->children[far]] );
      tMin = t[far];
  }
  else
  {
   int farNodeIdx = currNode->children[far];
   stack[stackIdx].t_node = &( BIHTree[farNodeIdx] );
   stack[stackIdx].t_tMin = t[far];
   stack[stackIdx].t_tMax = tMax;
   stackIdx++;
   currNode = &( BIHTree[currNode->children[near]] );
   tMax = t[near];
  }
}
```

Kód 12: Vyhodnocení uzlu s průsečíkem v obou dělicích rovinách

Vzdálenost a index nejbližšího objektu jsou ukládány do pomocné struktury `HitRecord`. Inicializační hodnota indexu je nastavena na -1. Jakmile je tato hodnota větší než -1, znamená to, že algoritmus našel průsečík s nějakým trojúhelníkem. Tato hodnota je následně vrácena z funkce.

6.3.5 Vykreslení na obrazovku

Jednotlivé pixely jsou v globální paměti uloženy jako jednorozměrné pole o velikosti $X * Y$[5]. Za předpokladu, že paprsek prochází skrz pixel na i-tém řádku a j-tém sloupečku. Pokud je nalezen průsečík s objektem, zapíše se barva objektu do pole na pozici $j * X + i$. Pokud nalezen není, zapíše se barva pozadí.



Obrázek 24: Mapování matice do lineární paměti.

Nyní je kompletní obrázek uložen v globální paměti grafického akcelerátoru. Avšak přístup k němu má pouze CUDA framework. Je potřeba provést určitou transformaci dat, aby k nim mělo přístup OpenGL API.

Během inicializace aplikace byla pomocí OpenGL vytvořena v paměti GPU textura. Framework CUDA omezeně podporuje manipulaci s OpenGL objekty. Po vytvoření textury byla zavolána CUDA funkce `cudaGraphicsGLRegisterImage(...)`. Registrace textury touto funkcí umožní frameworku CUDA přistupovat přímo k textuře. Funkce se zavolala s parametrem `cudaGraphicsRegisterFlagsWriteDiscard`, což specifikuje, že CUDA z textury nebude číst, ale bude do ní zapisovat a žádný předchozí obsah textury nebude zachován. Dále má funkce jeden výstupní parametr, do kterého funkce zapíše „handle“, přes který se bude k textuře přistupovat.

[5] X je šířka obrázku a Y je výška.

```
__host__ void Renderer::CreateTextureDst() {
// vytvoreni textury
glGenTextures( 1, &m_quadTexture );
glBindTexture( GL_TEXTURE_2D, m_quadTexture );

// nastaveni zakladnich parametru
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST );

// nastavuje reprezantaci textury
// nastavuje dvourozmernou texturu pro aktualni texturovaci jednotku
// vychozi jednotka je GL_TEXTURE0
glTexImage2D( GL_TEXTURE_2D, 0, GL_RGBA8UI_EXT, SCREEN_WIDTH,
              SCREEN_HEIGHT, 0, GL_RGBA_INTEGER_EXT,
              GL_UNSIGNED_BYTE, NULL );


//// registrace textury frameworkem CUDA
checkCudaErrors( cudaGraphicsGLRegisterImage(
                 &m_cudaTexResultRes, m_quadTexture, GL_TEXTURE_2D,
                 cudaGraphicsMapFlagsWriteDiscard ) );
}
```

Kód 13: Inicializace OpenGL textury a její registrace

Handle vložíme jako parametr do další CUDA funkce – `cudaGraphicsMapResources(...)`. Funkce vnitřně namapuje texturu(OpenGL objekt) na datové CUDA pole. CUDA funkce `cudaGraphicsSubResourceGetMappedArray(...)` pole zpřístupní.

