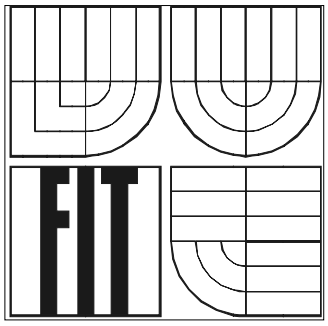
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



Knihovna pro generování a zobrazování scény uložené v BSP stromu

Bakalářská práce

2006 **Mário Roženský**

Knihovna pro generování a zobrazování scény uložené v BSP stromu

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Jana Pečivy
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Mário Roženský
20.4.2006

Abstrakt

Knihovna sloužící pro vytvoření a zobrazení scény formou BSP stromu. Naprogramovaná pomocí jazyka C++ a knihovny Open Inventor. Umožňuje několik voleb tvorby a zobrazení výsledného stromu. Strom vytváří ze zadaného uzlu v grafu scény. Práce dále popisuje možné využití této knihovny a BSP stromů obecně. Také jejich vhodnost/nevhodnost na různé typy scény.

Klíčová slova

BSP strom, algoritmus, Open Inventor, Binary space partitioning, knihovna, dělení prostoru, C++, PVS, portály

Poděkování

Jan Pečiva – významná pomoc s Open Inventorem a vedení projektu

Abstract

Main purpose for this library is to create and display scene converted into BSP tree. Programming language is C++ and additional library is Open Inventor. There are several possibilities allowing user to set parameters how to create and display the tree. The tree is created from scene graph node. This work also describes how can be BSP algorithm used in some cases. It also discusses different kinds of scenes and how is algorithm useful.

Keywords

BSP tree, algorithm, Open Inventor, Binary space partitioning, library, space partitioning, C++, PVS, portals

Obsah

1 Úvod

V současných aplikacích se stále zvyšuje komplexnost zobrazovaných scén, množství dat popisujících geometrii objektů a objevuje se problém jak vybrat pouze tu část geometrie, kterou pozorovatel vidí. Dnešní scéna má okolo 5 – 10 milionů polygonů, ale schopnost grafické karty je pouze v řádech statisíců. A to jsou většinou počty, které zvládne bez osvětlení a speciálních efektů. Proto byly vyvinuty algoritmy, které právě viditelnost řeší, je jich mnoho, každý je určený na něco jiného, některé z nich jsou vhodné na uzavřené prostory, některé naopak na otevřené prostranství. Některé jsou předpočítané, jiné jsou realtime počítány za běhu. Existuje mnoho algoritmů, které vykreslování urychlují na základě volby pouze toho co je vidět. Základem mnoha z nich je BSP strom, jenž sám o sobě nic neurychlí, ale pomůže při rozdělení scény na menší části.

BSP je zkratka pro Binary Space Partitioning. A slovo strom se přidává proto, protože při práci algoritmu vzniká jakýsi strom rozdělení scény na části. BSP pracuje převážně se statickými daty, ale dynamické rozšíření je také možné. Předpočítanost scény má velké výhody, protože nezatěžuje výpočty při startu aplikace. Jeho použití je určeno primárně na scény uzavřené, které se dají snadno rozdělovat. Příkladem uzavřené scény je například byt, nebo nějaká budova. Využití algoritmu najdeme při vizualizacích návrhů bytů, v CAD aplikacích, ale také při ray-tracingu nebo v počítačových hrách.

Algoritmus je výchozím bodem pro mnoho rozšíření a jiných algoritmů, které na něj navazují a rozšiřují ho. Takovými algoritmy jsou PVS a portály, dále algoritmy pro detekci kolizí, ray-tracing a mnohé další.

Poprvé jsem aplikaci tohoto algoritmu viděl ve hře DooM. Bylo to před 13 lety, byla to revoluce, dosud žádná hra nedokázala v reálném čase zobrazit trojrozměrné prostory. Do té doby byla prostorem pohybu pouze rovina a zdi byly vykreslovány tak, aby to budilo dojem trojrozměrného prostoru. Následovaly další hry, jmenujme například Quake, Unreal, nebo Half-Life a mnoho jiných, většinou založených na enginech z těchto her.

Tyto hry inspirovaly i tvůrce free enginů. Mimo jiné i mě, proto jsem se chtěl s touto problematikou blíže seznámit. A na základě mých předešlých zkušeností a zájmu vznikla tato práce. Již v minulosti jsem zkoušel algoritmy pro urychlení renderování jako například oct-tree, který je svým principem dělení prostoru podobný BSP stromům.

Práce v několika kapitolách popisuje problematiku ohledně BSP stromů. V druhé kapitole je popsána knihovna Open Inventor a způsob provázanosti mého programu s touto knihovnou.

Ve třetí kapitole je obecné seznámení s BSP stromy, jak se tvoří, k čemu slouží.

Čtvrtá kapitola se zaměřuje na použití dat rozdělených do stromu. Zaměřuje se zejména na praktické využití v počítačové grafice. Způsob vykreslování, detekce kolizí apod.

Pátá kapitola je praktickou ukázkou generování stromu a práce BSP algoritmu. V této kapitole jsou popsány postupy například jak strom vykreslit, nebo způsob dělení polygonu, pokud je potřeba.

V šesté kapitole popíšeme testování rychlosti výpočtu několika typů algoritmu. Také je zde vypočítána časová a prostorová složitost algoritmu. Jsou navrženy způsoby, jak strom optimalizovat.

Sedmá kapitola se zabývá nadstavbou nad BSP v podobě PVS a portálů a tím i urychlením vykreslování scény.

Osmá kapitola je zhodnocením práce.

Nakonec jsem přidal popis parametrů pro nastavení algoritmu a sadu obrázků vytvořených během práce s programem.

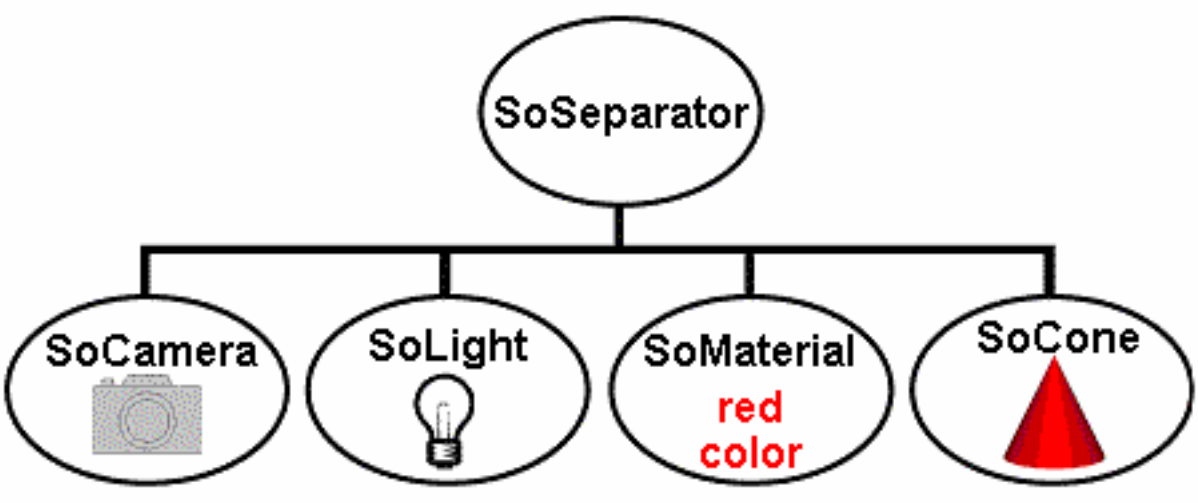
2 Open Inventor

Open inventor je objektově orientovaný soubor tříd pro práci s objekty v 3D prostoru. Je postaven nad grafickou knihovnou OpenGL. Inventor poskytuje mnohem příjemnější práci než OGL díky zapouzdření jednoduchých funkcí do tříd pracujících na vyšší úrovní. Programátor se nemusí zabývat způsobem fungování na jednotlivých grafických kartách, to za něj řeší knihovna. Avšak toto urychlení práce má i negativní stránku a ta se projeví pokud programátor chce implementovat něco, kdy potřebuje přistupovat k základním geometrickým datům, jako jsou trojúhelníky ve scéně, či ovlivnit způsoby vykreslování.

