MASARYKOVA UNIVERZITA
FAKULTA INFORMATIKY

Paralelní budování akceleračních struktur pro ray tracing

Diplomová práce

Ondřej Sochora

Brno, 2013

Prohlášení

Prohlašuji, že tato diplomová práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

Chci poděkovat svému vedoucímu RNDr. Marku Vinklerovi za jeho čas, ochotu a hodnotné rady, díky kterým mohla tato práce vzniknout.

Shrnutí

Tato diplomová práce se zabývá paralelním budováním stromu obalových těles (bounding volume hierarchy) za použití více vláken během stavby. Práce mimo jiné vysvětluje postup, jak lze zajistit, aby v úloze stavby BVH stromu byly zapojeny všechny dostupné procesory a jádra a dále navrhuje a realizuje tři možné implementace této úlohy. V závěru pojednává o přínosu a zlepšení navržených metod stavby oproti jednovláknové implementaci a prezentuje a porovnává dosažené výsledky všech implementací.

Klíčová slova: Metoda sledování paprsku, hierarchie obalových těles, multivláknové programování, paralelní budování akceleračních struktur

Obsah

Kapitola 1

Úvod

Už od počátku vzniku počítačové grafiky bylo jejím cílem co nejvěrněji zobrazovat reálný svět. S rozvojem grafického hardwaru a technologií se tohoto cíle dosahuje pořád snáze a rychleji. Nicméně i v současné době, byť se sebevýkonnějším hardwarem, je zobrazování komplexních scén pořád stále dosti náročné a je nutné hardwaru pomoci důmyslnými algoritmy a technikami. Nejlepším řešením jsou takové algoritmy, které lze do hardwaru přímo naimplementovat, avšak ne vždy je toto možné.

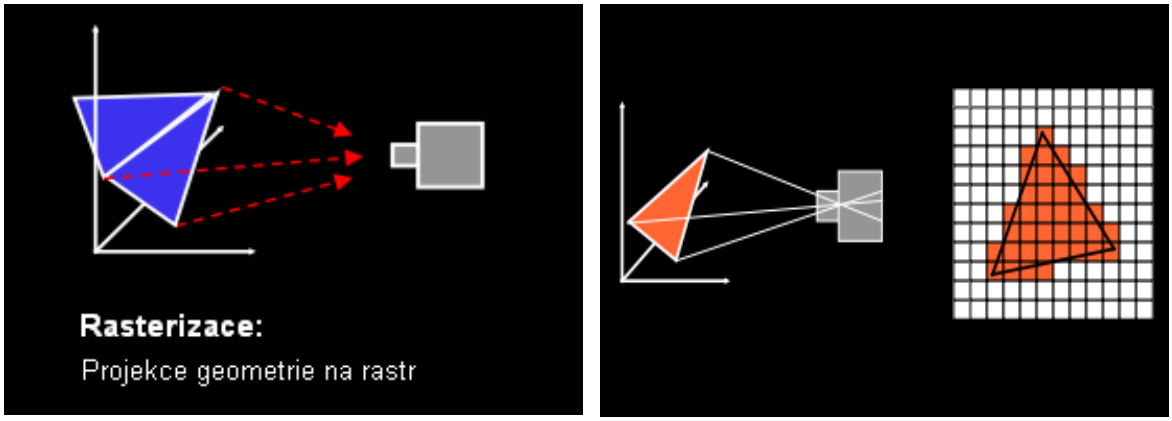
Účelem diplomové práce je představit tři implementace, které umožní zrychlit a zefektivnit algoritmus výstavby datové struktury, která je dále použita během vykreslování metodou ray tracing (algoritmus vykreslování scény, který používá paprsky, jež vrhá směrem do scény a získává tak informace o celé scéně a o jednotlivých objektech, které se v ní nachází). Práce se konkrétně zaměřuje na vícevláknovou stavbu stromu hierarchie obalových těles (BVH - bounding volume hierarchy). Toho je dosaženo rozdělením procesu stavby do více vláken, která paralelně zpracovávají data scény na více procesorech, respektive jádrech. Text také popisuje problémy, které vícevláknová implementace obnáší, jak se jim lze vyhnout a úspěšně je vyřešit. V závěru textu budou porovnány přístupy tvorby BVH stromu pomocí jednoho vlákna, vícevláknových metod s rovnoměrným a nerovnoměrným přidělováním jader a metody s využitím fronty uzlů.

Tato diplomová práce je rozšířením projektu autorů T. Aily a S. Laineho „Understanding the Efficiency of Ray Traversal on GPUs“ [LAK09], který implementuje vysoce rychlý raytracer (program používající algoritmus ray tracing), který využívá současný výkon grafických čipů a technologii CUDA od společnosti Nvidia [CUDA]. Práce se soustředí výhradně na vícevláknovou stavbu BVH stromu a rozšiřuje již autory navrženou jednovláknovou implementaci stavby.

1.1 Rasterizace

V počítačové grafice dominují dva základní algoritmy, které vykreslují trojrozměrnou scénu na dvourozměrné zobrazovací zařízení. První a v dnešní době stále převažující (zejména v real-time aplikacích) je rasterizační algoritmus, který je implementován přímo v grafických čipech. Zjednodušeně rasterizace funguje tak,

že se sekvenčně zpracovává trojúhelník za trojúhelníkem, každý z nich se promítne na rastrovou mřížku a spočte se, kolik pixelů daný trojúhelník na mřížce pokrývá. Pixely (resp. vzniklé fragmenty) se mohou ještě dodatečně filtrovat a upravovat v pomocných bufferech a mohou se na nich provádět další operace, například pomocí shaderů. Nevýhoda tohoto přístupu je, že s rostoucím počtem trojúhelníků lineárně narůstá i čas renderingu celé scény.



*Obrázek 1: Zobrazení principu rasterizace. Jednotlivé polygony scény se promítají na rastr a pro každý polygon se počítá, kolik elementů rastru (např. pixelů) polygon pokryje a jakou výslednou barvu každý obrazový element bude mít. (Reprodukováno z [SSM*05])*

1.2 Přednosti a zápory rasterizačního algoritmu

Ačkoliv oba algoritmy (rasterizace i raytracing) existují shodně dlouho, rasterizaci se věnovalo více úsilí a zájmu. Tento algoritmus je natolik efektivní a „jednoduchý“, že se jej podařilo implementovat do grafického hardwaru a v podstatě jej obsahují všechny současné i starší grafické čipy.

Další z výhod tohoto algoritmu tkví v pipelinovém zpracování dat. Pipelinové zpracování není přímo součástí rasterizačního algoritmu jako takového, ale pomáhá ušetřit čas při předzpracování dat pomocí operací, které jsou pro rasterizaci nezbytné. Než se nějaké grafické primitivum vykreslí na výstupním zařízení, prochází mnoha fázemi od zpracování jeho reprezentace v podobě vrcholů, přes testování v z-bufferu až po finální vykreslení. Protože je zbytečné, aby další primitiva čekající na zpracování vyčkávala na skončení všech operací na primitivu předchozím, jsou tato čekající primitiva zpracovávána ihned, jakmile je skončena následující operace na předchozím primitivu. Tento postup tedy vytvoří jistou posloupnost zpracovávaných primitiv, každé se nacházející v jiné fázi zpracování.

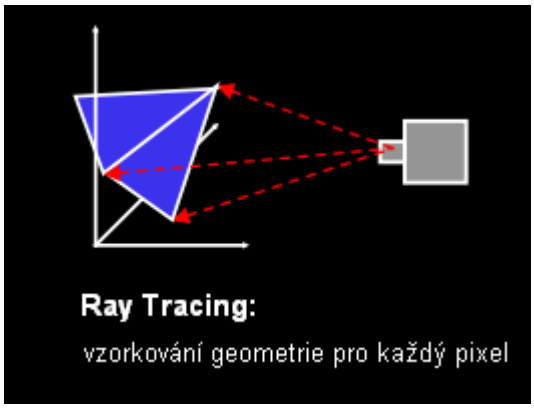
Bohužel tento přístup má i svá omezení a to taková, že v jeden časový okamžik lze zpracovat jeden jediný trojúhelník geometrie. To v momentu, kdy je potřeba znalost geometrie celé scény, třeba za účelem výpočtu globálního osvětlení nebo výpočtu vržení stínu na ostatní objekty ve scéně, je značně limitující. Zde je potřeba použít různých metod, které tento nedostatek kompenzují (je zapotřebí více průchodů vykreslovanými daty). Avšak tyto metody daný problém výpočtu pouze aproximují a výsledek je sice použitelný v mnoha případech, ale bývá dosti nepřesný (např. výskyt šumu nebo artefaktů na hranách stínů objektů).

I přes významná omezení se rasterizační algoritmus těší velké oblibě a tempo jeho vývoje a zdokonalování neustále roste. Pomocí shaderů lze grafické čipy poměrně dobře programovat a lze různě modifikovat klasický proces pipeliningu, což přináší nové možnosti a rozšiřuje dovednosti rasterizace.

1.3 Ray tracing

Filozofie ray tracingu je zcela odlišná. Algoritmus pro každý pixel rastru vrhá pomyslné paprsky směrem do scény a pro každý paprsek se sčítá výsledná barva v závislosti na barvě objektů ve scéně, které konkrétní paprsek protne. S návrhem použít paprsky pro výpočet obrazu scény přišel v roce 1968 Arthur Appel [App68]. Existuje plno vylepšení tohoto procesu pro dosažení vysoké kvality a věrnosti výstupu, například samostatné paprsky pro odlesky povrchů objektů či výpočty stínů, sekundární paprsky, paprsky pro generování fotonové mapy (kaustika) a jiné.

Samozřejmě další množství paprsků se promítne do celkové rychlosti algoritmu. Protože naivní implementace tohoto algoritmu je poměrně neefektivní (asymptotická časová složitost je kvadratická), používají se sofistikované datové struktury, které vhodně dělí prostor scény tak, že je možné velmi rychle určit těleso ve scéně, které je v daný okamžik zasaženo paprskem. Stejně tak i u rasterizace existují postupy, jak vykreslit pouze ty objekty, které jsou v záběru kamery – odstřel neviditelných ploch polygonů nebo odstřel objektů mimo záběr kamery a jiné. Urychlovací struktury jsou detailněji popsány v kapitole 2.

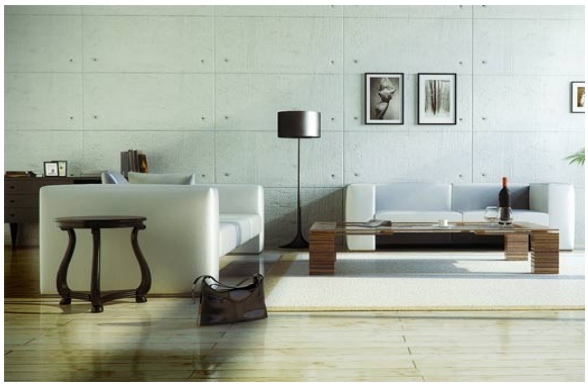


*Obrázek 2: Princip raytracingu. Z každého bodu rastru je vystřelen paprsek směrem do scény, který scénu vzorkuje a počítá tak výslednou barvu bodu. (Reprodukováno z [SSM*05])*

Cíl, kterého chce výzkum v počítačové grafice dosáhnout, je fotorealistické zobrazování scén, nejlépe v reálném čase. Fotorealistických výsledků se dnes již dosahuje, ale dosažení počtu potřebných snímků za sekundu u rozsáhlých scén je problém. Navíc rasterizačním algoritmem fotorealismu nelze skoro dosáhnout. V tomto případě zde hraje nezastupitelnou roli ray tracing. Zajímavým faktem je, že v případě ray tracingu od určitého počtu trojúhelníků ve scéně (cca nad 1 milion) se čas renderování prakticky nemění. Což je do jisté míry logické, protože (na rozdíl od rasterizace) čas vykreslení scény závisí na počtu paprsků vržených do scény, tedy na rozlišení rastru a ne na počtu plošek tvořících ta která tělesa.

Jeho velikou výhodou je to, že má „povědomí“ o celé scéně. Triviálně řečeno vzorkuje celou scénu pomocí paprsků, které vrhá z každého pixelu rastrové mřížky. Navíc, každý paprsek se může ve scéně libovolně pohybovat v podstatě donekonečna a sbírá tak informace o osvětlení, barvě ostatních těles, stínech, fyzikálních vlastnostech materiálů a mnoho dalších. Jednotlivé paprsky jsou na sobě nezávislé a proto se zde otevírá velký prostor pro paralelizaci jednak stále na běžných procesorech a za druhé, nyní velice aktuální - na grafických čipech.

Obrázky 3 a 4 na následující stránce ilustrují schopnosti, kterými ray tracing může disponovat. Mnohdy je už obtížné rozeznat, co je ještě syntetický výstup z nějakého grafického programu a kdy se jedná o reálnou fotografii.



Obrázek 3 a 4: Ukázka fotorealistického výstupu raytracingu, včetně stínů, odlesků, průhlednosti materiálů, refrakce, difrakce a dalších. Ray tracingem lze poměrně věrně napodobit fyzikální podstatu šíření světla ve scéně. (Převzato z [Tak09] a [Pxleyes])

1.4 Výhody a nevýhody ray tracingu

Oproti rasterizaci ray tracing efektivně řeší zastínění objektů – paprsek zasáhne první objekt a pokud se za tímto objektem nachází nějaké další objekty (většinou složitá geometrie o několika stovkách MB či GB), tyto mohou být ignorovány a ušetří se spousta paměti a času při výpočtu. Ray tracing v mnoha ohledech převyšuje rasterizační algoritmus, např. nemusí počítat kompletní odrazovou mapu (reflection map), ale odrazy spočítá pouze tam, kde je to relevantní.

Ovšem i ray tracing se musí vypořádat s několika problémy. Prvním a nejdůležitějším problémem je, jak rozdělit scénu tak, aby paprsek zasáhl jen ty konkrétní objekty a aby tyto objekty byly rychle k dispozici (algoritmus prochází všechny objekty ve scéně a testuje je na průnik s paprskem). Řešením jsou speciální datové struktury, které prostor scény dělí na menší podoblasti, které obsahují objekty samostatně nebo menší skupinu objektů. Protože tyto struktury většinou bývají hierarchické stromy, tento postup dělení scény může zredukovat čas procházení scény na $O(\log n)$. Navíc lze tyto struktury využít i pro jiné výpočty, například k detekci kolizí aj.