```
__host__ void Renderer::CreateTextureDst() {
cudaArray* texture_ptr;
checkCudaErrors( cudaGraphicsMapResources( 1, &m_cudaTexResultRes, 0 ) );
checkCudaErrors( cudaGraphicsSubResourceGetMappedArray(
                &texture_ptr, m_cudaTexResultRes, 0, 0 ) );

unsigned int num_texels = SCREEN_WIDTH * SCREEN_HEIGHT;
unsigned int num_values = num_texels * 4;
unsigned int size_tex_data = sizeof( GLubyte ) * num_values;
checkCudaErrors( cudaMemcpy2DToArray( texture_ptr, 0, 0,
                                      m_cudaDestResource,
                                      size_tex_data/SCREEN_HEIGHT,
                                      size_tex_data/SCREEN_HEIGHT,
                                      SCREEN_HEIGHT,
                                      cudaMemcpyDeviceToDevice ) );
```

Kód 14: Mapování OpenGL textury na CUDA pole

Algoritmus je ve fázi, kdy textura (OpenGL objekt) je zpřístupněna pro zápis CUDA frameworkem. Stačí zavolat funkci `cudaMemcpy2DToArray(...)`, která překopíruje pole pixelů v globální paměti (CUDA objekt) do jiného pole, které je mapováno přímo na texturu (OpenGL objekt). Nakonec zbývá pomocí OpenGL API zavolat kreslicí funkci `glDrawElements`, která spustí vykreslovací řetězec. Tímto řetězcem je textura namapovaná na dva trojúhelníky, které vyplňují plochu celé obrazovky. Ve vertex shaderu žádné výpočty neprobíhají. Pouze se souřadnice vrcholů trojúhelníků a texturovací souřadnice přepošlou ze vstupu na výstup. Ve fragment shaderu se provede samotné mapování textury na trojúhelníky. Pro schéma procesu viz Obr. 16.

```
{
 ...

 cudaArray* texture_ptr;
 checkCudaErrors( cudaGraphicsMapResources( 1, &m_cudaTexResultRes, 0 ) );
 checkCudaErrors(
    cudaGraphicsSubResourceGetMappedArray(
                &texture_ptr, m_cudaTexResultRes, 0, 0
    )
 );

 unsigned int num_texels = SCREEN_WIDTH * SCREEN_HEIGHT;
 unsigned int num_values = num_texels * 4;
 unsigned int size_tex_data = sizeof( GLubyte ) * num_values;
 checkCudaErrors(
    cudaMemcpy2DToArray(
        texture_ptr, 0, 0, m_cudaDestResource,
               size_tex_data/SCREEN_HEIGHT,
```

```
        size_tex_data/SCREEN_HEIGHT, SCREEN_HEIGHT,
                cudaMemcpyDeviceToDevice
    )
 );
 checkCudaErrors(
    cudaGraphicsUnmapResources(
        1,
        &m_cudaTexResultRes,
        0
    )
 );

 glClearColor( 0.2f, 0.3f, 0.3f, 1.0f );
 glClear( GL_COLOR_BUFFER_BIT );

 // bind Texture
 glBindTexture( GL_TEXTURE_2D, m_quadTexture );
 glEnable( GL_TEXTURE_2D );
 glDisable( GL_DEPTH_TEST );
 glPolygonMode( GL_FRONT_AND_BACK, GL_FILL );