Vývoj knihovny začal v roce 1991 firmou Silicon Graphics, jako nástavba nad OpenGL. V současnosti existují tři projekty založené na OI. My budeme používat knihovnu Coin3D [1].

Podrobnější popis, instalaci a ukázkové aplikace lze najít na serveru www.root.cz [2]. Dalším dobrým materiálem, ve kterém jsou výborně zdokumentovány třídy a popsány řešení rozličných problémů je The Inventor Mentor [3].

Scéna je v Open inventoru uložena v takzvaném grafu scény. Je to graf, ve kterém můžeme každému uzlu přiřadit synovské uzly, které přebírají vlastnosti rodičovského a dále je upravují či nastavují. Ukázka převzata z [2].

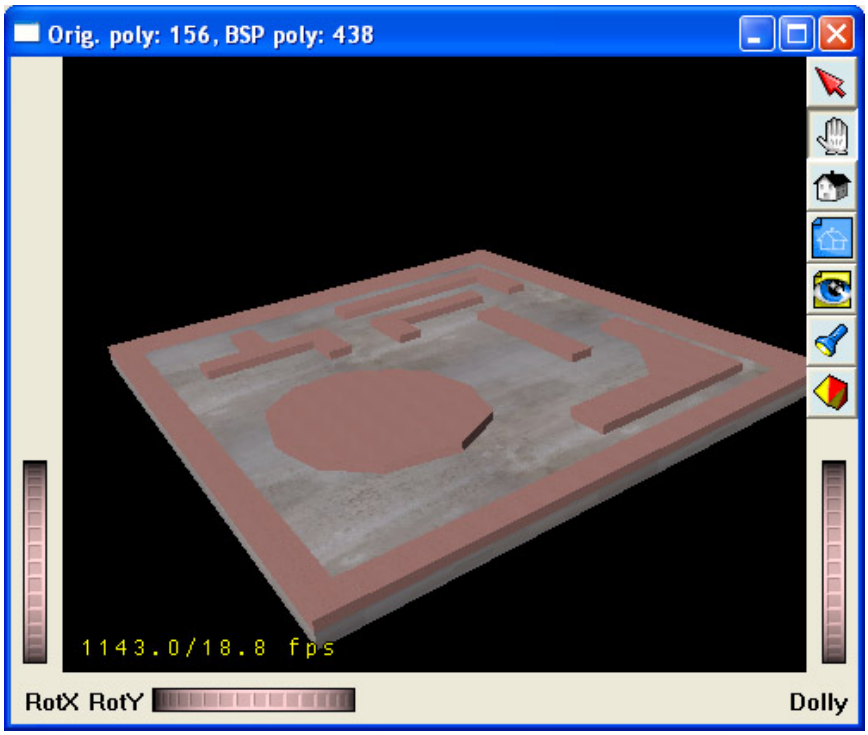


Obr 2.1: Graf scény v Open Inventoru

2.1 Knihovna pro BSP a Open Inventor

Knihovna není zaměřena na žádnou speciální vlastnost a použití BSP stromu, ale je navržena tak, aby byla co nejvíce obecná a tudíž tu byla možnost snadného rozšíření do budoucna. Možností k rozšíření je mnoho a některé z nich budou popsány v dalších kapitolách.

Knihovna funguje následujícím způsobem. Třídě je při vytvoření předán uzel grafu scény, který se má zpracovat do BSP stromu, tento je pak polygon po polygonu přebrán třídou a jsou načtena všechna potřebná data k vytvoření stromu. Poté co jsou trojúhelníky zpracovány, je možné původní uzel z grafu scény vyjmout a použít metodu třídy, která vytvořený strom přidá do scény namísto původního uzlu. Nyní při vykreslování uzlu se o vykreslování stará sama třída. Zvolil jsem způsob vykreslování, tak jak byl původně navržen a to odzadu dopředu. Neboli zjistí se nejprve pozice kamery, poté se pro každý uzel zjišťuje, zda je kamera před, nebo za dělící rovinou a pokud je před, tak se nejprve vykreslí zádní větev, potom objekty, které leží v rovině dělícího polygonu a nakonec přední větev. Pokud leží pozorovatel za rovinou, tak je postup opačný.



Obr 2.1: Scéna v Open inventoru

3 BSP strom

Tato kapitola shrnuje obecné informace o BSP stromech, jak se vytvářejí, k čemu se dají použít, případně i historii.

3.1 Co je BSP strom?

Binary Space Paritioning (BSP) tree, neboli česky „strom, který binárně dělí prostor“ reprezentuje rekurzivně a hierarchicky rozdělený n-rozměrný prostor na konvexní podprostory. Vytváření takového stromu je proces, při kterém se vezme prostor, určí se dělící „rovina“ a touto se daný prostor rozdělí na dva nové podprostory.

Obecně je „rovinou“ v n-rozměrném prostoru myšlen n-1 rozměrný objekt, kterým může být daný prostor rozdělen na dva další. Například pro trojrozměrný prostor je dělícím objektem rovina, pro dvourozměrný prostor je dělícím prvkem přímka.

Protože je tato datová struktura strom, tak má každý uzel své potomky. Protože jde o binární strom, tak má potomky dva a to přední a zadní uzel.

Obr. 3.1: Dělení objektu rovinou

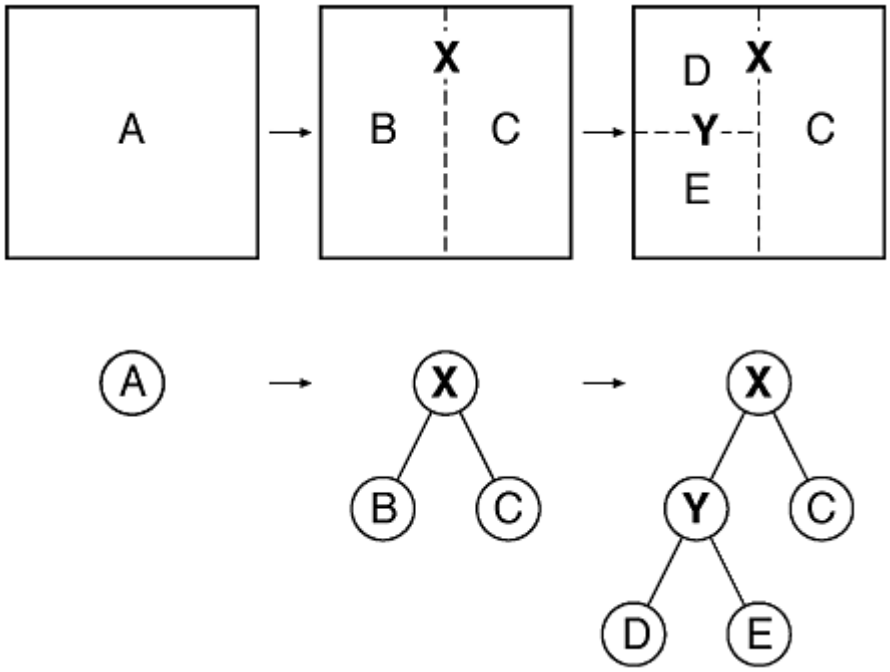
3.2 Použití BSP stromu

Rozdělení prostoru na podprostory je velmi výhodné a má široké možnosti využití zejména pro urychlení výpočtu ve složitých scénách díky určení a vymezení prostoru, ve kterém je potřeba počítat. Takovými aplikacemi jsou například algoritmy pro určení viditelných částí objektů, algoritmy na výpočet fyziky, ray-tracing, navigace v prostoru a mnoho dalších. Popisem možného způsobu urychlení se budeme zabývat především v sedmé kapitole.

3.3 Příklad vytváření stromu

Pro názornost a snazší pochopení, si omezíme prostor pouze na dva rozměry, takže budeme pracovat v rovině a jako dělící „rovinu“ použijeme přímku.

Demonstraci si můžeme ukázat například u dělení čtverce. Na začátku výpočtu máme celý čtverec, který chceme rozdělit na menší části. Vybereme první dělící přímku a rozdělíme čtverec v polovině podle osy Y na dvě části. Tímto jsme získali kořen stromu a dvě větve, každá určuje co je na jedné nebo druhé straně přímky. Nyní vybereme další dělící přímku, tentokrát v ose X a dělíme znovu. Nyní jsme již vybírali pouze z nově vzniklé části, ne z celého objektu. Takto můžeme pokračovat dále, dokud není výsledný čtverec/obdélník určité velikosti, nebo pokud nedosáhneme určitého počtu dělení. Následující obrázek ukazuje popisovaný příklad.