Pokud je třeba použít real-time ray tracing, zde vyvstává další problém. Tím je aktualizace těchto struktur na základě pohybu objektů ve scéně. Když se objekty ve scéně pohybují, je potřeba pro každý snímek strom objektů přebudovat v závislosti na nové pozici objektů. Až poté se může spustit vykreslování. Samotná přestavba stromu vyžaduje značnou režii, což se promítá i do výsledného počtu snímků za jednotku času. Lze využít faktu, že velká část scény zůstává statická a nemusí se přebudovat strom úplně celý, ale jen jeho dílčí podčásti.

Kapitola 2

Datové struktury urychlující výpočet raytracingu

Téměř všechny vykreslovací algoritmy v počítačové grafice mají podobný cíl (ať už je tento cíl primární nebo jako vedlejší efekt), a to nezobrazovat části scény, které v danou chvíli nejsou z pohledu kamery vidět nebo nejsou pro výpočet scény důležité. Nejčastěji se jedná o odvrácené strany mnohoúhelníků, objekty mimo záběr kamery nebo objekty, které jsou v zákrytu za jinými objekty. Není tomu jinak ani v případě ray tracingu.

Protože nejjednodušší implementace raytracingu je velice naivní, tzn. algoritmus testuje průnik paprsku i s objekty, které pro daný okamžik nejsou relevantní anebo jsou zcela mimo záběr kamery, je tato metoda implementace pomalá. Proto je potřeba prostor scény vhodně rozdělit na menší celky a zabývat se jen takovými částmi, které jsou z pohledu kamery vidět anebo se jiným důležitým způsobem podílí na výpočtu celé scény. Dělení prostoru zajistí některé z datových struktur, které jsou pro tento účel stvořeny.

2.1 Struktury dělicí prostor

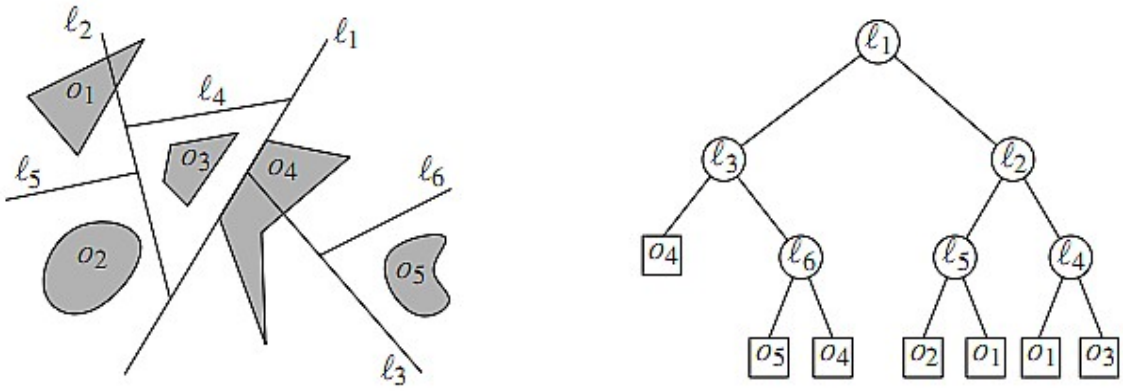
Existuje velké množství prostor dělicích struktur, které se jednak liší svými vlastnostmi a také, k jakému účelu jsou nejvíce vhodné. V drtivé většině mají tyto objekty hierarchickou strukturu, nejčastěji v podobě stromu, obecně n-árního. Právě díky tomu, že tyto struktury jsou povětšinou hierarchicky uspořádané, umožňují lokalizaci daného objektu v logaritmickém čase a tím mohou výrazně zrychlit vykreslování scény. V kontextu této práce není urychlení vykreslování cílem, nýbrž budování těchto dělicích struktur, konkrétně BVH stromu (bounding volume hierarchy tree, viz 2.2 – Strom obalových těles). V následujících odstavcích je shrnut pouze základní nástin nejvíce používaných struktur, které se v počítačové grafice nejvíce používají, s výčtem jejich nejdůležitějších vlastností.

2.1.1 BSP stromy

BSP strom (z angl. binary space partitioning tree) je standardní binární strom používaný ke třídění a vyhledávání polytopů v n-dimensionálním prostoru. Strom jako takový představuje celý prostor scény a každý jeho uzel reprezentuje konvexní podprostor. Každý z uzlů v sobě uchovává poloprostor, jež prostor,

který daný uzel reprezentuje, rozděluje na dvě poloviny. Navíc, v každém uzlu může být uloženo více polytopů. Obecně se lze setkat se stromy, které popisují dvou a třírozměrný prostor, avšak definice stromu není tímto limitována.

Konstrukce BSP stromu je proces, při kterém se daný prostor dělí nadrovinou, která protíná vnitřek daného prostoru. Výsledkem jsou dva podprostory, které mohou být následně dále rozdělovány rekurzivním opakováním takovéhoto dělení. Cílem BSP stromu je pro každou nadrovinu v koncových uzlech přesně určit, zda je před nebo za nadrovinou rodičovského uzlu. To lze zjistit podle normálových vektorů nadroviny. Nadrovina v n-dimensionálním prostoru je n-1 dimensionální objekt, který dělí daný prostor na dva podprostory. Tedy v trojrozměrném prostoru je dělicí nadrovinou 2D plocha, ve dvourozměrném prostoru je tímto dělicím prvkem přímka. Nadroviny uzlů mohou být orientovány v libovolných směrech nebo pouze v ortogonálních. Obecný BSP strom má velikost *O(nlog n)* a lze zkonstruovat v čase $O(n^2 log\ n)$. Průměrný asymptotický čas vyhledání objektu je roven *O(log n)*.[*BCK*09*]

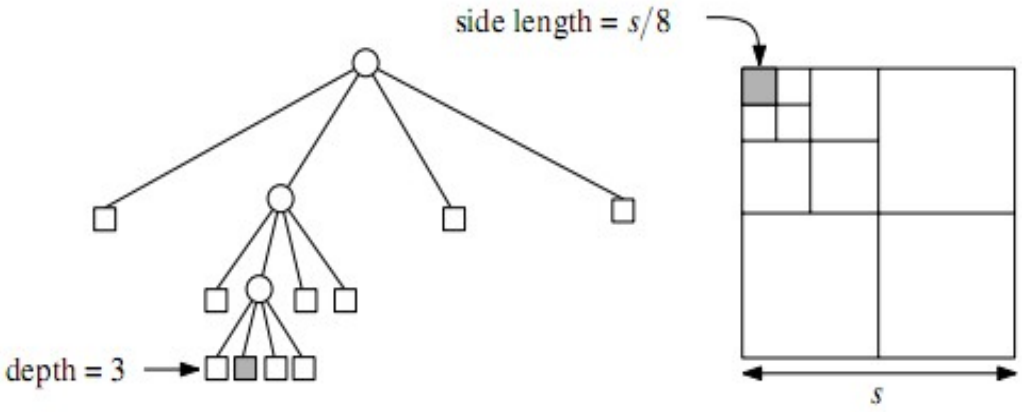


*Obrázek 5: Obecný BSP strom a jeho ekvivalentní grafová reprezentace. (Reprodukováno z [BCK*09])*

BSP stromy jsou velice všestranné, poněvadž dosahují velkého výkonu co se týče klasifikace objektů a jejich vyhledávání. Mají využití v mnoha aplikacích, jako např. odstraňování neviditelných povrchů, urychlování raytracingu až po plánování pohybu robotů. Taktéž se hojně uplatňují při řešení viditelnosti v počítačových hrách. Příklad obecného BSP stromu znázorňuje obrázek 5.

2.1.2 Kvadratické a oktalové stromy

Kvadratický strom (quadtree) je pravidelná dvourozměrná hierarchická struktura, která rekurzivně dělí prostor přesně v polovině na 4 stejně velké podprostory, tedy každý rodičovský uzel má 4 následníky. Jak si lze na obrázku 6 povšimnout, nejvyšší uzel, kořen, opět obsahuje celou scénu (zadaný prostor) a v závislosti na počtu a lokaci bodů či objektů je rozdělen na několik kvadrantů. Každý uzel (rodičovský nebo následník) má podobu čtverce. Uzel v hloubce i reprezentuje čtverec o délce hrany $s/2^i$, kde s je délka hrany kořene (čtverce obklopujícího celý prostor). Obecně lze říci, že kvadratický strom o hloubce d mající množinu n objektů je tvořen $O((d + 1)n)$ uzlů a lze zkonstruovat v čase $O((d + 1)n)$. [*BCK*09*]



*Obrázek 6: Kvadratický strom ve své stromové a odpovídající prostorové struktuře. (Reprodukováno z [BCK*09])*

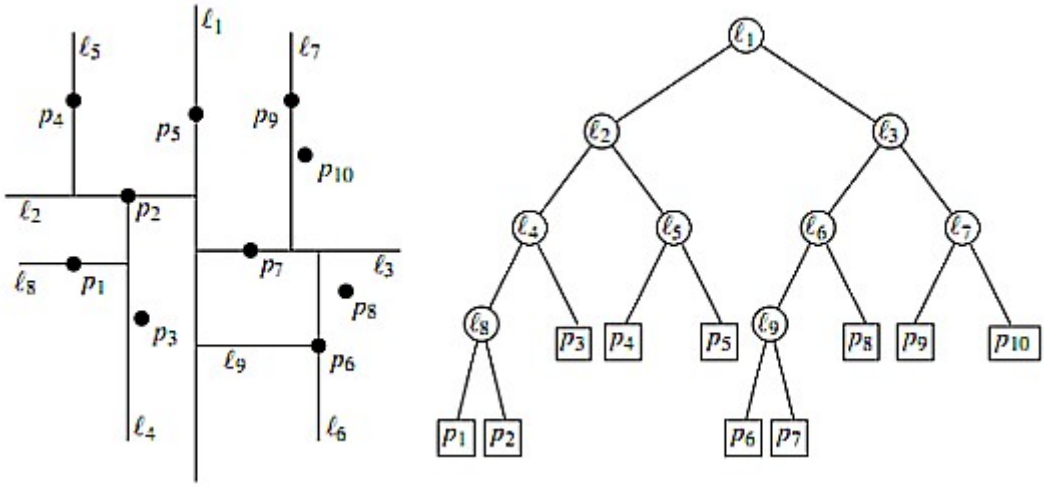
Oktalový strom neboli octree není ničím jiným než rozšířením kvadratického stromu do třírozměrného prostoru. To znamená, že celá scéna je obklopena jednou velkou krychlí (kořen stromu) a ta je dle topologie a složitosti scény rozdělena na několik menších krychlí (uzly), které obklopují dílčí objekty scény. Z toho vyplývá, že každý rodičovský uzel má nyní 8 následníků. Všechny principy výstavby a procházení stromu jsou v podstatě totožné jako v případě kvadratických stromů. Oba typy stromů, kvadratický i oktalový, jsou pouze speciálními případy BSP stromu – dělí prostor pouze v ortogonálních směrech.

2.1.3 K-d stromy

K-d stromy[1] jsou podobně jako kvadratické a oktalové stromy používány jako struktura dělení prostoru. Snaží se reprezentovat množinu bodů v k-dimenzionálním prostoru v takové podobě, která usnadní rychlé vyhledávání bodů, případně jiných objektů. K-d stromy, stejně jako všechny binární vyhledávací stromy, mají významnou výhodu rychlého vyhledávacího času *O(log n)*, mnohem efektivnějšího než naivní lineární algoritmy. Ve velké míře se k-d stromy používají pro ortogonální a jiné rozsahové vyhledávání a vyhledávání typu nejbližší soused (nearest neighbor).

Konstrukční nároky jsou logaritmické vzhledem k počtu objektů (bodů) ve scéně. Způsob, jakým se prostor dělí může být založen buď na heuristice nebo, v častějších případech, na mediánu souřadnic bodů. Prvotní množina bodů se rozdělí v polovině nějakou rovinou, např. podél osy y a vzniknou tak dvě podmnožiny; levá s x-ovými souřadnicemi bodů menšími než hodnota mediánu a pravá s x-ovými souřadnicemi většími než je hodnota mediánu prvotní množiny. Obě tyto podmnožiny se dále dělí stejným principem, nyní ale podél osy x. Tento postup se dále střídavě opakuje, dokud zbývají nějaké body.

Důležitý rys, který odlišuje k-d strom například od stromu oktalového je, že narozdíl od oktalového stromu, který prostor dělí pravidelně na konstantní počet částí, k-d strom dělí prostor na nestejně velké části rovinami, jež jsou rovnoběžné vůči jedné nebo více os souřadného systému. Toto se taktéž liší od obecných BSP stromů.