 // render container
 glUseProgram( m_shaderProgram );
 glBindVertexArray( m_quadVAO );
 glDrawElements( GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0 );
}
```

Kód 15: Kopírování obrazových dat do textury a volaní vykreslovací funkce

7 Testování

Aplikace byla testována na počítači s procesorem Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz a grafickou kartou NVIDIA GeForce GTX 1660 Ti s taktovací frekvencí 1500 MHz a 1536 CUDA jádry.

Algoritmus byl testován na standardních testovacích scénách stanfordského buddhy, stanfordského draka a stanfordského králíka. Protože práce není založená na metodách stínování, pixely jsou pouze obarveny jednou barvou, když se nalezne nejbližší průsečík. Scény byly testovány ve stejném rozlišení jako v původní implementaci tj. 640x480 px a 4 paprsky per pixel.

model	#trojúhelníků	doba konstrukce CPU BIH (ms)	doba konstrukce GPU BIH (ms)
Stanford králík	70k	n.a.	0.21
Stanford buddha	1.07M	765	0.6
Stanford drak	860k	657	0.629

Tabulka 1: Porovnání doby konstrukce původní a této implementace

model	fps CPU BIH	fps GPU BIH	render 1 snímku CPU BIH (ms)	render 1 snímku GPU BIH (ms)
Stanford králík	10.2	1.11	176	726.1
Stanford buddha	7.41	2.23	1837	444.2
Stanford drak	5.98	1.71	1557	586.2

Tabulka 2: Porovnání doby vykreslení jednoho snímku

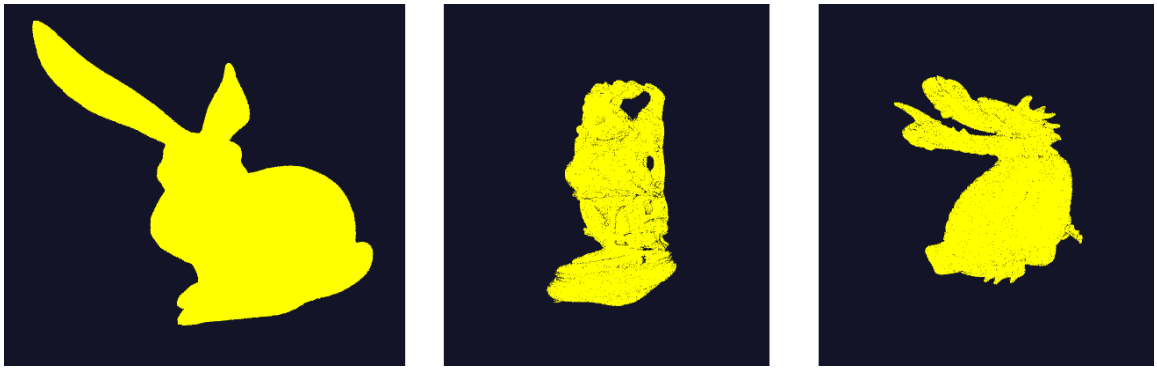
model	využitá paměť CPU BIH (MB)	využitá paměť GPU BIH (MB)
Stanford králík	3.36	8.63
Stanford buddha	47.39	134.84
Stanford drak	41.44	108.04

Tabulka 3: Porovnání paměťové náročnosti

model	BIH	KD-Tree viz[21]	BVH viz[22]
Stanford králík	0.21 ms	4.3 ms	n.a.
Stanford buddha	0.6 ms	9.3 ms	11.4 ms
Stanford drak	0.629 ms	8.7 ms	11.1 ms

Tabulka 4: Porovnání doby konstrukce různých urychlujících struktur

Obrázek 25: Renderované scény



Obrázek 26: Výstup z rendereru této práce. Pixely jsou vybarveny jednolitou barvou, protože stínování nebylo hlavním předmětem práce.

Výsledky dopadly smíšeně. Přestože samotná zrychlující struktura je vygenerovaná relativně rychle (v řádu desítek až stovek mikrosekund), vykreslení samotného obrázku trvá (v závislosti na počtu trojúhelníků ve scéně) velice dlouho (v řádu stovek milisekund). Na vině je pomalé samotné procházení urychlující struktury. Z analýzy profilovacího softwaru vyplývá, že je to způsobené datovou a výpočetní divergencí.

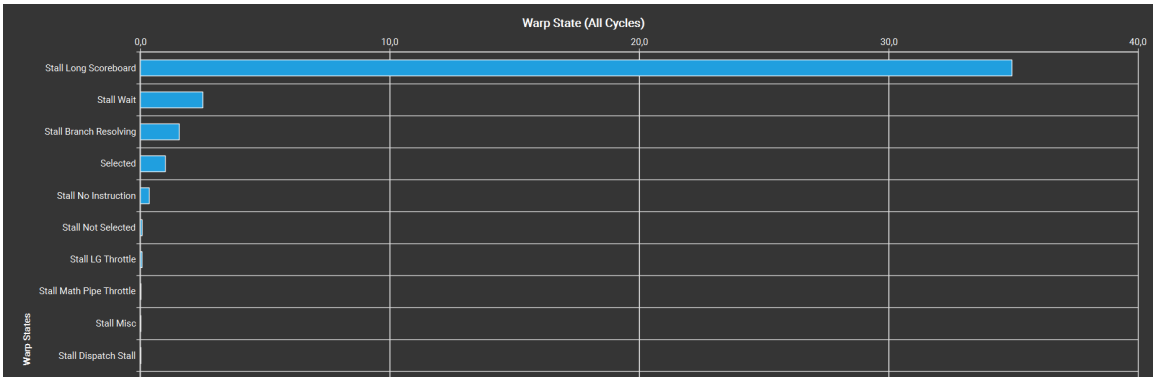
7.1 Výpočetní divergence

V jednom CUDA bloku běží N vláken. Všechna vlákna vykonávají stejnou instrukci. Problém nastane u větveného kódu. Když všechna vlákna projdou stejnou větví, je vše v pořádku. Pokud však alespoň jedno z vláken projde druhou větví, nelze potom kód vykonávat paralelně, protože dané vlákno vykonává jiné instrukce než zbytek bloku. V takovém případě se jednotlivé instrukce zařadí za sebe a všechna vlákna projdou oběma větvemi sekvenčně.

7.2 Datová divergence

Pokud každé vlákno prochází úplně jinou částí datové struktury, může každé vlákno projít i jiným počtem iterací a přistupuje k úplně jiným uzlům ve stromě. Což znamená, že