Obr. 3.2: Demonstrace generování stromu

4 Využití BSP stromu

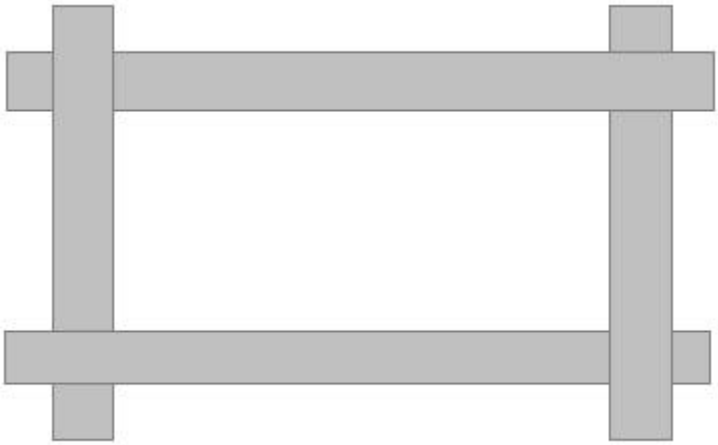
4.1 Odstranění neviditelných polygonů

Pravděpodobně nejčastější aplikací BSP stromů je odstranění neviditelných polygonů v trojrozměrných scénách. Je to elegantní a efektivní způsob, jak setřídit polygony tak, aby se mohl strom procházet metodou „depth-first.“ Použití můžeme ukázat při vykreslování odzadu dopředu pomocí „malířova algoritmu“, nebo zepředu dozadu na „scanline algoritmu.“

Při vybírání dělícího polygonu většinou používáme rovinu jednoho z polygonů, pokud není možné takový polygon nalézt, potom analyticky vypočteme dělící rovinu.

4.1.1 „Malířův algoritmus“

Algoritmus pracuje tak, že vykresluje polygony od nejvzdálenějšího po nejbližší. Skryté plochy budou překresleny polygony, které jsou před nimi. Avšak problém nastane, když se dva polygony vzájemně překrývají (viz. Obr 4.1). V tom případě dojde k problému, který polygon vykreslit dříve? Řešením je rozdělení polygonu.



Obr. 4.1: Překrývající se polygony

Důvod, proč je BSP algoritmus v tomto případně výhodný je ten, že dělení polygonů je automatickou součástí při vytváření stromu. V situaci, která je uvedena na obrázku, stačí rozdělit pouze jeden polygon a poté bude vykreslování prováděno tak, že se začne u vzdálené větve stromu a až se vykreslí, tak se bude vykreslovat větev blízká. A samozřejmě rekurzivně pro podstromy.

4.1.2 „Scanline algoritmus“

Procházení stromem ve směru od nejbližšího uzlu po nejvzdálenější je stejné, jako procházení opačné. Můžeme toho využít při scanline algoritmu, který používá masku, aby pixely, které jsou zakryté, nebyly zbytečně vykreslovány. Toto je velmi podstatné, pokud jsou počítány složité per-pixel výpočty, tak se provedou pouze na těch nejbližších polygonech, ale ne na těch, které zakrývají. Tuto výhodu „malířův algoritmus“ nemá a tudíž dochází ke zbytečnému překreslování bodů.

Algoritmus funguje tak, že si uchovává scanlines, neboli řádky pixelů, tak jak budou vykresleny na obrazovku. Nyní pokud máme polygon, který je blíže, než nějaký nově vykreslovaný, tak se stává polygonem maskovacím a tudíž pixely, které jsou za ním schovány se nevykreslí. Jedna z nejdůležitějších věcí, pro správné fungování algoritmu je, mít uloženy maskovací pixely a pro každý polygon porovnat, zda tyto pixely již nebyly kresleny. Pro uspořádání těchto dat je výhodné je reprezentovat jako BSP stromy pro snadnější vyhledání.

4.2 Zjišťování viditelnosti

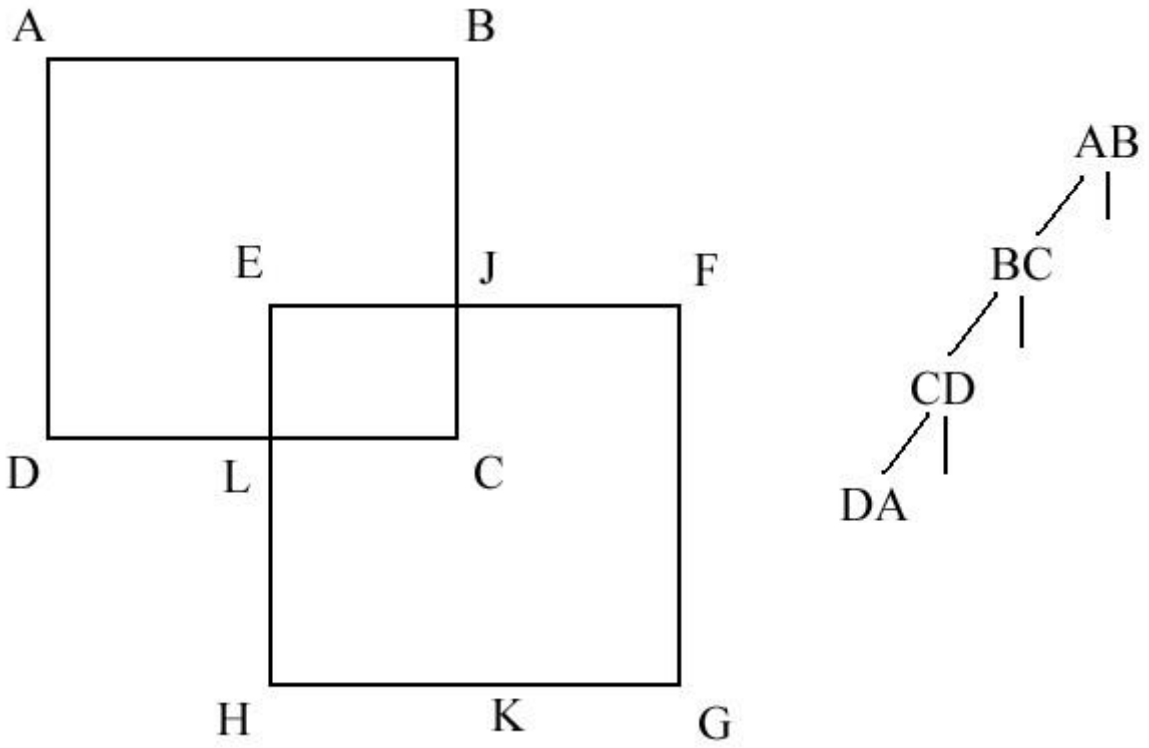
Analytické zjišťování viditelnosti je způsob, jak zjistit, zda jsou nějaké polygony vidět z určitého bodu ve scéně. Je to velmi důležité, protože dnešní scény mají miliony polygonů, ale současně jsou vidět pouze tisíce. Stejně tak, když se například počítá osvětlení povrchu, tak není potřeba počítat osvětlení na všech bodech, ale pouze na těch, které jsou z pozice světla viditelné. Více informací je v sedmé kapitole.

4.3 Bool operace

Test na před/za rovinou si můžeme také představit jako test uvnitř/vně objektu, díky tomu můžeme provést booleovské operace s objekty. Booleovské operace průnik, sloučení, součet a rozdíl se provádí podobným způsobem, jen je rozdíl kam do stromu se přiřadí nová větev.