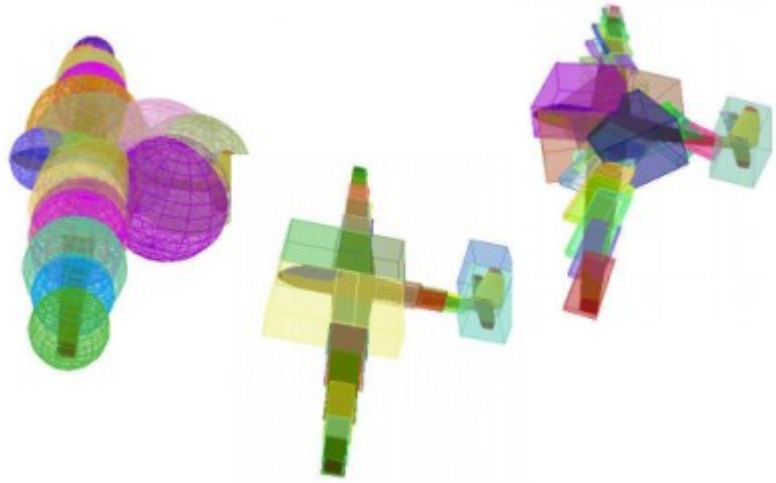


*Obrázek 7: Dvourozměrný k-d strom - grafová a prostorová reprezentace. (Reprodukováno z [BCK*09])*

1 Písmeno „k“ v názvu typu stromu uvádí dimenzi stromu – např. máme-li strom o 2 dimenzích, správně by se měl jmenovat 2d strom, nicméně ustálilo se používání názvu 2-rozměrný kd strom

2.2 Strom obalových těles

Strom obalových těles (angl. bounding volume hierarchy tree, BVH tree) je stromová struktura, která se v různých aplikacích hojně využívá k managementu celé scény. Zejména je vítaným nástrojem jako graf scény (OpenSceneGraph, VRML97, vektorové grafické aplikace), tedy že objekty logicky seskupuje do větších celků (znázorněno na obrázku 9). V některých případech reprezentuje i jejich prostorové uspořádání jakožto struktura dělicí prostor, ale toto nemusí být pravidlem. Taktéž nemusí být striktně binárním stromem, ale obecnou stromovou strukturou s různým počtem větví a uzlů.



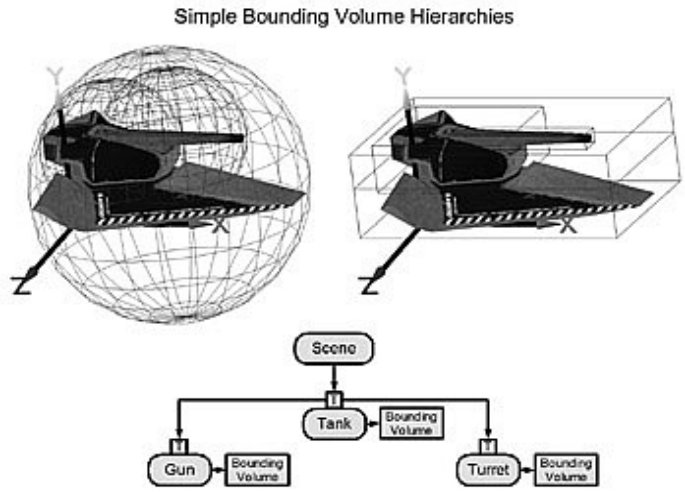
Obrázek 8 : Různé typy BVH stromu ve stejné úrovni dělení. (Reprodukováno z [Soch10])

BVH strom pro uložení geometrie scény, resp. geometrie objektů, jež jsou ve scéně, používá obalová tělesa, tzv. obálky. Jsou to ve své podstatě jednoduchá primitiva, která obklopují složitější geometrii objektů scény a snaží se je co nejlépe a nejtěsněji aproximovat (viz obrázek 8). Obalové těleso v daném uzlu v sobě zahrnuje i obaly ve svých potomcích a každá úroveň hierarchie aproximuje těleso přesněji než úroveň rodičů.

BVH strom by měl splňovat jisté vlastnosti. Nejdůležitější z nich je jistě rychlost jeho výstavby na úplném začátku výpočtu scény. Pokud se musí strom s každým snímkem znovu stavět, je tato fáze kritická. Ve většině případech se výstavba celého stromu provádí pouze jednou a dále, s měnícím se uspořádáním scény, se přestavují pouze jeho dílčí části. S tím souvisí i rychlost jeho přestavby v případě velmi dynamických scén jako jsou interaktivní simulace nebo hry. Toto je ale spíše záležitost algoritmická než složitost vycházející z podstaty této struktury. Jiným neméně významným faktorem je i spotřeba paměti. Je zjevné, že bude

nepraktický takový strom, který se sice velice rychle vystaví, nicméně bude zabírat ohromné množství paměti i u menších modelů. Při dnešních modelech s mnoha miliony trojúhelníků by byla situace s velikostí stromu prakticky neřešitelná, pokud by výsledný strom byl navržen špatně a byl by příliš rozsáhlý. Stejná věc by nastala i s uložením menších modelů, pokud by jich bylo ve scéně mnoho. Kupříkladu současné videohry mají databázi s řádově 10^6 modelů a i když jsou téměř všechny nízkopolygonální, mohla by nastat situace, kdy paměť nebude stačit.

Další požadavky se mohou týkat výběru obalových těles pro konkrétní modely, které bude strom ukládat. Při tom mohou být využity různé heuristiky nebo apriorní znalosti týkající se konkrétního účelu aplikace. V podstatě se jedná o problém výběru co možná nejtěsnějšího obalu, který bude respektovat tvar a topologii ukládaného objektu. Dále, při procesu samotné výstavby hierarchie by tato úloha měla probíhat automaticky, bez nutnosti vnějšího zásahu uživatele.



Obrázek 9: Dva ekvivalentní BVH stromy, každý používá jiné primitivum pro obálky (Reprodukováno z [CKM03])

Existují de facto dva způsoby výstavby stromu, a to shora dolů nebo zezdola nahoru. Metoda shora dolů začíná od kořene (prvním uzlem), který se rekurzivně dále dělí, dokud je k dispozici stále nějaká geometrie objektu (vrcholy, trojúhelníky apod.). Metoda zdola nahoru nejdříve vytvoří obalová tělesa pro jednotlivá grafická primitiva, kterými je objekt tvořen. V dalších krocích se rekurzivně shlukují obálky, které jsou blízko sebe a vytváří se společný obal. Rekurze probíhá tak dlouho, až dostaneme jediný obal pro celý objekt.

Ve fázi vykreslování, když vržený paprsek mine obal vybraného uzlu, pak jistě stejný paprsek mine i potomky daného uzlu a tito mohou být pro další zpracování výpočtu scény vynecháni. Tímto se zjednoduší testy na průnik paprsků objekty scény, stejně tak případné detekce kolizí mezi tělesy samotnými (pokud to daná aplikace vyžaduje). Obalová tělesa jsou povětšinou jednoduchá primitiva jako např. koule, kvádry (osově zarovnané - AABB nebo orientované - OBB), válce a další. Díky tomu, že tato obalová tělesa mají nízký počet vrcholů nebo konstantní průměr, se testy různých průniků a doteků velice zjednoduší a výpočty se mohou tak razantně zrychlit.

Proces stavby BVH stromu využívá pro jeho vytvoření algoritmus SAH (surface area heuristics). Jedná se o tzv. hladový algoritmus. Během procesu stavby stromu se algoritmus pokouší porovnat cenu rozdělení uzlu s cenou, když se uzel dělit nebude. Pokud je lokální cena rozdělení nižší než cena, kdy se uzel dělit nebude, uzel se rozdělí. V opačném případě se aktuální uzel změní na list (koncový uzel). Funkce, která odhaduje cenu rozdělení je

$$C_V(p) = K_T + K_I\left(\frac{SA(V_L)}{SA(V)}T_L + \frac{SA(V_R)}{SA(V)}T_R\right), \quad C_{NS} = K_I T,$$

kde K_T je cena průchodu, K_I je cena průniku trojúhelníků, $SA(V_L)$, $SA(V_R)$, $SA(V)$ jsou plochy povrchu levého, pravého a aktuálního uzlu, T_L, T_R, T je počet trojúhelníků v levém, pravém a aktuálním uzlu, $C_V(p)$ je cena rozdělení aktuálního uzlu a C_{NS} je cena, kdyby se uzel nerozdělil.

Existuje tedy 6T potenciálních dělicích pozic, skládajících se ze tří os (nebo polorovin) s minimální a maximální hodnotou pro každou osu z každého trojúhelníku. Pro každou z os se do nějakého seznamu uloží maximální a minimální souřadnice trojúhelníku, nejlépe s nějakým příznakem, např. START a END, aby bylo později možné určit, kde trojúhelník začíná a končí. Seznamy jsou poté seřazeny vzhledem k souřadnicím.

Pro každou dělicí pozici předpokládejme, že se trojúhelník nalézá v obou uzlech, tedy pro první dělicí pozici bude $T_L = 1$ a $T_R = T$. S postupem na novou dělicí pozici, pokud narazíme na příznak START, zvýší se T_L. Pokud je příznak roven END, v následujícím průchodu se sníží T_R (příznak odpovídá trojúhelníku, který je obsažen v obou uzlech). Nyní, když jsou známy T_L a T_R pro každou dělicí pozici, mohou se vyhodnotit plochy v závislosti na dělicí pozici a určit odhady pro K_T a K_I. S každým průchodem se ohodnotí funkce $C_V(p)$ a uchová se nejlepší cena a dělicí pozice p. Když se vyhodnotí všechny potenciální dělicí pozice, porovná se nejlepší cena s C_{NS} a pokud $C_V(p) < C_{NS}$, uzel se rozdělí.

Kapitola 3

Vícejádrové procesory

Pro co nejvyšší rychlost výstavby BVH stromu na konkrétním počítači by bylo jistě vhodné, kdyby šlo využít maximum toho, co daná stanice nabízí; jinými slovy pokud stanice obsahuje více procesorů (nebo jeden procesor s více jádry), aby tyto zdroje byly plně využity. Tato kapitola přináší postupy, jak vytěžit z CPU jednotek co nejvíce a pokud možno, zaměstnat všechny dostupné procesory a jádra, která se na dané stanici nachází. K účelu zjištění počtu jader procesorů na konkrétním počítači je třeba použít vestavěný inline assembler, který je integrován ve vývojovém studiu Microsoft Visual Studio. K získání informací o procesoru je nutné použít několik assemblerovských instrukcí, které ale lze pohodlně umístit do těla C++ funkce. Dnes již některé překladače nabízí speciální makra (intrinsic funcions nebo jen zkráceně intrinsics), která jsou aliasem konkrétních sekvencí assemblerovských instrukcí a jsou již pro překladač optimalizována.

3.1 Získávání informací o procesoru

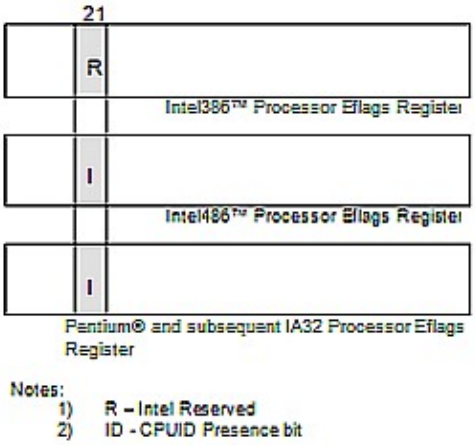
Jazyk symbolických adres (dále jen JSA) je nízkoúrovňový programovací jazyk (angl. assembly language; často nesprávně zaměňován s pojmem assembler – assembler je překladač JSA), který je tvořen symbolickou reprezentací jednotlivých strojových instrukcí a konstant potřebných pro vytvoření strojového kódu programu pro danou procesorovou architekturu. Pro úkol detekce počtu jader procesoru a procesorů obecně budou stačit základní instrukce pro práci s registry, tj. ukládání hodnot do registrů procesoru a jejich čtení, bitové posuny a speciální instrukce cpuid.

Instrukce cpuid je assemblerovská instrukce, která v závislosti na konkrétním nastavení registru EAX v procesoru poskytuje nepřeberné množství údajů jako např. název, model, typ a sériové číslo procesoru, název výrobce procesoru, velikosti L1 a L2 cache a mnoho dalších. S novými procesory a technologiemi instrukce cpuid nabízí stále více dalších informací. Pro problém vícevláknového budování BVH stromu bude stačit alespoň informace o počtu jader procesoru a zda procesor podporuje například i SSE instrukce pro paralelní zpracování dat (SIMD).

3.2 Třída CPUInfo

K tomuto účelu byla vytvořena třída CPUInfo. Mimo jiné také nabízí možnost detekce konkrétní verze SSE technologie (pokud ji procesor podporuje) a také umožňuje získat plný název výrobce procesoru. Jako první krok, který tato třída po inicializaci vykoná, je otestování procesoru, zda vůbec instrukci cpuid obsahuje ve své instrukční sadě a zda ji umí vykonat. V současné době tento krok již není potřeba, poněvadž všechny procesory firmy Intel (pro platformu PC) od modelů i486DX-S tuto instrukci ve své instrukční sadě obsahují. Výjimku mohou tvořit starší procesory značky VIA nebo Cyrix. 64bitové procesory instrukci cpuid obsahují všechny, ať se jedná o jakéhokoli výrobce. Test na přítomnost instrukce cpuid je zde pouze z důvodu zachování kompatibility se staršími čipy.

To, zda procesor obsahuje instrukci cpuid, lze zjistit pomocí příznaku ID v registru EFLAGS na 21. bitu (viz obrázek č. 10). Pokud lze programově nastavit a vymazat tento příznak, pak procesor instrukci cpuid ve své instrukční sadě obsahuje. Assemblerovský kód této rutiny je uveden v příloze B – třída CPU-Info.cpp.



Obrázek 10: Lokace 21. bitu v registru EFLAGS (Reprodukováno z [Intel12])

Assemblerovský kód lze přímo napsat do těla C++ funkce mezi složené závorky { }, před otevírací závorku bloku je třeba uvést direktivu `__asm`, která překladači v MS Visual Studio indikuje, že bude následovat blok assemblerovských instrukcí. MS Visual Studio obsahuje vestavěný překladač MASM (Microsoft Macro Assembler), který používá syntax firmy Intel [Msdn].

V minulosti, kdy strojový překlad zaostával za kódem psaným člověkem, měl vestavěný assembler smysl. Tedy ty části aplikace, po kterých byl vyžadovaný vysoký výkon, bylo vhodné (ale i pohodlné) psát tímto způsobem. V současné době jsou ale překladače na takové úrovni optimalizace, kdy tento způsob psaní kódu (assemblerovské instrukce uvnitř C funkcí) žádný vyšší výkon nepřináší, naopak může program dokonce znatelně zpomalit. Důvod je prostý – překladač nemůže dopředu vědět, jaké operace uživatel-programátor v bloku assemblerovského kódu provede a tedy nemůže provádět téměř žádné optimalizace.