přistupuje do zcela jiné paměti. Kvůli tomu klesá efektivita rychlé cache paměti. Místo aby vlákna opakovaně využila data nahraná do cache paměti, musí přistupovat do globální paměti. Přístup do globální paměti je násobně pomalejší než do cache.



Obrázek 27: Výstup z profilovacího softwaru. Zobrazuje v jakých stavech se vlákna nacházela po dobu běhu programu. Nejvíc času vlákna trávila ve stavu „Stall Long Scoreboard“, což v praxi znamená, že čekala na nahrání dat z globální paměti. Na dalších řádcích se vlákna nacházela ve stavu: 2. Stall Wait; 3. Stall Branch Resolving; 4. Selected; 5. Stall No Instruction.

Co se týče paměťové náročnosti, kvůli několika pomocným polím je GPU implementace mnohem náročnější na prostor než původní CPU implementace.[18] Kromě pole, kde jsou uloženy jednotlivé uzly stromu jsou v paměti uloženy tyto:

- pole s trojúhelníky
- pole se všemi Morton kódy
- pole s unikátními Morton kódy
- pole s indexy, které mapují setříděné Morton kódy na příslušné trojúhelníky
- pole s počtem duplicitních Morton kódů
- pole s indexy prvních výskytů unikátních Morton kódu v poli s duplicitami.

Vyčísleno například pro model králíka: Králík ma cca 70k trojúhelníků. Každý trojúhelník má svůj Mortonův rozklad. Pole s indexy obsahuje stejný počet prvků jako pole trojúhelníků. Unikátních Mortonových kódů nechť je pro jednoduchost výpočtu 60k. Pole duplicit a pole prvních výskytů má stejný počet prvků jako pole unikátních Mortonových kódů. Všechny elementy v polích mají 4bajtový datový typ (`int` nebo `float`). Pole s trojúhelníky je navíc tvořeno třemi floaty. Z toho plyne:

$$(3+2)*4*70k+3*4*60k \doteq 2.1\,\text{MB} \tag{21}$$

Za předpokladu, že urychlovací struktura má přibližně podobnou paměťovou zátěž jako CPU verze algoritmu a s odečtením pole trojúhelníků zabírá GPU verze algoritmu zhruba o 1.8 MB více, než původní CPU implementace.

8 Závěr

8.1 Shrnutí

Cílem práce bylo prozkoumat možnosti implementace urychlující struktury Bounding Interval Hierarchy na grafické kartě. Změřit dobu konstrukce, dobu procházení strukturou a počet vykreslených snímků za sekundu pro potenciální použití v realtime aplikacích. Naměřené časy porovnat s původní implementací BIH i s ostatními urychlujícími strukturami.

Práce prokázala, že pokud má být konstrukce urychlující struktury paralelizována, nelze strukturu konstruovat metodou shora dolů. Paralelizace jednotlivých stromových hladin u této metody vede k neefektivnímu pokrytí výpočetních zdrojů na grafické kartě. Algoritmus se musí změnit fundamentálním způsobem.

Díky vlastnostem Mortonova rozkladu a specifického očíslování jednotlivých vnitřních uzlů struktury bylo možné rozbít datovou závislost mezi rodiči a potomky všech vnitřních uzlů. Na základě přerušení této závislosti lze všechny vnitřní uzly zpracovávat paralelně, což vede k masivnímu urychlení budování struktury. Jak lze vidět z tabulky (Tab. 1), doba konstrukce se pohybuje v řádu desetin milisekund. Jistý podíl na rychlosti konstrukce může mít i fakt, že všechny datové struktury jsou alokovány již předem a během konstrukce neprobíhá žádná dynamická alokace paměti.