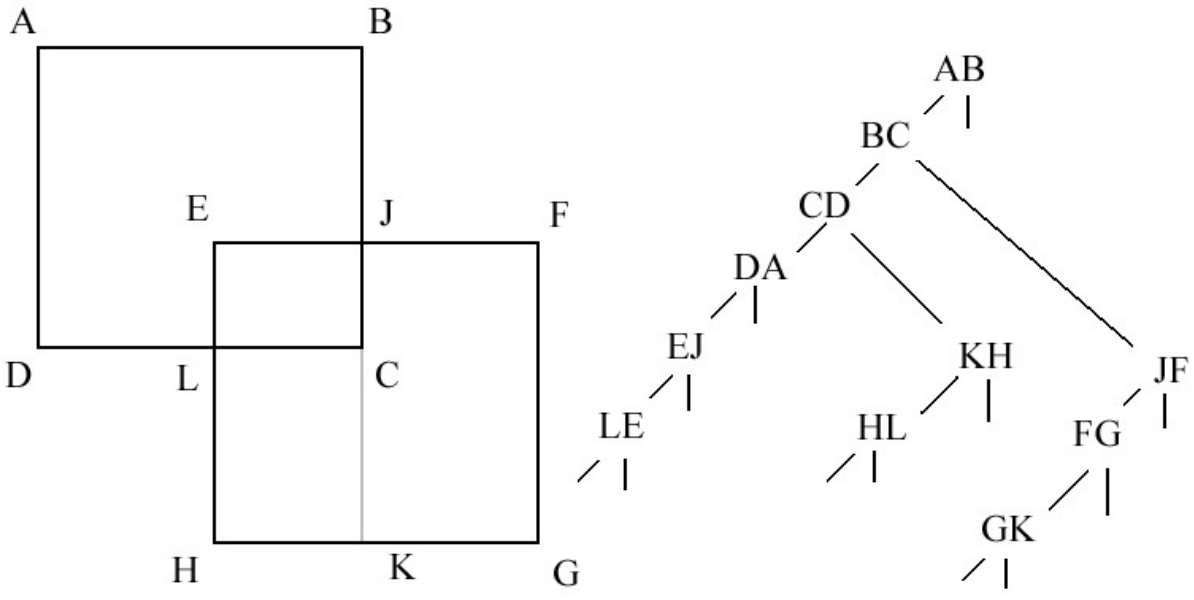
Možnost tohoto využití BSP stromu je například v CSG (Constructive Solid Geometry) grafice, kde se objekty konstruují z jednodušších objektů typu koule, kvádr, válec právě pomocí přičítání, odčítání, průniků, sjednocení.

Ukážeme si operaci sloučení. Máme dva čtverce ABCD a EFGH. budeme konstruovat strom nejprve čtverce ABCD. Pokud je bod uvnitř všech úseček ohraničujících čtverec, považujeme bod za vnitřní, jinak je bodem vnějším. Jako dělící přímky vybíráme přímky rovnoběžné s hranami. Prázdný uzel ve stromu ukazuje směrem mimo objekt.



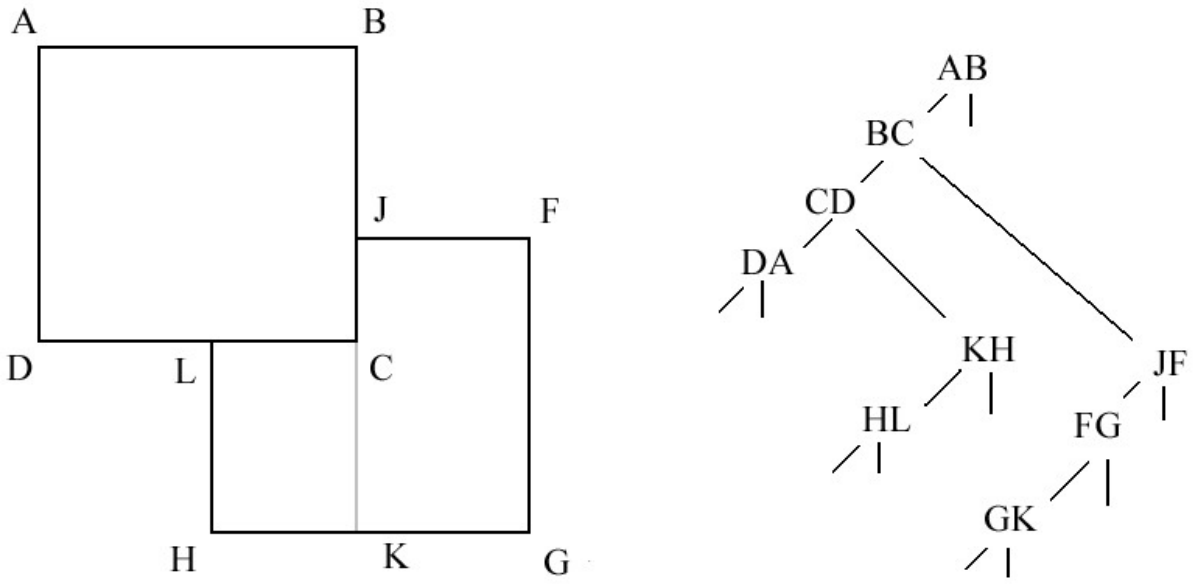
Obr. 4.2: Úvodní situace

Čtverec EFGH je postupně po hranách vkládán do stromu, podle toho, zda leží uvnitř, nebo vně. Pokud musí být některá hrana rozdělena, jsou vytvořeny nové úsečky a ty jsou vloženy do dvou větví stromu. Úseky EL a EJ leží uvnitř nově vznikaného útvaru, takže není potřebné je zahrnovat do stromu, ale pro lepší názornost je přiřadíme do větve, která náleží prvnímu čtverci.



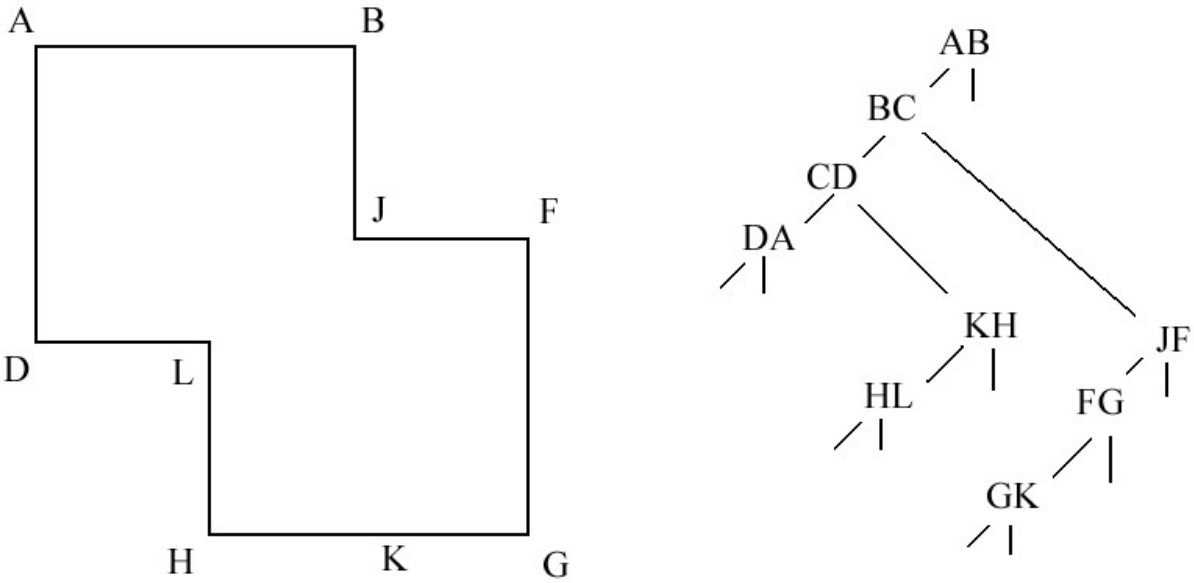
Obr. 4.3: Přidání druhého čtverce a dělení hranou BC

Ze stromu odstraníme ty uzly, které jsou uvnitř útvaru. Nyní nám zbývá odstranit poslední přebývající segmenty a to CJ a CL, pokud se nám to podaří, tak nám zůstane obrys nově vzniklého objektu. Toto provedeme tak, že budeme dělit rovinou JF. Tímto nám vzniknou dva nové segmenty. A jak již jsme si řekli dříve, vnitřní segmenty můžeme vypustit, takže dostaneme výsledný útvar.



Obr. 4.4: Téměř výsledný útvar a strom

A zde je již výsledek sloučení dvou překrývajících se čtverců.



Obr. 4.5: výsledný útvar

4.4 Detekce kolizí

Při detekci kolizí můžeme použít dva způsoby. Prvním je detekce, zda paprsek protíná nějaký objekt, což je záležitost raytracingu, druhým je zjišťování, zda se protínají dva objekty.

Pohyb v prostoru se dá popsat jednoduchou rovnicí: P = A + (T * V), kde P je pozice, kde se nacházíme, A je počáteční bod, T je doba pohybu a V je směrový vektor pohybu. Když máme výchozí bod a bod, do kterého se dostaneme za časový okamžik, můžeme zjišťovat, zda nenastala kolize. Toto ověříme tak, že pokud je jeden bod uvnitř a druhý vně, tak kolize nastala, ovšem i pokud jsou oba body mimo objekt, tak může nastat kolize například skrz roh objektu.