Dále jakoby předpokládá, že programátor bude provádět nestandardní operace s registry a zásobníkem a před vstupem do sekce __asm zazálohuje všechny registry a příznaky na zásobník a po opuštění __asm sekce tyto opět ze zásobníku vyjme. Tato režie je ale tak náročná, že seberychlejší kód psaný v JSA je ve výsledku tak pomalý, že je rozumnější jej psát v čistém C nebo C++ kódu, anebo ručně přilinkovat .asm soubor s kódem během sestavování. Avšak pro úlohy, které je možné vykonat předem, před nějakým časově nebo výkonově kritickým výpočtem, je stále vestavěný inline assembler dobrým nástrojem.

Jaké informace instrukce cpuid vrátí závisí na tom, jakou hodnotu má v sobě uložen registr EAX před zavoláním této instrukce. Např. pro získání názvu výrobce je potřeba volat instrukci cpuid s EAX registrem nastaveným na 0. Řetězec obsahující název výrobce (resp. části názvu) je následně vrácen v registrech EBX, ECX a EDX. Je standardem, že řetězec s názvem výrobce je složen ze 12 ASCII znaků. Tabulka 1 uvádí obsahy registrů EBX až EDX nejběžnějších procesorů.

	Intel	AMD	Cyrix	VIA	NexGen
EBX	`Genu`	`Auth`	`Cyri`	`VIA`	`NexG`
EDX	`ineI`	`enti`	`xIns`	`VIA`	`enDr`
ECX	`ntel`	`cAMD`	`tead`	`VIA`	`iven`

Tabulka 1 – obsahy registrů po zavolání instrukce cpuid s EAX = 0

3.3 Detekce počtu procesorů a jejich jader

V nedávné minulosti byl standardem počítač s jedním procesorem, který obsahoval jedno jádro. Jak narůstala potřeba většího výkonu, výrobci hardwaru přicházeli s novými řešeními a technologiemi, které umožnily základní desky osadit více procesory nebo jedním procesorem s více jádry. Protože některá řešení vyžadují vyšší náklady a procesor je ve výsledku poměrně drahý, vymyslely se postupy, jak další fyzická jádra jednoduše emulovat (kupříkladu technologií HyperThreading od firmy Intel).
V současnosti běžný domácí počítač často obsahuje vícejaderný procesor; jádra mohou být přímo fyzicky přítomna anebo jsou emulována jakožto jádra logická – tedy z vnějšku procesor může vypadat, jako by měl mnohem více jader, i když fyzicky jich má méně.

Najít přesný počet jader v systému nemusí být jednoduché, je třeba nejprve zjistit, kolik jader je v systému nakonfigurovaných, jinými slovy – kolik procesorů se v počítači fyzicky nachází a kolik jich má operační systém dovoleno použít. V běžných desktopových PC budou ve většině případů pro operační systém k dispozici všechny procesory, ale například ve velkých serverech, nebo velkých výpočtových systémech tomu tak již být nemusí.

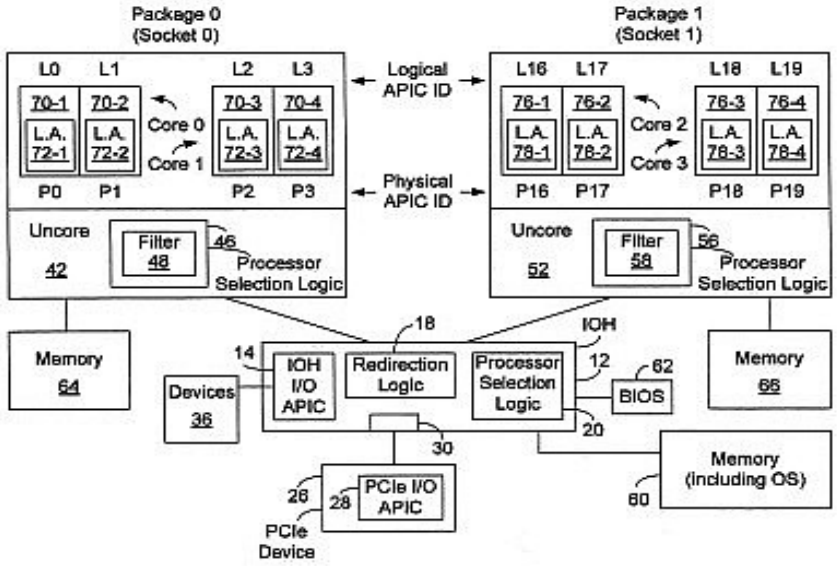
Třída CPUInfo nabízí rozhraní, díky kterému lze zjistit přesný počet všech procesorů přítomných v systému, počet všech fyzických a logických jader. Ke zjištění těchto informací je potřeba použít funkci z Windows API, konkrétně

```
BOOL WINAPI GetProcessAffinityMask(

__in   HANDLE hProcess,
__out  PDWORD_PTR lpProcessAffinityMask,
__out  PDWORD_PTR lpSystemAffinityMask
),
```

která pro běžící proces `hProcess` vrací v argumentu `lpProcessAffinityMask` bitový vektor, v němž je uchováván počet procesorů (nikoli jader), kterými daný proces může být vykonáván, v argumentu `lpSystemAffinityMask` vrací počet všech zjištěných procesorů, které má operační systém k dispozici. Pokud si oba argumenty nejsou rovny (`lpProcessAffinityMask` má nižší hodnotu než `lpSystemAffinityMask`, tedy k některým procesorům nemá proces přístup a nemůže zjistit přesný počet procesorů a jader), třída CPUInfo oznámí, že se v systému nachází pouze jeden procesor.

V dalším kroku je potřeba si udělat přehled o logickém a fyzickém umístění všech procesorů a jader (viz obrázek č. 11) - v této fázi se zjišťuje jejich identifikátor. Procesory a jejich jádra jsou rozděleny do tzv. svazků (packages) a každý z těchto svazků má unikátní identifikátor (APIC ID). Například procesor Intel® Core™ i3-540 disponuje dvěma fyzickými jádry a každé z nich obsahuje dvě logická jádra. To vše je v rámci jednoho svazku. Svazek je v podstatě fyzický čip, který se osazuje do patice základní desky. Pro představu - má-li základní deska patice 2, tedy osazuje se dvěma procesory, systém bude obsahovat celkem 2 svazky. Každý z nich však může mít jiné parametry.



Obrázek 11: Schéma systémové topologie procesorů a svazků. (Reprodukováno z [Pat13])

Každý APIC identifikátor lze dále hierarchicky rozdělit na 3 části: SMT identifikátor, identifikátor jádra a svazku – tyto 3 složky definují tzv. procesorovou topologii (system topology enumeration). SMT úroveň (zkr. pro Simultaneous Multithreading – technika, která umožňuje hardwarovou podporu simultánně provádět více nezávislých vláken; SMT lze v podstatě považovat za onu emulaci více logických jader) představuje nejvnitřnější složku procesorové topologie, tedy je reprezentována nejméně signifikantní částí APIC identifikátoru. Pokud je šířka SMT rovna nule, značí to, že o úroveň výš (v úrovni nad SMT, což je úroveň jádra) existuje pouze jeden logický procesor. Pokud je šířka SMT rovna jedné, o úroveň výš se nachází logické procesory dva atd. Obdobně je to na úrovni identifikátoru jádra - pokud je šířka bitové masky rovna nule, znamená to, že procesor má jedno fyzické jádro, pokud je šířka rovna jedné, má procesor fyzická jádra dvě.

Z těchto údajů lze vhodným algoritmem postupně zjistit příslušný počet logických a fyzických jader na každém svazku a tedy celkový počet všech jednotek v systému. Vhodným algoritmem se rozumí takový algoritmus, který vyextrahuje část APIC identifikátoru korespondující dané hierarchické úrovni (SMT, jádro a svazek) a na základě šířky této části a posunu k 0. bitu APIC ID postupem popsaným v předchozím odstavci získá přehled o celé topologii systému. APIC identifikátor každého svazku lze získat přepnutím vlákna na druhý procesor systémovou funkcí

```
DWORD_PTR WINAPI SetThreadAffinityMask(HANDLE hThread,
                                       DWORD_PTR dwThreadAffinityMask)
```

a opět použít instrukci cpuid s EAX registrem nastaveným na 1. Výsledek bude uložen v registru EBX.

Zjednodušeně vypadá celkový algoritmus následovně:

- zjisti počet fyzických procesorů v systému (funkcí `GetProcessAffinityMask()`)
- pokud se počet procesorů, na kterých může aktuální proces běžet nerovná celkovému počtu procesorů v systému, skonči (algoritmus vrátí pro počet procesorů jedničku, protože alespoň na jednom procesoru se tento algoritmus provádí)
- přepínej proces na každý svazek a zjisti jeho APIC ID
- z konkrétního APIC ID podle šířky bitové masky každé úrovně zjisti, kolik logických procesorů a fyzických jader každý svazek má

Takto lze velmi přesně zanalyzovat konfiguraci systému a podle počtu zjištěných jader v dalších fázích podle požadavků aplikace generovat tolik vláken, kolik bude potřeba. Výsledkem tohoto postupu je počet jader systému včetně logických jader, které emuluje technologie HyperThreading. Takovýto postup není zcela triviální a k návrhu tohoto algoritmu je potřeba znát detaily architektury konkrétních čipů nebo alespoň dané řady čipů, což je možné díky velice podrobné technické dokumentaci, kterou na svých webových stránkách poskytuje například společnost Intel Corporation.

3.4 Práce s vlákny

Na spuštěnou aplikaci (zavedený program v operační paměti) lze pohlížet jako na tzv. proces. Operační systém pro každý proces vytváří virtuální adresní prostor, což je určitý blok paměti, ve kterém spuštěný proces existuje a používá vyhrazenou paměť. Každý proces je tvořen alespoň jedním vláknem; to je objekt operačního systému, který provádí strojový kód, kterým je aplikace tvořena. Nicméně proces se může sestávat z několika vláken, která mohou běžet současně nebo postupně. Během životního cyklu procesu mohou vlákna libovolně vznikat a zanikat, vždy ale musí existovat primární vlákno. To je vlákno, jež vznikne spuštěním procesu a zavedením do operační paměti a z něj mohou dále vznikat další, pracovní vlákna. Ale i ta mohou dále dávat za vznik novým vláknům, není to výsada pouze vlákna primárního.

Zde je důležité si uvědomit, že všechna vlákna vytvořená v rámci jednoho procesu využívají tentýž virtuální adresní prostor. Protože adresní prostor je definován procesem a nikoli vlákny, je možné sdílet data procesu více vlákny. Tato skutečnost má za následek to, že vlákna mohou provádět stejný programový kód nad jinými daty. Pokud tento fakt nebude brán na zřetel při vývoji aplikace, mohou si vlákna navzájem data modifikovat a v aplikaci se budou objevovat chyby, které lze velmi ztěžka odhalit. Avšak sdílení společných dat nemusí být vždy špatné, sdílenými daty lze např. zajistit komunikaci a synchronizaci mezi vlákny.

3.4.1 Synchronizace vláken

Způsobů, jak vyřešit sdílení dat v adresním prostoru procesu existuje celá řada. Nejčastějším problémem je zajistit, aby si vlákna sdílená data navzájem nepřepisovala. Navíc nelze předem určit, v jaký okamžik začne konkrétní vlákno vykonávat svou činnost a kdy skončí. Tato režie je plně v kompetenci operačního systému a je nepredikovatelná.

Dále do souběžnosti vláken vstupuje přerušení, které může mít vliv na pořadí operací a ovlivnit tak korektnost programu. Pro příklad - operační systém vláknu odebere procesor v okamžiku, kdy již provedlo svou operaci, ale ještě nezapsalo správný výsledek. Poté, co druhé vlákno provede celou operaci, první vlákno pokračuje v činnosti, zapíše tedy svůj výsledek do proměnné, čímž ale přepíše výsledek práce předchozího vlákna. Výsledkem bude tedy jiná, nesprávná hodnota než by měla být, pokud by první vlákno dokončilo svou činnost bez přerušení.

Jedním ze způsobů zajištění synchronizace vláken je kritická sekce. To je blok kódu, který musí být vykonán pouze jedním vláknem v daném časovém okamžiku. Další vlákna do tohoto bloku nemají přístup. Až po opuštění kritické sekce prvním vláknem do ní mohou vstoupit další, která potřebují daný kód vykonat, ale opět jednotlivě. S pojmem kritické sekce souvisí tzv. mutex (zkratka z angl. mutual exclusion – vzájemné vyloučení).

Mutex je algoritmus, jak zabezpečit kritickou sekci. Může být na něj pohlíženo i jako na jakýsi zámek, který v jeden časový okamžik může být uzamčen pouze jedním vláknem. Pokud je mutex uzamčen nějakým vláknem a jiné se pokusí mutex také uzamknout, pak se vlákno zablokuje. Až po odemknutí mutexu předchozím vláknem, které jej uzamklo, se čekající vlákno odblokuje a může mutex, potažmo kritickou sekci použít.

Rozdíl mezi kritickou sekcí a mutexem je v tom, že kritická sekce je lokální pro daný proces, tzn. kritická sekce je sdílená mezi vlákny v rámci jednoho procesu, ale vláknům jiných procesů je nepřístupná. Naopak mutex je globální v celém systému a všechny procesy a jejich vlákna mají do něj přístup. Režie kritické sekce je navíc i mnohem nižší než režie mutexu. Na druhou stranu na vzájemné vyloučení je možné čekat daný časový limit, zatímco při čekání na uvolnění kritické sekce není možné stanovit žádný limit. Může se tedy stát, že vlákno v kritické sekci může být libovolně dlouho a může tak vzniknout uváznutí. Mutexy, respektive různé implementace mutexů (monitory, semafory) mívají často podporu přímo v operačních systémech.