Jako problém se ukázalo procházení struktury. Pokud vlákna přistupují do globální paměti, nahrají si nejbližších N bytů do značně rychlejší cache paměti. Bohužel při procházení struktury se vlákna v jednom bloku mohou každé dostat do jiné části struktury, tudíž dochází k častému přepisování cache a tím k častějšímu přistupování do pomalé globální paměti.

Bez další vhodné úpravy této části algoritmu je vykreslování snímků velice pomalé (viz Tab. 2) a v aplikacích reálného času nepoužitelné. Nicméně samotná konstrukce podává velmi solidní výsledky a potenciál pro použití v realtime aplikacích má.

8.2 Doporučení

Předmětem dalšího zkoumání se nabízí způsob uložení struktury v paměti grafického akcelerátoru. Současný způsob vyplynul z metody číslování vnitřních uzlů během budování struktury. Případně se lze zabývat efektivním procházením struktury ve stávající podobě.

Další skrytý potenciál se nachází v použité technologii. Framework CUDA momentálně neposkytuje žádný přístup k RTX technologii. Pokud by někdy v budoucnu společnost NVIDIA poskytla CUDA API k využití RT jader, naskýtá se příležitost využít tato jádra k efektivním výpočtům průsečíků.

Struktura se s každým snímkem konstruuje znova. Zde je prostor pro zkoumání budování BIH tzv. „on demand“– struktura se přebuduje pouze, když je to potřeba.

Literatura

[1] jbikker. RayTriangle Intersection. May 2020. Available from: https://jacco.ompf2.com/wp-content/uploads/2020/05/image-7-1024x489.png

[2] TheBeard. CUDA Memory Hierarchy. May 2020. Available from: http://thebeardsage.com/wp-content/uploads/2020/05/cudamemoryhierarchy-768x535.png

[3] Wetzstein, G. The Graphics Pipeline and OpenGL I: Transformations. 2022. Available from: https://stanford.edu/class/ee267/lectures/lecture2.pdf

[4] KhronosGroup. Rendering Pipeline Overview - OpenGL Wiki. Nov. 2022. Available from: https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview

[5] Segura, R.; Feito, F. Algorithms to Test Ray-Triangle Intersection. Comparative Study. In *Journal of WSCG*, volume 9, Jan. 2001, pp. 76–81.

[6] Cramerovo pravidlo. May 2023, page Version ID: 22825343. Available from: https://cs.wikipedia.org/w/index.php?title=Cramerovo_pravidlo&oldid=22825343

[7] Möller, T.; Trumbore, B. Fast, minimum storage ray/triangle intersection. In *ACM SIGGRAPH 2005 Courses on - SIGGRAPH '05*, Los Angeles, California: ACM Press, 2005, p. 7, doi:10.1145/1198555.1198746. Available from: http://portal.acm.org/citation.cfm?doid=1198555.1198746

[8] About CUDA. Mar. 2014. Available from: https://developer.nvidia.com/about-cuda

[9] CUDA C++ Programming Guide. July 2023. Available from: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[10] Thrust - Parallel Algorithms Library. Available from: https://thrust.github.io/

[11] Lagae, A.; Dutré, P. Compact, fast and robust grids for ray tracing. In *ACM SIGGRAPH 2008 talks*, Los Angeles California: ACM, Aug. 2008, ISBN 978-1-60558-343-3, pp. 1–1, doi:10.1145/1401032.1401059. Available from: https://dl.acm.org/doi/10.1145/1401032.1401059