Při výpočtu můžeme uvažovat dva přístupy. Zaměříme se na rychlost a oželíme přesnost. Vypočítáme si bod uprostřed spojnice mezi počátečním a koncovým bodem a následně testujeme, zda se nachází na stejné straně objektu, jako bod výchozí. Nebo pokud nám záleží na přesnosti, tak pohyb nahradíme paprskem a testujeme, kde nám protne některý objekt. Tato metoda je přesná, ale je výpočetně náročnější.

4.5 Dynamické scény

BSP stromy jsou primárně určeny pro statické scény, kdy je geometrie předpočítána a uložena. Jakékoliv realtime výpočty a dělení by byly velmi náročné. Avšak práce s dynamickými objekty je také možná. Většinou se to provede tak, že dynamický objekt je nahrazen bodem. Toto má tu výhodu, že test před/za rovinou je velmi rychlý. A vložení takového bodu do stromu jako jeden uzel je také efektivní, nemusí se nic dělit a přepočítávat. Pokud je takový objekt vkládán do stromu jako potomek jiného dynamického uzlu, tak se vypočítá vzdálenost od pozorovatele, nebo se použije dělící rovina, jejíž normála je vektor z bodu k pozorovateli.

5 Popis práce algoritmu

V této kapitole popíšeme práci algoritmu, od předání geometrie, přes rozdělení rovinami až po vykreslení správné větve stromu. Vše bude popisováno v trojrozměrném prostoru

Práce algoritmu by se dala shrnout do tří kroků:

1. Nalézt dělící rovinu
2. Rozdělit polygony touto rovinou
3. Rekurzivně pokračovat ve dvou nově vzniklých částech
4. Vykreslení

5.1 Hledání dělící roviny

Způsob nalezení dělící roviny zaleží na následném použití stromu a jaké kritéria jsou při výpočtu upřednostněna (zda čas, či paměťový prostor). Dle kritérií můžeme vybírat dělící rovinu náhodně, jako jeden z polygonů ve scéně, nebo dělení rovinou, která je zarovnána podle jedné z os.

Většinou je naším záměrem, aby byl strom vyvážený, to znamená, že v každé jeho větvi je přibližně stejný počet polygonů. Toto nás sice stojí určitý čas, ale procházení takovým stromem je mnohem rychlejší. Pokud dělící rovina některý polygon protíná, je nutné ho rozdělit. A jednotlivé nově vzniklé části přiřadit do podprostoru, kam patří.

```
n_cur_score = abs(n_f_num - n_b_num) * 5 + n_s_num * 75;
```

Toto je příklad rovnice, podle které je počítána váha daného polygonu. *n_cur_score* je výsledné ohodnocení. *(n_f_num - n_b_num) * 5* nám vyjadřuje vyváženost stromu, a přiřazujeme mu váhu 5 a *n_s_num * 75* vyjadřuje počet polygonů, které budeme muset dělit. Dělení má přiřazenu velmi vysokou váhu, protože naším cílem je mít vyvážený strom s minimálním počtem dělení.

Dle této rovnice vybíráme polygon s nejnižším ohodnocením.

5.2 Dělení polygonů

Rozdělení polygonů je prováděno dle jejich vztahu k dělící rovině a to zda leží před ní, nebo za ní, případně zda je rovinou rozdělen. Toto se zjišťuje pomocí normály roviny a bodu polygonu.

Pokud dělící rovina některý polygon protíná, je potřeba ho rozdělit. Možností, jak toho docílit je více. Buď můžeme využít přímo funkcí Open inventoru, nebo obecně. Zjistíme si rovnici dělící roviny, do ní dosadíme oba body hrany, kterou dělíme. Vypočteme poměr vzdáleností od krajních bodů. A poměrem k původním souřadnicím získáme novou pozici. Uvádím ukázku kódu.

```
SbVec3f SoBSP::EdgeVert(SbVec3f *v1,SbVec3f *v2,SbPlane *plane)
{
	float x,y,z,x2,y2,z2,a,b,c,d,aa,bb,t;

	plane->getNormal().getValue(a,b,c);
	d = plane->getDistanceFromOrigin();

	v1->getValue(x,y,z);
	aa = a * x + b * y + c * z + d; // dosazení prvního bodu

	v2->getValue(x2,y2,z2);
	bb = a * x2 + b * y2 + c * z2 + d;  // dosazení druhého bodu

	if(bb - aa == 0)// singulární případ (oba leží ve stejné vzdálenosti
	return *v1;		// od roviny na stejné straně, nebo na rovině)


	t = aa / (bb - aa);// poměr vzdáleností

	return SbVec3f(x + t * (x - x2),y + t * (y - y2),z + t * (z - z2));
	// výpočet průsečíku
}
```

Kód 5.1: Ukázka kódu pro rozdělení hrany

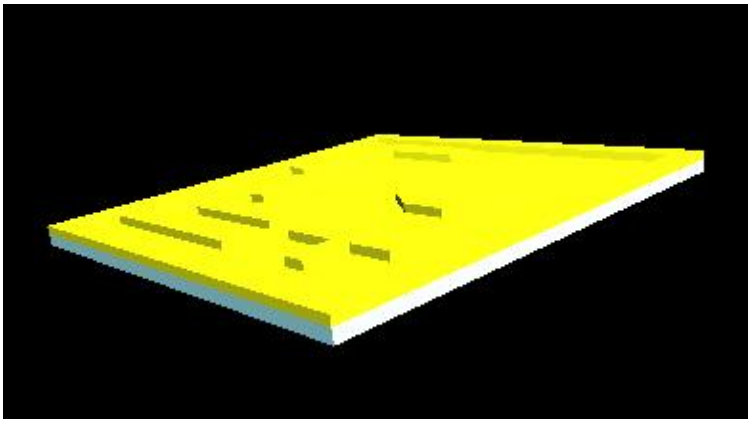
Polygony rozdělujeme tak dlouho, dokud není splněna nějaká podmínka, většinout to bývá hloubka stromu, nebo počet polygonů ve větvi.

5.3 Vykreslování stromu

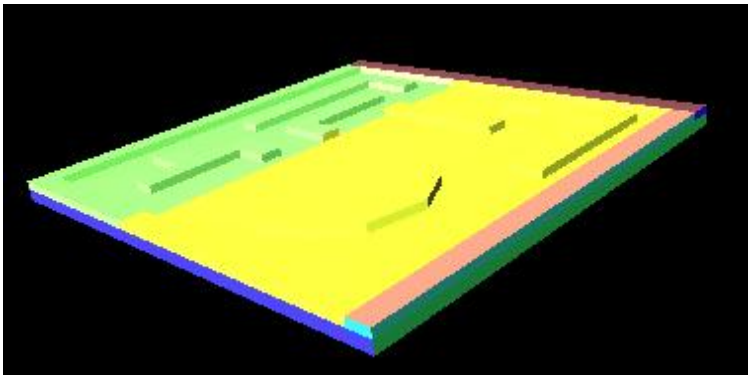
Procházíme strom od kořene a v každém uzlu zjistíme, zda bod, kde se nachází kamera leží před nebo za dělícím polygonem tohoto uzlu. Jestliže leží před, tak vykreslujeme nejprve přední uzel a poté zadní uzel. Pokud leží za, je postup opačný. Ale pokud kamera je za dělícím polygonem, znamená to, že to co je před ním „není vidět“ a tudíž je zbytečné to vykreslovat. Pokud tedy nevykreslujeme vše záměrně, jako například průhledné plochy, nebo drátěný model. Následující pseudokód ukazuje, jak funguje tento algoritmus.

```
Render(node)
{
        Otestuj pozici kamery a normálu dělící roviny
        Jestliže je kamera před
                Pokud přední uzel existue
                        Render(přední uzel)

                Vykresli polygony co jsou v rovině

                Pokud zadní uzel existue
                        Render(zadní uzel)
        Jinak pokud je kamera za
                Pokud zadní uzel existue
                        Render(zadní uzel)

                Vykresli polygony co jsou v rovině

                Pokud přední uzel existue
                        Render(přední uzel)
}
```

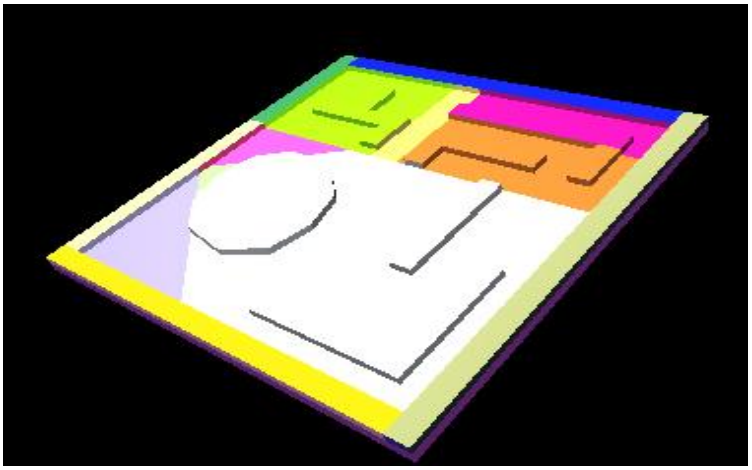
Ukázka Postupného rozdělení modelu. Model byl zapůjčen z projektu Daniela Výchopně – Tank Hunters [5]. Každý uzel je zobrazen jinou barvou. Je patrné, jak uzlů a barev přibývá s postupným dělením.