Implementace mutexu se navzájem liší svou výkonností a případem použití. Například semafor je vhodné použít, pokud je potřeba kritickou sekcí zabezpečit větší část kódu, ale procesor musí přepnout na další vlákno celý kontext a tato režie je výkonově docela náročná. Naproti tomu tzv. spinlock je vhodné použít tam, kde se kritická sekce uzamkne na velmi krátký okamžik. Princip spinlocku je takový, že když nějaká vlákna chtějí přistoupit ke zdroji už drženém jiným vláknem, přepnou se do cyklu, ve kterém čekají na uvolnění drženého zdroje. Při tomto aktivním čekáním (busy-waiting) ale čekající vlákna dále spotřebovávají prostředky CPU. Ale i spinlock má svá negativa. Jestliže bude spinlock uzamčen po delší dobu, zabraňuje dalším vláknům v pokračování a plánovač operačního systému je donucen přehodnotit prioritu vláken a doby jejich spuštění a přerušení. Navíc hrozí, že vlákno, které uzamklo spinlock, bude přerušeno a ostatní vlákna budou v cyklech čekat na uvolnění spinlocku, zatímco přerušené vlákno nebude dělat žádný pokrok k tomu, aby spinlock uvolnilo.

3.4.2 Vlastní manipulace s vlákny

Prostředky pro vznik a práci s vlákny se v prostředí Windows nachází v knihovně Windows API. Zde se nalézá funkce `CreateThread()`, která zapříčiní vytvoření vlákna. Podrobný popis jejích argumentů lze najít na [Thread]. Nejdůležitějšími parametry funkce jsou ukazatel na funkci, kterou bude vlákno provádět a ukazatel na strukturu s daty, ve které se vláknu mohou předat různé inicializační proměnné, většinou argumenty funkce, kterou bude vlákno vykonávat. Během vytváření vlákna lze též určit, zda má vlákno začít pracovat ihned nebo zda se vytvoří vlákno uspané, které se aktivuje později jinde v kódu.

Během doby, kdy vlákno vykonává nějakou činnost, lze se na něj dotazovat a získávat tak o něm další informace jako například jeho identifikátor, handler, prioritu nebo zda vlákno skončilo. Prioritu vlákna lze během doby jeho běhu libovolně měnit. Dokonce lze programově nastavit, kolik jader systému budou vlákna procesu využívat a která jádra to budou. Obecně se ale manuální nastavování běhu vláken uživatelem moc nedoporučuje a toto se doporučuje plně přenechat na operačním systému a ten pomocí svých plánovačů a dalších nástrojů nejlépe a efektivně automaticky přiřadí aplikační vlákna jádrům. Windows API nabízí širokou škálu dalších možností a funkcí, jak s vlákny pracovat, ale to už je mimo rámec této práce.

Dobrým programátroským zvykem je, zvlášť přihlédneme-li k modernímu objektovému programování, vytvořit nějakou ucelenou třídu, která by do sebe práci s vlákny zapouzdřila. Projekt od společnosti Nvidia Corporation, jež tato práce rozšiřuje [LAK09], již obsahuje třídu `Thread`, která umožňuje základní, ale dostatečnou práci s vlákny a nabízí několik metod, které usnadňují multivláknové programování (vytvoření vlákna, uspání, pozastavení, zjištění ID aktuálně bežícího vlákna a další). V souboru s třídou `Thread` jsou ještě navíc implementována základní synchronizační primitiva (monitor, spinlock), která jsou při paralelní stavbě BVH stromu použita.

Existuje i robustní knihovna od firmy Intel s názvem „Threading Building Blocks“, která také usnadňuje vývoj multivláknových aplikací a navíc je vysoce optimalizovaná a navržená právě pro procesory Intel [Itbb]. Obsahuje různá synchronizační primitiva a taktéž datové struktury navrženy pro bezpečnou práci s vlákny (např. thread-safe for cyklus nebo různé kontejnery podobné těm ze standardní knihovny šablon, ale navržené pro bezpečné použití ve vícevláknových aplikacích). Je napsána v jazyce C++ a její velkou výhodou je použitelnost na většině operačních systémů.

Kapitola 4

Využití paralelismu při budování akceleračních struktur

Při budování BVH stromu se princip paralelního zpracování, resp. tvorby jeho větví sám nabízí. Výstavba BVH stromů v této práci v jednotlivých implementacích (jedno i vícevláknových) probíhá směrem do hloubky, tzn. je-li možnost pro aktuální uzel vytvořit nějakého následníka, takový následník se vytvoří a v něm se testuje stejná možnost.

Sekvenční algoritmus v novém uzlu, v případě, že má stále k dispozici nějakou geometrii zpracovávaného modelu, vytvoří dva následníky, vybere si jednoho z nich a v něm dále pokračuje. Pokud je již dosaženo ukončovacích kritérií (např. potřebné hloubky stromu nebo určitý počet trojúhelníků v uzlu), algoritmus vytvoří místo uzlu list (koncový uzel) a backtrackingem se vrací o úroveň výš a pokračuje následníky, které ještě nezpracoval. Takovýto přístup je jednoduchý a funguje. Nicméně efektivita není vysoká, protože v jednom okamžiku se zpracovává pouze jeden jediný uzel.

4.1 Více vláken

Rozšířením jednovláknové implementace o paralelismus se otevírají nové možnosti, jak rychle a efektivně vystavět BVH strom. Na druhé straně zde vyvstávají nové problémy, které jednovláknová implementace nemusí řešit – synchronizaci a souběh vláken. Jak už bylo v kapitole 3.4 vysvětleno, předem nelze určit, v jakém okamžiku se začne konkrétní vlákno provádět a kdy skončí a podle tohoto nelze navrhnout úlohu, která předem očekává, že v daný čas to či ono vlákno skončí a předá výsledek. Lze ale zajistit, aby vlákna spolu komunikovala nebo na sebe počkala – to bude popsáno dále v této kapitole.

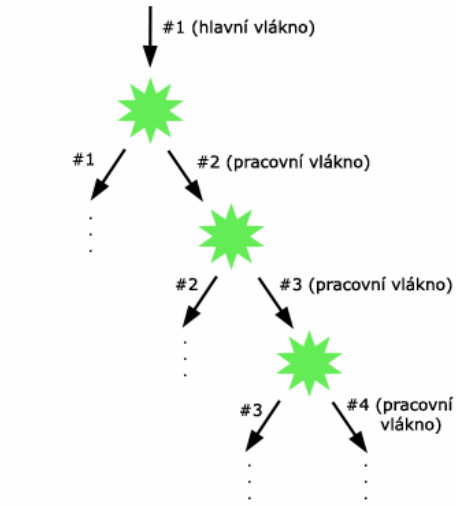
Filozofie vícevláknového přístupu spočívá ve vytváření tolika pracovních vláken, kolik je v systému jader, na kterých vlákna procesu stavby hierarchie mohou běžet. V okamžiku, kdy se v aktuálně zpracovávaném uzlu mají vytvořit dva jeho potomci, respektive levý a pravý podstrom hierarchie, je nyní vhodný moment vytvořit nové vlákno a tomuto novému vláknu předat jeden z podstromů ke zpracování (předá se de facto ukazatel na jednoho potomka aktuálního uzlu, který slouží jako kořen budoucího podstromu). V ideálním případě se bude každé vlákno zpracovávat na samostatném jádru, což ale nelze obecně zaručit, protože přidělování vláken jádrům má na starosti operační systém a může se tak stát, že na

jednom jádru poběží více než jedno vlákno aktuálního procesu. Toto může nastat třeba když jedno z jader bude zpracovávat intenzivní úlohu nějakého jiného procesu. Během testování metod se však ukázalo, že vlákna procesu stavby BVH stromu opravdu pokaždé pracovala separátně na samostatném jádru.

4.2 Vícevláknová implementace nerovnoměrného přidělování vláken

Princip metody spočívá ve zvolení jednoho z podstromů, ve kterém se budou moci dále, v dalších úrovních rekurze, vytvářet nová pracovní vlákna. První okamžik, kdy se začne nové vlákno vytvářet je tehdy, když se zpracovává kořen celé hierarchie, tzn. téměř ihned, kdy se začne BVH strom stavět. Aby byl algoritmus více flexibilní a lépe se adaptoval na prostorové rozložení scény, po prvním dělní scény SAH algoritmem se určí podle objemu dat, kterému z podstromů (zda pravému či levému) bude umožněno, aby v dalších etapách stavby mohl vytvářet nová pracovní vlákna. Pomocné třídy, které se podílejí na stavbě hierarchie nabízí prostředky, pomocí kterých je možné zjistit, jaký objem dat, respektive kolik trojúhelníků od dané úrovně rekurze budou budoucí podstromy hierarchie obsahovat.

Samotný proces, kdy se vlákna vytváří, žádný strom vláken nebo jinou datovou strukturu, která by obsahovala instance vláken, fyzicky nebuduje. Jedná se pouze o rekurzivní zanořování, kdy jedno vlákno vytváří vlákno jiné a tento proces vytváření vláken si lze představit jako jakýsi strom vláken, i když žádný takový strom v podstatě neexistuje (viz obrázek 12).



Obrázek 12: Schéma nerovnoměrného vytváření vláken během stavby BVH stromu. V tomto případě se vlákna vytváří pouze pro nejpravější podstromy BVH stromu

Vytvoření všech vláken proběhne prakticky ihned, v počáteční fázi stavby hierarchie. Staví-li se strom například na čtyřjádrovém procesoru, jedno vlákno už běží (je to vlákno aplikační), a zbývá vytvořit tři nová vlákna. Ta se vytvoří během zpracování tří prvních uzlů, kořene a dvou potomků. Jednoduchý pseudokód této úlohy může vypadat následovně:

```
function buildNode(cores: integer):
 if (card(cores) > 0) do
    create new thread t;
    rightNode = t.start(buildNode(cores - 1));
    leftNode = buildNode(0);
 else
    leftNode = buildNode(0);
    rightNode = buildNode(0);
 endif;
end;
```

Na začátku stavby stromu se podle prvního rozdělení scény SAH algoritmem zvolí, který podstrom bude moci vlákna vytvářet a který nikoli. Určeme, že například v pravém podstromu je více geometrie než v levém. Od této chvíle budou moci jen pravé podstromy v dalších úrovních rekurze vytvářet nová vlákna. V argumentu funkce `buildNode` (funkce generující uzly) se rekurzí přenáší do další úrovně, kolik jader je ještě k dispozici, tzn. kolik vláken se může vytvořit, aby každé jádro dostalo jedno vlákno. Tento počet se s každou úrovní rekurze sníží o jedna, protože se vždy v pravém uzlu vytvoří nové vlákno. Levým uzlům je vždy v argumentu funkce předávána nula – tím je zajištěno, že levé uzly nebudou nikdy vlákna vytvářet. Pokud volná jádra již nejsou k dispozici (proměnná `cores` je rovna nule), pak žádné uzly, ani levé ani pravé, nová vlákna nevytváří.

V konečné fázi, kdy je hierarchie téměř postavena, se zpětně, jak se rekurze zásobníkově vrací, všechny uzly a podstromy navzájem pospojují do jednoho konečného BVH stromu. V této fázi je potřeba, aby ta vlákna, která v rekurzi vznikla dříve a skončila i svůj výpočet dříve, než vlákna, která vznikla později, na tato pozdější vlákna počkala. Kdyby na sebe vlákna nečekala, dřívější vlákna by sice vrátila nějakou část hierarchie, ale s neplatnými ukazateli na části, které v době vzniku části aktuálního podstromu ještě neexistovaly a tyto neplatné ukazatele by mohly zapříčinit dokonce i pád programu. Čekání vláken umožňuje metoda `join()` třídy `Thread`.

4.3 Vícevláknová implementace rovnoměrného přidělování vláken

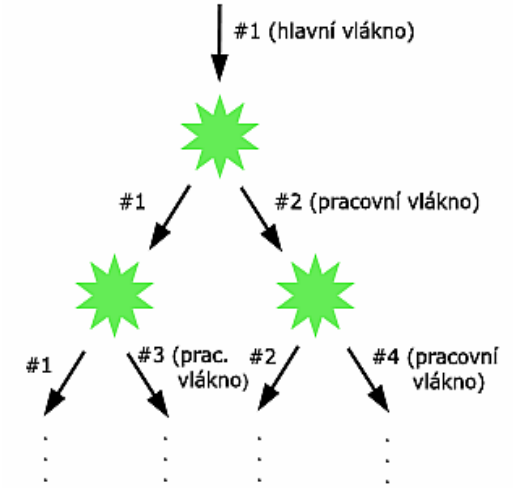
Implementace rovnoměrného přidělování vláken během stavby BVH stromu se velmi podobá implementaci nerovnoměrného přidělování vláken, rozdíl spočívá v pravidelném rozdělení tvorby vláken v čase tvorby uzlů. Nyní mohou vlákna vytvářet všechny uzly, ne jako to bylo v předchozím případě, kdy vlákna mohly generovat pouze uzly na předem dané straně stromu. Zde již nehraje roli poměr objemu geometrie scény v budoucích podstromech. Myšlenka se opírá o předpoklad, že ve chvíli, kdy se zpracovává kořen nebo uzly v počátečních úrovních rekurze, bude v obou podstromech dostatek geometrie na to, aby bylo vůbec výhodné paralelní budování započít, i přes zvýšenou režii, která s paralelismem souvisí.

I zde se opět předává parametr počtu volných jader prostřednictvím argumentu funkce `buildNode`, avšak výpočet hodnoty volných jader se zjišťuje jinak. Počet jader se pro každý uzel na další úrovni rekurze vydělí celočíselně dvěma a tento výsledek se předá jako výchozí hodnota počtu volných jader v další úrovni rekurze. Vysvětlení pseudokódem a obrázkem č. 13. Pro zjednodušení má funkce `buildNode` v pseudokódu pouze jeden argument.

```
function buildNode(cores):
var newCoresNumber: integer;
 if (card(cores) > 0) do
    create new thread t;
    if(cores.odd) do
      newCoresNumber = cores div 2;
      leftNode = t.start(buildNode(newCoresNumber));
      rightNode = buildNode(newCoresNumber);
    else if(cores.even) do
      left = newCoresNumber = (cores div 2)-1;
      right = newCoresNumber = cores div 2;
      leftNode = t.start(buildNode(newCoresNumber));
      rightNode = buildNode(newCoresNumber);
    endif;
 else
    leftNode = buildNode(0);
    rightNode = buildNode(0);
 endif;
end;
```

Dělení dvěma se zde provádí z důvodu, že případě, že by argument funkce `buildNode` byl lichý, zajímá nás dolní celá část. Při prvním volání funkce během tvorby prvního uzlu – kořene – je funkci předán celkový počet jader snížený

o jedna. To proto, že už jedno vlákno existuje, a to vlákno hlavní, aplikační, což je vlákno spuštěného programu. Pro představu - máme-li čtyřjádrový procesor, bude před započetím stavby hierarchie v argumentu `buildNode` počet roven třem. Ještě tři nová vlákna mohou být vytvořena pro tři volná jádra. Číslo tři se vydělí dvěma, což vrátí výsledek roven jedné a ten se předá jako vstupní argument funkce `buildNode` pro oba budoucí nové uzly. V případě, že by se v systému nacházel lichý počet jader (málo častá varianta), bude argument funkce `buildNode` na začátku stavby sudý a jeden z podstromů bude moci vytvořit o jedno vlákno více než druhý podstrom.