[12] Hapala, M.; Havran, V. Review: Kd-tree Traversal Algorithms for Ray Tracing. *Computer Graphics Forum*, volume 30, no. 1, Mar. 2011: pp. 199–213, ISSN 01677055, doi:10.1111/j.1467-8659.2010.01844.x. Available from: https://onlinelibrary.wiley.com/doi/10.1111/j.1467-8659.2010.01844.x

[13] Dinh, F. *Hierarchie obalových těles*. Diplomová práce, Masarykova univerzita, Fakulta informatiky, Brno, 2016. Available from: https://is.muni.cz/th/jto7n/

[14] Kuckir, I. *Datové struktury pro zobrazování nepolygonální geometrie*. bakalářská práce, Univerzita Karlova, Matematicko-fyzikální fakulta, Katedra softwaru a výuky informatiky, 2013. Available from: https://dspace.cuni.cz/handle/20.500.11956/53507

[15] Walter, B.; Bala, K.; Kulkarni, M.; et al. Fast agglomerative clustering for rendering. In *2008 IEEE Symposium on Interactive Ray Tracing*, Los Angeles, CA, USA: IEEE, Aug. 2008, ISBN 978-1-4244-2741-3, pp. 81–86, doi:10.1109/RT.2008.4634626. Available from: http://ieeexplore.ieee.org/document/4634626/

[16] Meister, D.; Ogaki, S.; Benthin, C.; et al. A Survey on Bounding Volume Hierarchies for Ray Tracing. *Computer Graphics Forum*, volume 40, no. 2, May 2021: pp. 683–712, ISSN 0167-7055, 1467-8659, doi:10.1111/cgf.142662. Available from: https://onlinelibrary.wiley.com/doi/10.1111/cgf.142662

[17] Bounding volume hierarchy. Apr. 2023, page Version ID: 1147718410. Available from: https://en.wikipedia.org/w/index.php?title=Bounding_volume_hierarchy&oldid=1147718410

[18] Wächter, C.; Keller, A. Instant Ray Tracing: The Bounding Interval Hierarchy. *Symposium on Rendering*, 2006: p. 11 pages, ISSN 1727-3463, doi:10.2312/EGWR/EGSR06/139-149, artwork Size: 11 pages ISBN: 9783905673357 Publisher: The Eurographics Association. Available from: http://diglib.eg.org/handle/10.2312/EGWR.EGSR06.139-149

[19] Karras, T. Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees. *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics*, 2012: p. 5 pages, ISSN 2079-8679, doi:10.2312/EGGH/HPG12/033-037, artwork Size: 5 pages ISBN: 9783905674415 Publisher: The Eurographics Association. Available from: http://diglib.eg.org/handle/10.2312/EGGH.HPG12.033-037

[20] Karras, T. Thinking Parallel, Part III: Tree Construction on the GPU. Dec. 2012. Available from: https://developer.nvidia.com/blog/thinking-parallel-part-iii-tree-construction-gpu/

[21] Li, Z.; Wang, T.; Deng, Y. Fully parallel kd-tree construction for real-time ray tracing. In *Proceedings of the 18th meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, San Francisco California: ACM, Mar. 2014, ISBN 978-1-4503-2717-6, pp. 159–159, doi:10.1145/2556700.2566638. Available from: https://dl.acm.org/doi/10.1145/2556700.2566638

[22] Zlatuška, M.; Havran, V. Ray Tracing on a GPU with CUDA—comparative study of three algorithms. Feb. 2010. Available from: http://hdl.handle.net/11025/11043

Seznam zkratek

AABB axis aligned bounding box