Obr. 4.1: První dělení vodorovnou rovinou



Obr. 4.2: Páté dělení



Obr. 4.3: Výsledný objekt po 10 děleních

6 Efektivita BSP algoritmu

6.1 Prostorová a časová složitost

Výpočet můžeme rozdělit na několik částí. Tou první je načtení a vložení vertexů do pole. Toto pole je neseřazené, takže jde o pouhé vložení se časovou složitostí O(1).

V dalším kroku vybíráme dělící polygon a rozdělujeme prostor. Pokud volíme rovinu náhodně, ne dle žádných parametrů, tak počet nově vzniklých plygonů může být až n^3. Takže z toho můžeme vyjádřit prostorovou složitost jako O(n^3). Při použití „chytřejší“ metody pro hledání dělící roviny bude časová složitost pro konstrukci stromu vyšší, ale razantně se redukuje počet polygonů, takže se i zmenšuje prostorová složitost. Paterson and Yao [4] dokázali, že pro trojrozměrný prostor platí při vhodné volbě dělící roviny v nejhorším případě n^2 polygonů. Tímto se snižuje počet z n^3 na n^2. Tím je i výsledný strom mnohem menší a jeho následné procházení při vykreslování je rychlejší a algoritmus výkonnější. To je velmi důležité, protože procházení stromem a výběr správného uzlu je často úzkým hrdlem aplikace.

Nyní porovnáváme polygon s dělící rovinou a přiřadíme ho do přední větve, zadní větve, nebo ho rozdělíme. Časová složitost tohoto kroku je O(n^2).

Rekurzivní opakování tohoto procesu přidává časovou složitost O(n).

Ve výsledku tedy dostaneme prostorovou složitost O(n^3), která může být zjednodušena na O(n^2) výběrem správné roviny a O(n^3) časovou složitost.

6.2 Vylepšení algoritmu

6.2.1 Minimalizace počtu dělení

Při dělení polygonů se zvyšuje počet polygonů ve scéně. To má za následek snížení rychlosti vykreslování. Proto je snaha zachovat počet polygonů co nejnižší. Tato minimalizace se dá docílit například analyticky přiřazením velké váhy při výběru dělícího polygonu. Je potřeba znát celkovou geometrii aby bylo možno vybrat správnou dělící rovinu.

6.2.2 Vyvážení stromu

Vyváženost stromu je důležitá kvůli časové náročnosti při prohledávání, protože u nevyváženého stromu mají některé větve větší hloubku, tudíž i větší časovou náročnost. Toto je podstatné obzvláště při raytracingu. Méně však už při odstraňování neviditelných ploch. A to proto, protože tam více záleží na počtu polygonů, než na hloubce stromu.

6.2.3 Vyvážení x počet dělení

Záleží na konkrétním použití stromu a podle toho zvolit metodu. Jistým kompromisem, jak vybrat vhodný dělící polygon a přitom zachovat rychlost je vybrat náhodně několik kandidátů na dělící rovinu a z nich vypočítat, který by byl nejlepší.

6.3 Měření výkonu

6.3.1 Výběr dělící roviny

V knihovně jsou implementovány tři metody výběru dělícího polygonu a jsou to:

FindRandRoot – vybere dělící rovinu náhodně
FindSemiRandRoot – Vybere n potenciálních rovin a z nich vypočte nejvhodnější
FindRoot – Vypočítá nejvhodnější rovinu (vyvážený strom a minimum trojúhelníků)

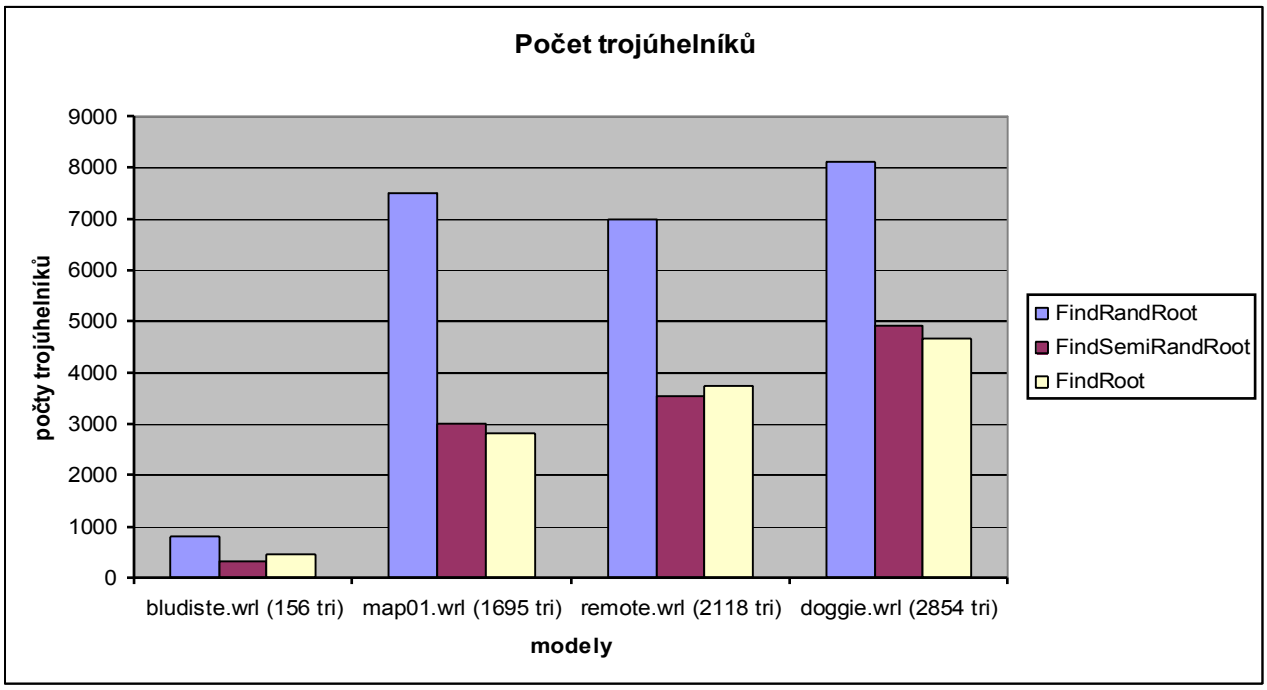
Model	Trojuhelníky, čas(s)		
	FindRandRoot	FindSemiRandRoot	FindRoot
bludiste.wrl (156 tri)	800tri / 1s	330tri / 1s	438tri / 1s
map01.wrl (1695 tri)	7500tri / 4s	3000tri / 6s	2804tri / 49s
remote.wrl (2118 tri)	7000tri / 3s	3550tri / 8s	3735tri / 63s
doggie.wrl (2854 tri)	8100tri / 4s	4900tri / 11s	4670tri / 110s

Tab. 6.1: Algoritmy, doba výpočtu a počty trojúhelníků
Výpočty byly prováděny na počítači AMD 3000+, 1GB RAM, GF6600GT