Obrázek 13: Schéma rovnoměrného vytváření vláken během stavby BVH stromu. Vlákna se symetricky distribuují do každého podstromu BVH stromu.

To znamená, že v aktuálním uzlu se vytvoří nové vlákno, které obsadí další jádro a dvě jádra zbývají, tedy každý potomek aktuálního uzlu může vytvořit ještě jedno vlákno. Výsledkem je rovnoměrné rozložení vláken ve všech podstromech. Zbytek průběhu stavby hierarchie obálek, včetně čekání vláken na sebe sama se již neliší od způsobu provedení v předchozí metodě.

4.3 Komplikace plynoucí z vícevláknového přístupu – metoda s rovnoměrným a nerovnoměrným přidělováním vláken

U obou předchozích metod se vyskytla menší překážka v podobě dvou členských proměnných, které se musí ochránit před vzájemnou modifikací vlákny mezi sebou. Jedná se o celočíselnou proměnnou `dimension`, která v sobě nese informaci o polorovinách, kterými SAH algoritmus dělí scénu a proměnnou `m_right-`

`Bounds`, což je pole AABB obálek, které určují pravé ohraničení trojúhelníků. Pokud by se tyto dvě proměnné nechaly sdílené mezi vlákny, docházelo by k chybám za běhu programu, BVH strom by nemusel být vybudován správně (chyběly by části scény, model by obsahoval díry) a mohl by dokonce nastat i pád programu.

Je tedy potřeba, aby každé vlákno mělo svoji vlastní nezávislou lokální paměť s těmito dvěma proměnnými – jakýsi kontext – a to přesně v okamžiku, kdy se nové vlákno vytváří. Těsně před vytvořením nového vlákna musí vzniknout kopie obou proměnných. V ní je uložen aktuální stav obou proměnných a tato kopie je výchozím bodem pro funkce v novém vlákně, které obě proměnné používají a vlastně i výchozím bodem pro celý budovaný podstrom. Pro každé vlákno a tedy i podstrom je kopie těchto proměnných unikátní a není potřeba, aby pro správnou stavbu podstromu bylo nutné použít informace z jiné kopie.

Implementačně toho bylo dosaženo tak, že kopie proměnné `dimension` byla předána jako argument funkce `buildNode` volané v novém vlákně a ta ji poté předává dál do dalších úrovní rekurze. Tento princip využívá faktu, že v multivláknové aplikaci má každé vlákno svůj vlastní kontext, tj. oddělenou část zásobníku, kam se ukládají argumenty funkcí, různé příznaky a ukazatele na hodnoty předávaných funkcím (zejména registry EBP, ESI, EDI a ESP). Proto se může proměnná `dimension` tímto způsobem propagovat v rekurzi jako argument funkce – není způsob, jak by si v tuto chvíli mohla vlákna tuto proměnnou navzájem modifikovat. Naproti tomu členské proměnné tříd (tj. proměnné instance tříd) jsou sdílené metodami, které je nějakým způsobem používají, ve všech vláknech.

Proměnnou `m_rightBounds` by také šlo předávat funkcím jako argument, nicméně, protože se jedná o dynamické pole AABB objektů, staticky alokované a poměrně velké, nebyl by mechanismus předávání příliš efektivní z důvodu kopírování velkého objemu dat při každém volání funkce, která by jej používala jako svůj argument. Tento problém lze vyřešit přidáním univerzálního ukazatele typu `void` ve třídě `Thread` a ten bude svázán s korespondující kopií AABB pole. Poté lze pouze znát identifikátor vlákna (lze kdykoli zjistit funkcí `GetCurrentThreadId()` z Windows API), který byl vygenerován při vytvoření vlákna a v nějakém asociativním kontejneru podle klíče <id vlákna, instance vlákna> dohledat odpovídající instanci třídy `Thread` pro aktuální vlákno a přes ukazatel přistoupit k AABB poli.

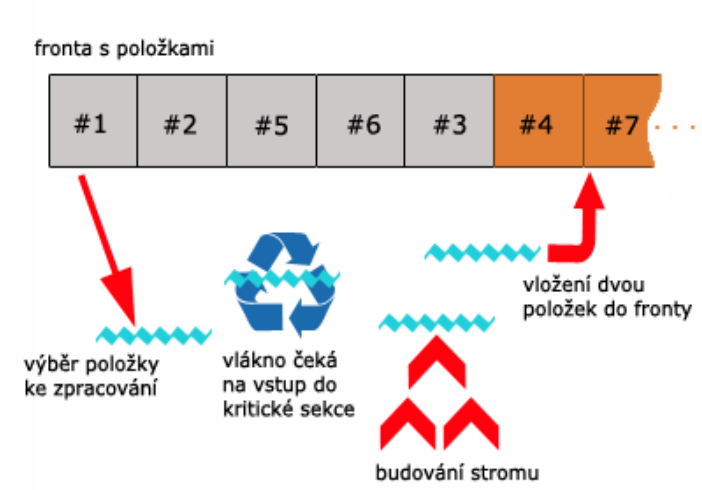
4.4 Vícevláknová implementace stavby BVH stromu s užitím fronty uzlů

Třetí z metod používá ke stavbě BVH stromu frontu uzlů, ze které si vlákna uzly postupně vybírají a zpracovávají. Přesněji řečeno – do fronty nejsou ukládány přímo uzly stromu jako takové, ale pouze informace nutné k vytvoření jednoho uzlu stromu. Tento návrh implementace vede k tomu, že se zátěž výpočtu rovnoměrně rozdělí co nejefektivněji mezi různá vlákna. Vlákna nepřetržitě pracují do té doby, dokud jsou ve frontě nějaké informace.

Potřebné informace pro vznik uzlu hierarchie jsou sloučeny ve struktuře `NodeContext`. Každá struktura tvoří položku fronty a obsahuje unikátní informace pro vznik uzlu hierarchie a tedy jedna položka fronty odpovídá jednomu konkrétnímu uzlu hierarchie. Struktura mimo jiné obsahuje obecné informace jako např. úroveň zanoření v rekurzi (což je stejné jako úroveň patra stromu, kde se bude uzel nacházet), počáteční a koncové indexy do pole trojúhelníků tvořících model, ukazatel na aktuální instanci třídy (nahrazuje ukazatel `this` v nestatických metodách) a zejména pak ukazatel na předchůdce uzlu a příznak, zda se bude nový uzel nacházet v podstromu nalevo či napravo. Na základě těchto informací poté každé vlákno vytvoří vnitřní anebo koncový uzel (list) a správně uzel přiřadí svému předchůdci, tzn. na správné místo ve stromu.

Algoritmus začíná vytvořením kořene stromu pomocí metody `buildNode` s počátečními parametry. Metoda `buildNode` během vytváření kořene též naplní frontu prvními dvěma položkami. Následně se vytvoří dostatečný počet pracovních vláken, respektive instance třídy `Thread`. Dostatečným počtem se rozumí takový počet, který odpovídá počtu logických jader v systému. Do tohoto počtu je zahrnuto i vlákno hlavní, tedy vlákno procesu. Po vytvoření vláken tato ihned začnou paralelně odebírat z fronty položky a z informací, které položky uchovávají, se začnou generovat uzly BVH stromu (viz obrázek 14). Během vytváření uzlu BVH stromu se do fronty vždy přidají dvě nové položky (informace pro budoucí levý a pravý uzel – následníka aktuálně tvořeného uzlu). V případě, že zrovna generovaný uzel bude uzel koncový, tj. list stromu, žádné položky se do fronty nepřidávají. Koncový uzel se pouze naváže na svého předka.

Zde ovšem vyvstávají dva důležité aspekty. První z nich je, že od určitého okamžiku, kdy je v budoucích uzlech poměrně málo geometrie ke zpracování, je další přidávání položek (informací ve strukturách `NodeContext`) do fronty neefektivní a spíše kontraproduktivní, protože režie zahrnující přidávání a odebírání položek do a z fronty převyšuje užitný výkon a degraduje ho tak. Navíc, v zájmu zachování souběžnosti vláken, je potřeba frontu před přidáním a odebrá-



Obrázek 14: Nákres algoritmu stavby BVH stromu s využitím fronty. Vlákna cyklicky dotazují frontu a vybírají z ní struktury s informacemi pro vznik uzlu BVH stromu. Při vzniku nekoncového uzlu se do fronty vloží dvě nové položky ke zpracování dalšími vlákny. Pokud je fronta prázdná, každé vlákno krátký okamžik počká a dotáže se fronty znovu. Jestliže i přesto je fronta prázdná a neexistuje žádné další vlákno, které by signalizovalo ostatním, že provádí výpočet, dotazující se vlákno ukončí svou existenci.

ním položky uzamknout kritickou sekcí a tato operace ještě více (ne však až tak dramaticky) zpomaluje celkový výkon celého procesu výstavby stromu. Je tedy třeba určit, od jakého okamžiku je vhodné zbytek stromu nebo jeho dílčí části budovat jako u obou předchozích metod, tedy rekurzí.

Existuje několik návrhů, jak tento problém vyřešit. Například práce [CKL*10] pojednává, že optimální doba, kdy je vhodné začít budovat podstromy pomocí rekurze, nastává tehdy, když je počet uzlů budovaného stromu roven nebo větší než počet jader v systému (zde není přímo uvedeno, zda je také použita fronta nebo ne). Článek [SSK01] zase uvádí, že po experimentování s velikostí fronty (v kontextu článku task pool) se jeví jako nejpraktičtější velikost fronty jako nějaký násobek počtu jader v systému. Konkrétně pro 4jádrový porocesor byla nejoptimálnější velikost fronty 256 položek. Vzhledem k faktu, že oba články používají jinou implementaci SAH algoritmu, i v porovnání s touto prací, nelze oba přístupy jednoduše převzít.

Proto v rámci této práce byla taktéž provedena heuristika na několika modelech v kombinaci s podmínkami, za kterých má nastat rekurzivní stavba podstromů. Pro testy byly vybrány modely *Happy Buddha* (1087 K), *Fairy Forest* (174 K), *Stanford Dragon* (50 K), *Soda Hall* (2169 K), *Conference* (282 K) a *Icosahedron* (1800 K). Byly vybrány čtyři podmínky pro započetí rekurzivního dělní a každá podmínka byla na všech modelech otestována ve čtyřech bězích. Zvolené

podmínky pro započetí rekurzivního dělení jsou vypsány v následujícím přehledu; Q značí frontu, T počet vláken nebo jader, L hloubku rekurze:

1. $|Q| >= T$
2. $|Q| >= 2*T$
3. $|Q| >= T^T$
4. $L > 4$

První a třetí podmínka byla přejata z již zmíněných článků [CKL*10] a [SSK01], nicméně se neukázaly jako nejvíce vhodné. První podmínka byla dokonce nejhorší ze všech čtyř vybraných. Druhá a třetí podmínka vykazovaly víceméně podobné výsledky, respektive prokazatelně nepřevládala ani jedna z nich. Situace se ale mění při srovnání druhé a čtvrté podmínky. Zde nastal jev, kdy každá z obou podmínek podává lepší výkon v závislosti na velikosti modelu. Malým scénám, do ~500K (tedy Stanford Dragon, Conference a Fairy Forest) prokazatelně svědčí čtvrtá podmínka, zatímco pro velké scény (Icosahedron a Soda Hall) se evidentně jeví lepší podmínka druhá. V případě středně velkých scén, v rozmezí ~500 K až ~800 K (Happy Buddha), je výkon vyrovnaný.

Výsledkem tedy je, že na základě provedené heuristiky jsou v implementaci zkombinovány podmínky 2 a 4 a podle velikosti scény je použita buď první nebo druhá z nich. Princip, který obě podmínky od sebe odlišuje se skrývá v opětovném přidávání struktur s informacemi pro uzly, když se fronta zmenší. Podmínka založená na hloubce rekurze garantuje to, že jakmile se rekurze dostane pod čtvrtou úroveň zanoření, od té doby se už nikdy do fronty nebude nic přidávat a zbytek stromu se dobuduje pouze pomocí rekurze. Naopak podmínka číslo 2 říká, že pokud velikost fronty bude rovna dvojnásobku počtu spuštěných vláken, začne se strom budovat rekurzivně, ale jen do doby, kdy velikost fronty klesne pod tuto úroveň. Potom se mohou do fronty přidávat další položky.

Druhý aspekt, který vyvstal z implementace zpracování uzlů stromů pomocí fronty je potřeba zajistit určitou úroveň komunikace mezi vlákny. Základní fungování klíčové části implementace, tedy cyklické vyjímání struktur s informacemi z fronty jednotlivými vlákny, by se mohlo opírat o invariant testující neprázdnost fronty. Jinými slovy – dokud jsou ve frontě stále ještě nějaké položky, vlákna v cyklu budou tyto položky z fronty odebírat a zpracovávat (během tvorby uzlu také přidávat nové položky do fronty). Pokud bude fronta prázdná, cyklus skončí a vlákna (vyjma hlavního) zaniknou. Mohou však nastat situace, kdy fronta bude prázdná, ale cyklus se ukončit nesmí a vlákna nesmí zaniknout.