API application programming interface

BIH bounding interval hierarchy

BVH bounding volume hierarchy

DNN deep neuron network

FPS frames per second

Přílohy

A Obsah přiloženého flashdisku

Na přiloženém mediu se nachází:

- Výsledná zkompilovaná aplikace
- Zdrojové kódy
- Visual Studio 2019 solution
- 3D modely Stanfordského buddhy, králíka a draka

Zdrojový kód je třeba kompilovat přes Microsoft Visual Studio verze 2019. Dále je pro kompilaci potřeba mít nainstalovaný CUDA Toolkit verze 11.3 a výš.

Zdrojové kódy jsou dostupné i na Githubu:
https://github.com/rehakvoj1/BIH-GPU-Raytracer

Na médiu se nachází 3 scény pro rendering. Validní vstupy pro aplikaci:

- buddha
- dragon
- bunny

Univerzita Hradec Králové
Fakulta informatiky a managementu

Zadání diplomové práce

Autor: **Bc. Vojtěch Řehák**

Studium: I2000317

Studijní program: N1802 Aplikovaná informatika

Studijní obor: Aplikovaná informatika

Název diplomové práce: **Zobrazení 3D scény metodou raytracing se zaměřením na urychlující datové struktury**

Název diplomové práce AJ: Rendering of 3D scene using raytracing method with focus on accelerating data structures

Cíl, metody, literatura, předpoklady:

Cíl práce:

Cílem práce je prozkoumat metody zobrazení prostorové scény metodou sledování paprsku (raytracing). Zaměřit se na optimalizaci a akceleraci výpočtu s využitím programovatelných grafických karet (GPU). Zvýšená pozornost bude věnována struktuře Bounding Interval Hierarchy.

Postup prací:

1. Prozkoumat principy vizualizace metodou raytrancing.
2. Vytvořit přehled existující metod a implementací. Zaměřit se na Bounding Interval Hierarchy a porovnat s běžněji používanými strukturami (BVH,KD-strom).
3. Pro vybranou metodu navrhnout implementaci se zřetelem na optimalizaci a akceleraci výpočtu s využitím GPU.
4. Navržené řešení implementovat a otestovat pro vhodné prostorové scény.
5. Zhodnotit dosažené výsledky.

WÄCHTER, Carsten a Alexander KELLER, 2006. Instant Ray Tracing: The Bounding Interval Hierarchy. *Symposium on Rendering* [online]. 11 pages. ISSN 1727-3463. Dostupné z: doi:10.2312/EGWR/EGSR06/139-149

KAJIYA, James T., 1986. The rendering equation. In: *the 13th annual conference*: *Proceedings of the 13th annual conference on Computer graphics and interactive techniques - SIGGRAPH '86* [online]. Not Known: ACM Press, s. 143–150 [vid. 2021-10-11]. ISBN 978-0-89791-196-2. Dostupné z: doi:10.1145/15922.15902

KARRAS, Tero a Timo AILA, 2013. Fast parallel construction of high-quality bounding volume hierarchies. In: *the 5th High-Performance Graphics Conference*: *Proceedings of the 5th High-Performance Graphics Conference on - HPG '13* [online]. Anaheim, California: ACM Press, s. 89 [vid. 2021-10-11]. ISBN 978-1-4503-2135-8. Dostupné z: doi:10.1145/2492045.2492055

Zadávající pracoviště: Katedra informatiky a kvantitativních metod,
Fakulta informatiky a managementu

Vedoucí práce: Ing. Bruno Ježek, Ph.D.

Oponent: prof. RNDr. PhDr. Antonín Slabý, CSc.

Datum zadání závěrečné práce: 12.1.2021