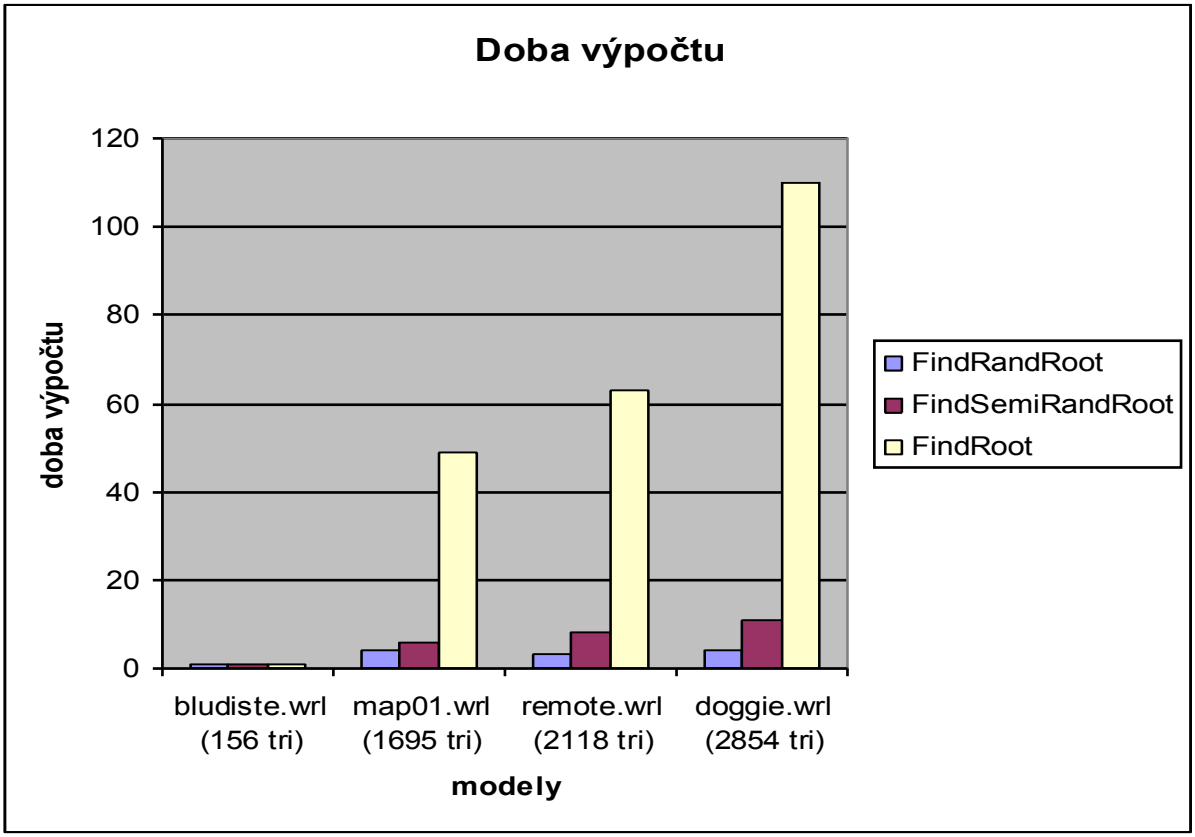
Hodnoty v tabulce jsou průměrné z 10 pokusů a zaokrouhlené, ale pro názornost a možnost porovnání to stačí. Výška stromu byla nastavena na 10. Pro metodu FindSemiRandRoot byl nastaven počet náhodných potenciálních rovin na 50.

Z tabulky je patrné, že metoda FindRandRoot má konstantní časovou složitost (čas byl měřen od startu aplikace, tudíž je zahrnuta i doba načítání modelů, inicializace atd.), ale počet trojúhelníku je velmi vysoký. Metoda FindRand má nejlepší výsledky co se týče vyváženosti a počtu polygonů (v jistých případech ji může metoda FindSemiRandRoot překonat, ale zde dojde k nevyváženosti stromu). Pro praktické použití doporučuji prostřední metodu má velmi dobré výsledky, např. u nejsložitějšího modelu se zrychlila doba výpočtu 10x, ale nárůst trojúhelníků je pouze 5%.

Pro názornost dodávám ještě dva grafy:



Graf 6.1: Počet trojúhelníků



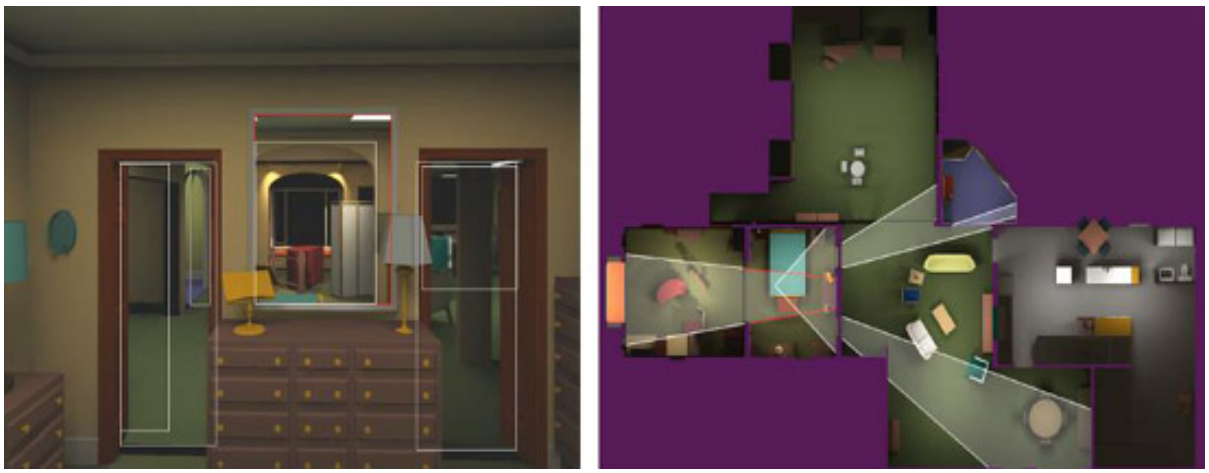
Graf 6.2: Doba výpočtu

7 Rozšíření BSP stromu

Protože BSP algoritmus slouží pouze k rozdělení prostoru, ale ne už k optimalizaci jeho vykreslování (renderuje se vše). Tak je potřeba udělat nějakou „nadstavbu“, která umí vybrat pouze ty části scény které budou vidět. Vybral jsem dva dnes asi nejpoužívanější a spolu související algoritmy. PVS – Potentially Visible Sets a portálový algoritmus. Oba dva jsou velmi složité a propracované. Hlubší popis by zabral na druhou bakalářskou práci, ale pokusím se aspoň nastínit jejich vytváření a fungování.

7.1 Portály

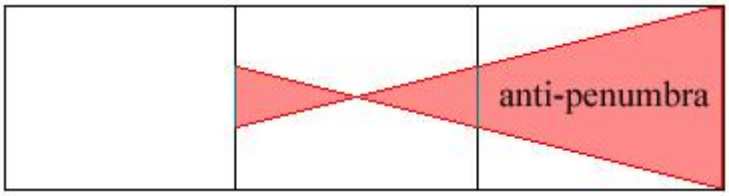
Nejprve bych vysvětlil fungování portálů, protože generování PVS je na nich založeno. Portál označuje v reálu nějaký průchod, nebo spojení. Stejně tak to bude i v grafice. Tentokrát nám bude portál spojovat dva bloky objektu. Například to můžou být dveře mezi dvěmi místnostmi. Protože jde o rozšíření BSP, tak nám takový portál bude spojovat dva uzly stromu.



Obr 7.1: Vlevo pohled pozorovatele, vpravo ukázka pohledů přes portály převzato z [6]

Postup při vytváření portálu je následující. V každém uzlu si vytvoříme portál. Při zařazování polygonu se testovalo, zda je před, nebo za dělící rovinou. Stejně tak s portálem, pokud leží přímo na rovině, tak leží v obou částech stromu. Proto ho zařadíme do obou větví a takto pokračujeme celým stromem, až dojdeme nakonec. Pokud portál přesahuje objekt, tak se může „vyčnívající“ část oříznout. Nyní vytvoříme portál v dalším uzlu a pokračujeme stejným způsobem.