Jedna z těchto situací může nastat, když fronta bude obsahovat méně položek než bude běžících vláken. Máme-li ve frontě tři položky a čtyři běžící vlákna, tři z vláken si položky ve frontě rozdělí a čtvrté vlákno detekuje prázdnou frontu a bude se chtít ukončit. Jenže tři zbylá vlákna právě začala zpracovávat položky a tvořit uzly a mohla by tak do fronty další nové položky přidat. Čtvrté vlákno musí tedy nějak zjistit, zda existuje alespoň jedno další vlákno, které v danou chvíli pracuje (pravděpodobně může do fronty přidat položky). Pokud takové vlákno existuje, vlákno, které se nyní hodlalo ukončit, musí určitou dobu počkat a po uplynutí této doby se znovu dotázat na prázdnost fronty. Takto si všechna vlákna (pokud daná situace nastane) získávají povědomí o ostatních vláknech.

Pro tento účel jakési primitivní mezivláknové komunikace byla vytvořena struktura `ThreadLocalContext`, která má dvě členské proměnné – logickou hodnotu `isRunning` a ukazatel na instanci třídy `Thread` pro dané vlákno. Proměnnou `isRunning` si každé vlákno nastavuje v závislosti na tom, zda zrovna pracuje s frontou a vytváří uzel stromu nebo ne. Pokud vlákno vstupuje do bloku, kdy si vybírá položku z fronty, nastaví proměnnou `isRunning` na `true` a nenastaví na `false` dříve, než skončí jeho zpracování uzlu (vnitřního nebo koncového). Tento mechanismus je poměrně efektivní a jeho režie je docela nízká.

Nicméně stále může nastat situace, kdy fronta bude prázdná a všechna vlákna budou mít proměnnou nastavenou na `false` a přesto by algoritmus neměl skončit. Tato situace plyne z nedeterminismu chování vláken, resp. přidělování procesorového času jiným procesům. Situace nastane tak, že žádné z vláken aktuálně nebuduje uzly stromu, místo toho jsou například dvě z vláken několik málo instrukcí před nastavením `isRunning` na `true` a zbylá vlákna, těsně po tom, kdy si nastavila `isRunning` na `false`, se zrovna dotazují na živost ostatních vláken. Pokud by se zrovna dotazující vlákna, po zjištění, že žádné z vláken nepracuje, ukončila, zbytek výpočtu by se o ně zbytečně ochudil a probíhal by pouze na dvou zbylých vláknech, místo třeba čtyř.

Řešení této situace je poměrně snadné. Když vlákno zjistí, že ostatní vlákna mají proměnnou `isRunning` rovnu `false`, neukončí se ihned, ale chvíli počká (mezitím mohlo některé z vláken začít pracovat s frontou a má tedy `isRunning` rovno `true`) a poté se znovu dotáže na stav ostatních vláken. Pokud ani podruhé nedetekuje jiné běžící vlákno, pak je vysoká pravděpodobnost, že je budování stromu u konce a nyní se může ukončit. Časové kvantum čekání musí být dostatečně dlouhé na to, aby alespoň jedno z vláken přidalo do fronty položky anebo si nastavilo `isRunning` na `true`.

Sloučení všech přístupů k řešení komunikace mezi vlákny nastíní následující pseudokód:

```
while(true) do
      if not |Q| = 0 do
      threadQueue[currentThreadID].isRunning := true;
      lock Q;
            item := Q.front();
      unlock Q;
      buildNode(item);
      threadQueue[currentThreadID].isRunning := false; continue;
      else //|Q| = 0
            if ∃ threadQueue[i].isRunning = true do
                  currentThread.sleep(); continue;
            else //threadQueue[i].isRunning = false
                  if currentThread.attempt < 2
                        currentThread.sleep();
                        currentThread.attempt + 1; continue;
                  else exit loop;
                  endif;
            endif;
      endif;
endwhile;
```

Ve všech implementacích byly jako synchronizační primitivum použity spinlocky. Důvod je ten, že jednak implementace spinlocku už byla dodána v projektu [LAK09] a za druhé, z hlediska režie kritické sekce se jedná o nejjednodušší a nejrychlejší způsob, jak zajistit souběžnost vláken. Pro srovnání byly místo spinlocků použity monitory (taktéž implementace dodána v rámci stejného projektu), avšak ke znatelné změně výkonu, ať k lepšímu nebo horšímu, nedošlo. Navíc, implementace monitoru obsahuje také několik spinlocků, tedy lze očekávat, že režie činnosti chráněného bloku kritickou sekcí by mohla být lehce zvýšená. Naměřené časy ale nic takového neindikovaly, patrně byl nárůst režie tak nízký, že byl pod rozlišovací schopností časovače, který měřil dobu trvání stavby BVH stromu.

Velikosti bloků chráněných spinlockem jsou co nejmenší, aby zbytečně nedocházelo ke dlouhému blokování ostatních vláken a spotřebovávání procesorového času. V tomto momentu je dobré, že vlákna aktivně čekají (busy-waiting) a jakmile je kritická sekce volná, ihned do ní může vstoupit jiné vlákno a zbytečně se neztrácí výkon, který by byl nutný pro režii přepínání kontextů vláken při jejich uspávání a uvedení zpět do aktivního stavu.

Kapitola 5

Prezentace výsledků a statistiky

Navržené metody byly testovány na dvou odlišných vícejádrových systémech a na každém z nich se testovalo pět scén v rozsahu od cca 170 tisíc až dva miliony trojúhelníků. Testy probíhaly na obou strojích s operačním systémem Windows 7. První stroj obsahoval procesor Intel® Core™ 2 Duo T 7200, 2.00 GHz s dvěma jádry a s pamětí o kapacitě 1024 MB. Druhý stroj byl vybaven čtyřjádrovým procesorem Intel® Core™ 2 Quad Q9550, 2.83 GHz s kapacitou paměti 4096 MB. Jako referenční scény byly vybrány Happy Buddha, Fairy Forest, Icosahedron, Conference a Soda Hall. Scény a jejich parametry jsou uvedeny na obrázku 15 a v tabulce č. 2.

Obrázek 15: (zleva) Happy Buddha (Stanford University), Fairy Forest (University of Utah), Icosahedron, Conference – (Anat Grynberg & Greg Ward), Soda Hall (University of Berkley)

scéna	***Fairy Forest***	***Conference***	***Happy Buddha***	***Icosahedron***	***Soda Hall***
počet vrcholů	100 732	166 940	543 652	908 988	4 189 233
počet trojúhelníků	174 117	282 759	1 087 716	1 800 000	2 169 132
počet uzlů	100 771	146 505	713 287	920 765	1 168 687

Tabulka 2: řádky udávají počet vrcholů, trojúhelníků a uzlů výsledného BVH stromu pro každou z testovaných scén

Implementace všech vícevláknových metod je napsána v jazyce C++ a zkompilována ve vývojovém prostředí MS Visual Studio 2010 Ultimate. Cílovou platformou byla architektura x86 a pro kompilaci byly zapnuty volby Favor fast code (/Ot), Whole pogram optimization (/GL) a povolena volba použití SSE2

instrukcí (/arch:SSE2). Naměřené výsledky prezentují časy jednovláknové implementace stavby BVH stromu, vícevláknové implementace s nesymetrickým přidělováním jader, vícevláknové implementace se symetrickým přidělováním jader a vícevláknové implementace s využitím fronty. Dále je prezentován procentuální nárůst vícevláknových metod v porovnání s jednovláknovou implementací. Tabulka 3 a 4 prezentuje naměřené časy metod na dvou testovacích sestavách.

Intel® Core™ 2 Duo T 7200, 2.00 GHz (2 vlákna), 1024 MB RAM							
Model	**jednovláknová**	**symetrická**		**nesymetrická**		**fronta**	
		čas	**nárůst**	**čas**	**nárůst**	**čas**	**nárůst**
Happy Buddha	41,87 s	22,85 s	**45,42 %**	22,81 s	**45,52 %**	23,31 s	**44,32 %**
Fairy Forest	6,04 s	5,74 s	**4,96 %**	5,77 s	**4,47 %**	3,6 s	**39,66 %**
Icosahedron	67,64 s	40,68 s	**39,85 %**	39,87 s	**40,96 %**	40,07 s	**40,67 %**
Conference	10,35 s	6,37 s	**38,69 %**	6,32 s	**38,93 %**	5,93 s	**42,66 %**
Soda Hall	101,09 s	94,94 s	**8,56 %**	93,38 s	**7,62 %**	92,46 s	**8,53 %**

Tabulka 3 – naměřené výsledky všech metod na dvoujádrovém systému

Intel® Core™ 2 Quad Q 9550, 2.83 GHz (4 vlákna), 4096 MB RAM							
Model	**jednovláknová**	**symetrická**		**nesymetrická**		**fronta**	
		čas	**nárůst**	**čas**	**nárůst**	**čas**	**nárůst**
Happy Buddha	27,37 s	10,37 s	**62,11 %**	15,15 s	**44,64 %**	11,67 s	**57,36 %**
Fairy Forest	3,98 s	2,69 s	**32,41 %**	2, 68 s	**32,66 %**	2,06 s	**48,24 %**
Icosahedron	44,5 s	17,91 s	**59,75 %**	22,52 s	**49,39 %**	18,13 s	**59,25 %**
Conference	6,79 s	3,72 s	**45,21 %**	3,61 s	**46,83 %**	3,05 s	**55,08 %**
Soda Hall	66,51 s	45,89 s	**31,0 %**	45,8 s	**31,13 %**	38,1 s	**42,71 %**

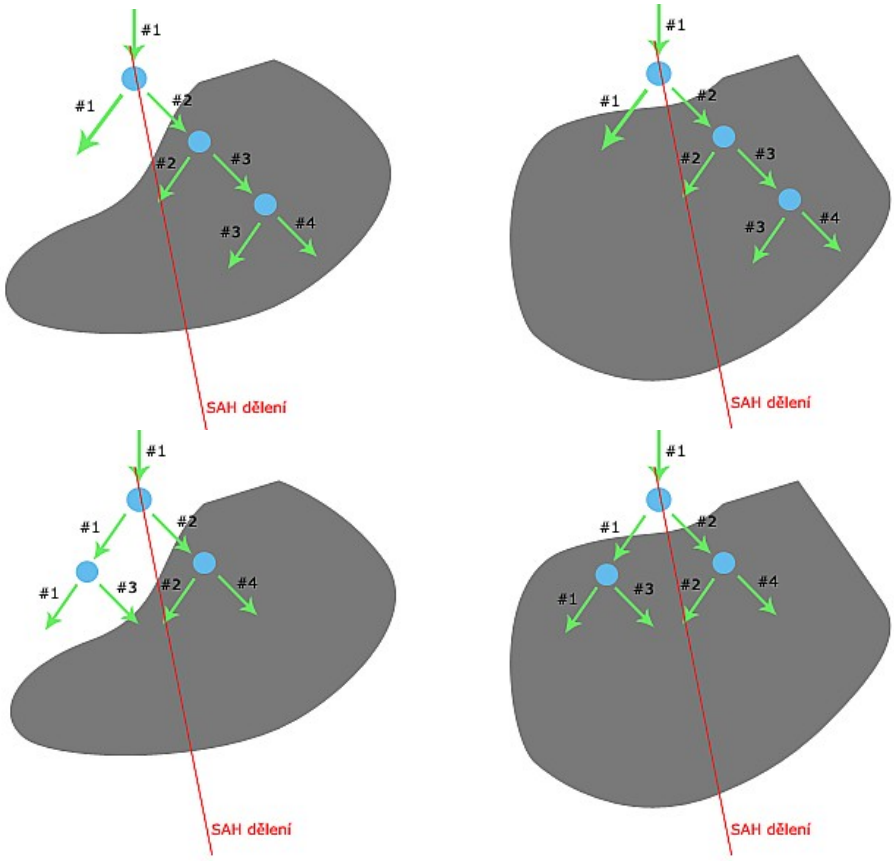
Tabulka 4 – naměřené výsledky všech metod na čtyřjádrovém systému

5.1 Porovnání koncepcí metod, výhody a nevýhody

Během testování všech metod se zjistilo, že je jejich výkon výrazně ovlivněn tvarem scény, resp. rozprostřením vrcholů modelu v prostoru. Z naměřených hodnot lze vyvodit závěr, že čím více je scéna symetrická, ať už podle počátku souřadného systému nebo podle lokálních souřadnic scény, lze pozorovat tendenci většího výkonu metody s rovnoměrným vytvářením vláken. V tomto případě výkon metody s nerovnoměrným vytvářením vláken se zdá být výrazněji horší (například u modelu icosahedronu). V případě obecných scén metoda s nerovnoměrným vytvářením vláken dosahuje spíše srovnatelných hodnot nebo minimálně horších, v řádu desetin, případně několik jednotek procent než metoda s rovnoměrným vytvářením vláken.

Když algoritmus SAH dělí symetrickou scénu, každá polorovina po rozdělení obsahuje do značné míry podobný nebo stejný objem geometrie scény. Tím je zajištěno, že každé, třeba ještě nevytvořené vlákno, obdrží rovnoměrný objem dat. Zde je jasně vidět, že výhodu má metoda s rovnoměrným přidělováním vláken a naopak v nevýhodě je nyní metoda, která vlákna přiděluje nerovnoměrně. V jejím případě musí jedno vlákno zpracovat celý obrovský podstrom dat samotné, zatímco na druhý podstrom dat je vláken vyhrazeno více. Na dvoujádrovém systému tento rozdíl není tak evidentní, protože kvůli absenci více jader každý podstrom geometrie bude zpracovávat jedno vlákno a tedy budou první dvě metody takřka ekvivalentní, rozdíl se více projeví ve čtyř a vícejádrovém prostředí.

V případě nesymetrické scény se stává předchozí nevýhoda metody s nerovnoměrným vytvářením vláken výhodou. Při prvním rozdělení scény algoritmem SAH se přidělí více vláken té oblasti, kde je geometrie mnohem víc a kde je též zapotřebí více výpočetní kapacity. Naopak nyní metoda s rovnoměrným přidělováním vláken umožní oblasti s nízkým objemem dat pracovat s více vlákny. To může zapříčinit situaci, kdy je oblast s méně daty zpracována velmi rychle, ale vlákna ukončí svou existenci dříve než by měla a oblast, kde je dat mnohem více, zbytečně trpí nedostatečnou výpočetní silou. Znázornění všech případů rozdělení je na obrázku č. 16.



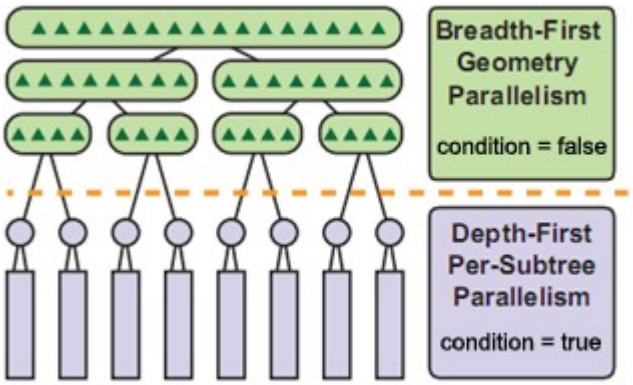
Obrázek 16:
(levý horní) generování vláken metodou s nerovnoměrným přidělováním na málo symetrické scéně. Pro oblast, kde je geometrie scény o poznání méně plně stačí jedno vlákno. V oblasti, kde je geometrie mnohem více, je taktéž vláken víc a vlákna jsou tedy lépe využita.

(pravý horní) generování vláken metodou s nerovnoměrným přidělováním na symetrické scéně. Oblast s jedním vláknem trpí nedostatečnou výpočetní kapacitou, zatímco oblast s více vlákny je zpracována dříve a vlákna jsou příliš brzy ukončena.

(levý dolní) generování vláken metodou s rovnoměrným přidělováním na málo symetrické scéně. Oblast s velkým objemem geometrie nemá dostatek vláken, zatímco oblast s méně daty má vláken zbytečně moc.

(pravý dolní) generování vláken metodou s rovnoměrným přidělováním na symetrické scéně. Vlákna jsou rovnoměrně distribuována do obou podobně velkých oblastí a žádná není znevýhodněna nedostatkem výpočetní síly.

Třetí metoda, metoda s použitím fronty, slučuje výhody obou předchozích metod. Její koncept lze popsat jako rovnoměrnou distribuci vláken na vyžádání (on-demand). V počáteční fázi se uzly stromu zpracovávají do šířky, po splnění podmínky pro rekurzivní stavbu se strom nebo jeho části budují do hloubky, případně se oba koncepty kombinují (obrázek 17). Tím je dosaženo maximální efektivity a výpočetní kapacita je vždy k dipozici tam, kde je nejvíce potřeba. Tento přístup navíc snižuje náchylnost vůči nesymetrii scény, pro kterou se BVH strom staví, nicméně neeliminuje ji zcela.



*Obrázek 17: Pokud je splněna podmínka, části BVH stromu se začnou budovat rekurzívně. Přejato z [CKL*10].*

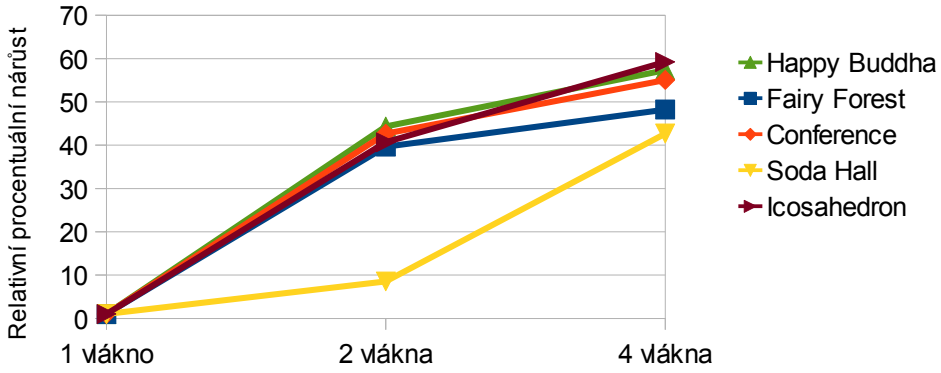
Metoda s použitím fronty v naměřených výsledcích převážně dominuje. Nejvíce je tato dominance patrná na čtyřjádrovém a vícejádrovém systému. Na dvoujádrovém systému je metoda srovnatelná s oběma předešlými. Tato dominance bude postupně s narůstajícím počtem jader stále více převažovat, ale jen do určité míry. V jednom okamžiku se jí metody bez fronty budou výkonově opět přibližovat, protože režie spojená s managementem fronty bude převyšovat rychlost stavby BVH stromu. Navíc její současná implementace obsahuje více kritických sekcí, než mají implementace nefrontových metod a tedy více jader nebude zaručovat razantní nárůst výkonu, poněvadž vlákna budou muset čekat na vykonání kritické sekce jiným vláknem. V tento moment již nebude hrát roli, zda čekajících vláken bude deset nebo čtyřicet, výkon se již nebude moci zvýšit.

Mohlo by se očekávat, že zdvojnásobením počtu vláken se automaticky dvojnásobně zkrátí čas potřebný ke stavbě BVH stromu, ale není tomu tak. Je potřeba si uvědomit, že režie spojená s vlákny si vyžádá určité kvantum výpočetního času a především čekání vláken na přístup do kritických sekcí je nejvíce zpoma-

lovacím prvkem algoritmu. Bohužel, bez kritických sekcí se algoritmus neobejde. Dále, samotná stavba BVH stromu je pouze část celého algoritmu, tedy další související části algoritmu jako např. inicializace dalších pomocných datových struktur a jejich naplnění daty probíhá stále sériově.

Amdahlův zákon [Amd67] říká, že maximální teoretický nárůst výpočetního výkonu, jaký může být získán s N vlákny pro algoritmus (program), jehož část P může být paralelně zpracovávána, je dán vztahem $1/((1-P)+(P/N))$. Ze vztahu vyplývá, že s rostoucím počtem vláken se bude čas na provedení úlohy zkracovat. Jakmile se ale bude N neustále zvyšovat, dojde k situaci, kdy se přidávání dalších vláken na výpočetním čase nijak neprojeví. Vztah se zredukuje na $1/1-P$. Tedy, za předpokladu, že například 15 % doby stavby stromu se provádí sériově (do této doby není zahrnut čas potřebný k načtení scény do paměti, pouze čas uplynulý od inicializace potřebných datových struktur s spuštění samotné stavby BVH stromu) a zbylých 85 % se vykoná paralelně, podle vzorce se dosáhne maximálně 6,6násobného zrychlení.

V úvahu je také potřeba vzít i omezení rychlosti vzniklé použitým hardwarem, především rychlost pamětí a frekvence sběrnice mezi pamětí a procesorem a jejich vzdáleností. V průběhu stavby musí procesor prakticky neustále přistupovat do operační paměti, protože buď načítá geometrii scény pro zpracování anebo do paměti ukládá mezivýsledky, což jsou v podstatě fragmenty budovaného BVH stromu. Výsledky tedy značně reflektují výkon použité architektury a jsou ovlivněny propustností pamětí a sběrnice, ne jenom výpočetním výkonem procesorů. Graf 1 zobrazuje relativní nárůst výkonu frontové metody u každé scény vzhledem k počtu jader.



Graf 1: Relativní nárůst výkonu frontové metody pro jednotlivé scény na jednom, dvou a čtyř jádrech. Je vidět, že v případě scény Soda Hall je nárůst výkonu na čtyřjádrovém systému oproti dvoujádrovému nejvyšší ze všech scén.

Kapitola 6

Závěr

Tato práce představila problematiku paralelní stavby hierarchie obalových těles a popsala tři možné implementace stavby. Naměřené výsledky vykazují snížení doby stavby BVH stromu u některých scén více jak o polovinu, v některých případech se doba snížila téměř na třetinu původní doby jednovláknového zpracování. Mimo jiné byl popsán postup, jak v daném systému do úlohy zapojit všechny dostupné procesory a jádra a určit tak optimální počet pracovních vláken vzhledem k systému, aby nedošlo ke zbytečnému zatížení a snížení výkonu z důvodu vysoké režie spojené s obsluhou vláken.

Byly navrženy a implementovány tři metody – metoda s rovnoměrným přidělováním vláken, kdy každý podstrom hierarchie obdrží stejný počet pracovních vláken, metoda s nerovnoměrným přidělováním vláken, kdy podstrom hierarchie s převažujícím množstvím dat obdrží téměř všechna vlákna (alespoň jedno vlákno je vyhrazeno pro druhý podstrom s méně daty) a metoda s využitím fronty uzlů, kdy si vlákna uzly z fronty postupně vybírají a zpracovávají je.

Na pěti testovaných scénách (viz tabulky 3 a 4) více převažovala metoda s použitím fronty, zatímco obě nefrontové metody vykazovaly zhruba srovnatelný výkon, až na jeden dva výjimečné případy (např. model Happy Buddha na čtyřjádrovém systému), kde metoda s rovnoměrným přidělováním vláken předčila i metodu, která používá frontu. Nárůst rychlosti stavby BVH stromu se u všech metod blížil dvojnásobku rychlosti jednovláknové implementace, v mnoha případech byl dvojnásobek překročen a dokonce na čtyřjádrovém systému se výkon blížil ke trojnásobku jednovláknové implementace.

Existuje mnoho dalších návrhů, které implementují úlohu stavby hierarchie obálek a stejně tak by bylo možné zmíněné tři metody dále zdokonalit. Jako jedna z možných cest je vektorizace základních algebraických operací za použití SIMD instrukcí, konkrétně technologie SSE. Tímto by se mohly základní operace s vektory a souřadnicemi trojúhelníků scény značně urychlit. Další cesta, jak urychlit výpočet, by mohla vést k přesunutí provedení této úlohy na grafických čipech za pomoci technologie CUDA od společnosti NVIDIA [CUDA].

Reference

[Amd67] AMDAHL, G., M. *Validity of the Single Processor Aproach to Achieving Large Scale Computing Capabilities*. In Proc. Spring Joint Computer Conf. New York (1967) pp 483 –485.

[App68] APPEL, A. *Some Techniques for Shading Machine Renderings of Solids*. New York, 1968. pp 37-45. IBM Research Centre.

[SSM*05] SLUSALLEK, P., SHIRLEY, P., MARK, B., STOLL, G., WALD, I. *Introduction To Realtime Ray Tracing.* 56 stran. SIGGRAPH 2005.

[Msdn] Inline Assembler Overview. MICROSOFT. *MSDN – the Microsoft Developer Network* [online]. 2010, 9.12.2010 [cit. 2013-03-21]. Dostupné z: http://msdn.microsoft.com/en-us/library/5f7adz6y%28v=vs.100%29.aspx

[BCK*09] de BERG, M., CHEONG, O., van KREVELD, M., OVERMARS, M. *Computational Geometry : Algorithms and Applications* 3. rev. ed. Springer-Verlag, 2008. 386 stran.

[Itbb] Intel® Threading Building Blocks. *Intel® Threading Building Blocks* [online]. 2013 [cit. 2013-02-14]. Dostupné z: http://threadingbuildingblocks.org

[CKL*10] CHOI, B., KOMURAVELLI, R., LU, V., SUNG, H., Robert L. BOCCHINO, R. L., V. ADVE, S. V., HART, J. C. *Parallel SAH k-D Tree Construction.* University of Illinois at Urbana – Champaign. High Performance Graphics 2010, 10 stran.

[SSK01] SHEVTSOV, M., SOUPIKOV, A., KAPUSTIN, A. *Highly Parallel Fast KD-tree Construction for Interactive Ray Tracing of Dynamic Scenes.* Intel Corporation, Volume 26, Number 3. Eurographics 2007.

[Intel12] Intel Corporation. *Intel® Processor Identification and the CPUID Instruction.* Application Note 485, 128 p., May 2012.

[LAK09] LAINE, S., AILA, T., KARRAS, T. *Understanding the Efficiency of Ray Traversal on GPUs.* NVIDIA Research. 5 stran, Proc. High-Performance Graphics 2009

[CKM03] CLINGMAN, D., KENDALL, S., MESDAGHI, S. *Practical Java Game Programming.* Charles River Media Game Development. 171 stran. 2003

[CUDA] CUDA Parallel Computing Platform. *NVIDIA.com* [online]. 2013 [cit. 2013-05-10]. Dostupné z: http://www.nvidia.com/object/cuda_home_new.html

[Soch10] SOCHOR, J. *Detekce kolize. Detekce kolize: Skripta k předmětu PA010: Počítačová grafika.* Brno, 2010, 109 s.

[Pat13] World Patents. PATTERA INC. w.*pat.tc* [online]. 2013 [cit. 2013-05-10]. Dostupné z: http://w.pat.tc/WO2009032757A2fp.png

[Tak09] TAKAHASHI, Dean. Caustic Graphics to create graphics chips with novel ray-tracing technology. In: *VentrureBeat.com* [online]. 2009 [cit. 2013-05-20]. Dostupné z: http://venturebeat.com/2009/03/08/caustic-graphics-to-create-graphics-chips-with-novel-ray-tracing-technology/

[Pxleyes] 5 Things You Need to Know About Raytracing. In: *Pxleyes.com* [online]. 2013 [cit. 2013-04-27]. Dostupné z: http://www.pxleyes.com/blog/2010/03/5-things-you-need-to-know-about-raytracing/

[Thread] CreateThread function (Windows). MICROSOFT. *Windows Desktop Development – Windows Dev Center* [online]. 2012, 21.11.2012 [cit. 2013-04-02]. Dostupné z: http://msdn.microsoft.com/en-us/library/windows/desktop/ms682453%28v=vs.85%29.aspx

Příloha A: Soubory a zdrojové kódy na přiloženém CD

Obsah nosiče obsahuje:

- adresář rt – adresářová struktura projektu se všemi zdrojovými soubory, testovacími scénami, testovacími skripty a spustitelným projektem MS Visual Studio 2010
- soubor rt.exe – spustitelný program projektu