S portály souvisí pojem „anti-penumbra“, nenašel jsem český ekvivalent, proto budu používat toto slovo. Co to je? Tento pojem nejsnáze popíšeme, když stojíme ve dveřích jedné místnosti a díváme se přes druhé dveře do další místnosti. To co je vidět, to je anti-penumbra. Nejlépe to ilustruje následující obrázek:



Obr 7.2: Anti-penumbra

7.2 PVS

Nyní když už máme vysvětleny portály, můžeme přistoupit k PVS. Zjednodušeně řečeno, je to seznam uzlů, které je vidět z aktuálního uzlu. Takže při vykreslování si zjistíme uzel, ve kterém se nachází kamera. A nyní již nevykreslujeme všechny uzly, tak jako v BSP stromu, ale pouze ty uzly, které můžeme vidět. Nyní ale stále vykreslujeme i ty uzly, které sice by jsme mohli vidět, ale kamera je otočena směrem ke zdi, takže je nevidíme. Proto se tato metoda rozšiřuje ještě o ořezání uzlů, které nejsou ve výhledu. Tím dosáhneme obrovského zrychlení, kdy z budovy, která se skládá z 1 000 000 polygonů vidíme pouze jednu místnost s 10 000 polygony. Je to velmi výborné urychlení.

8 Závěr

Zde popisovaný algoritmus měl v minulosti velký význam pro posunutí kvality a rychlosti zobrazování o veliký kus dopředu. Je využíván v obrovském množství aplikací. A určitě se bude používat i nadále v budoucnu. Má mnoho různých modifikací a variant.

Seznámil jsem se s knihovnou Open Inventor. Je to velmi mocný nástroj, který do značné míry zrychluje vývoj trojrozměrných aplikací a také zrychluje svými optimalizacemi jejich běh. Měl jsem již kdysi možnost setkat se s různými vývojovými nástroji, či jen grafickými API(OpenGL nebo Direct3D). Také jsem již v minulosti implementoval některé jednodušší algoritmy na optimalizaci vykreslování či jiných výpočtů, ale BSP algoritmus mi připadal složitý a proto jsem ho stále odkládal. Nyní vím že to není pravda, algoritmus je ve své podstatě jednoduchý a přitom velmi účinný. Mnou vytvořená knihovna umožňuje vytvoření takového stromu po jednoduchém nastavení parametrů automaticky. Stará se také o vykreslovaní. Skládá se z funkcí, které je snadné přeimplementovat dle vlastních požadavků.

Do budoucna bych chtěl v rozšiřování knihovny pokračovat, například při tvorbě diplomové práce. Vidím zde možností doplnění o Potentially Visible Sets, neboli určení, které části scény by mohly být vidět z daného uzlu. Nebo také zakomponovat portály, to by mohla být zajímavá práce, je tam mnoho možností na vylepšení. Také je možnost udělat několik variant typů BSP algoritmu. Aby si mohl programátor vybrat ten, který je pro jeho aplikaci nejvhodnější.

Literatura

[1] Manuál a příklady dostupné na adrese http://www.coin3d.cz (4/2006)

[2] Článek dostupný na adrese http://www.root.cz/clanky/open-inventor/ (4/2006)

[3] Manuál a ukázkové příklady dostupné na adrese http://www-evasion.imag.fr/Membres/Francois.Faure/doc/inventorMentor/sgi_html (4/2006)

[4] M. S. Paterson and F. F. Yao. Efficient binary space partitions for hidden-surface removal and solid modeling. Discrete Comput. Geom., 5:485–503, 1990.

[5] D. Výchopeň ,Tank Hunters, http://merlin.fit.vutbr.cz/wiki/index.php?title=Inventor_Projects_2005 (4/2006)

[6] Luebke, D. and Georges, C. Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets, 1995

[7] Diskusní skupiny na adrese http://www.ceskehry.cz/forum (4/2006)

Přílohy

a) Popis parametrů programu

V souboru SoBSP.h je možna změnit implicitní nastavení nekolika definovaných parametrů pro BSP strom. Tyto parametry je možno nastavit také před generováním stomu pomocí funkce SetParams(...).

BSP_HEIGHT

Určuje maximální výšku stromu, dokud se bude počítat. 1 znamená, že se nebude nic dělit a všechny polygony budou přiřadezy kořenovému uzlu.

BSP_MAX_TRI

Určuje maximální počet trojúhelníků v uzlu. Pokud je počet trojúuhelníku větší, dochází k dělení.

BSP_METHOD

Metoda výběru dělícího polygonu. Jsou to tyto 3:

- BSP_METHOD_RAND – vybere náhodný polygon
- BSP_METHOD_SEMI_RAND – vybere několik náhodných polygonů a z nich vypočítá ten nejvhodnější(podobně jako následující metoda)
- BSP_METHOD_BEST – Prohledává všechny polygony v uzlu a vypočítá, který je nejvhodnější s ohledem na vyváženost stromu a počet nově vzniklých polygonů

BSP_SEMIRAND_COUNT

Určuje počet vzorků pro metodu BSP_METHOD_SEMI_RAND

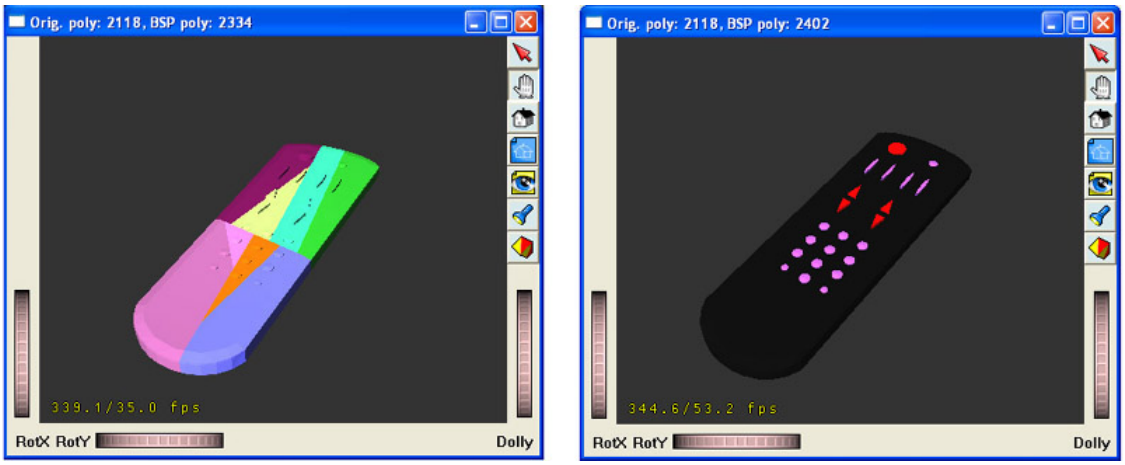
BSP_VIEW_PLANES

Zobrazí dělící roviny, vhodné pro ladění aplikace

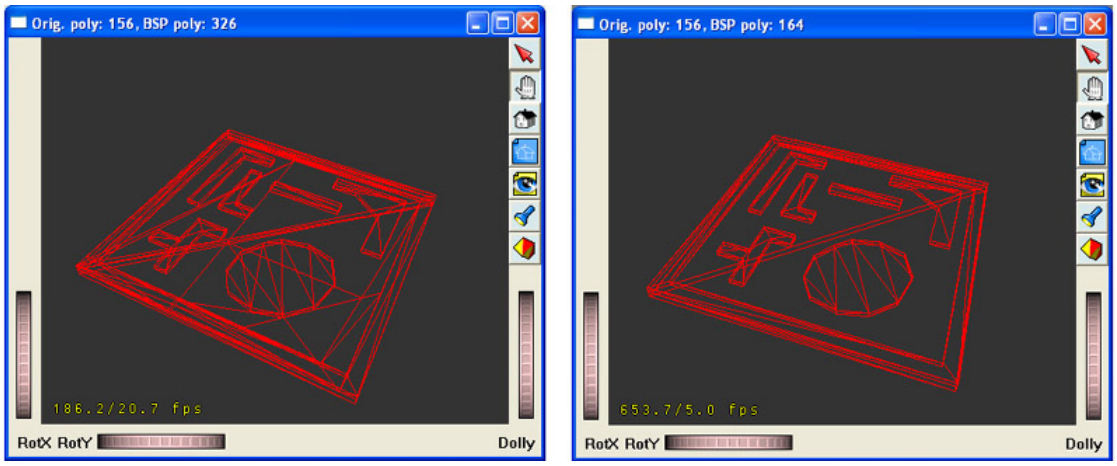
U programu BSPview.exe je možné jako parametr zadat jméno modelu, který chce uživatel načíst a zobrazit se základními parametry.

Př.: „**BSPview.exe bludiste.wrl“**

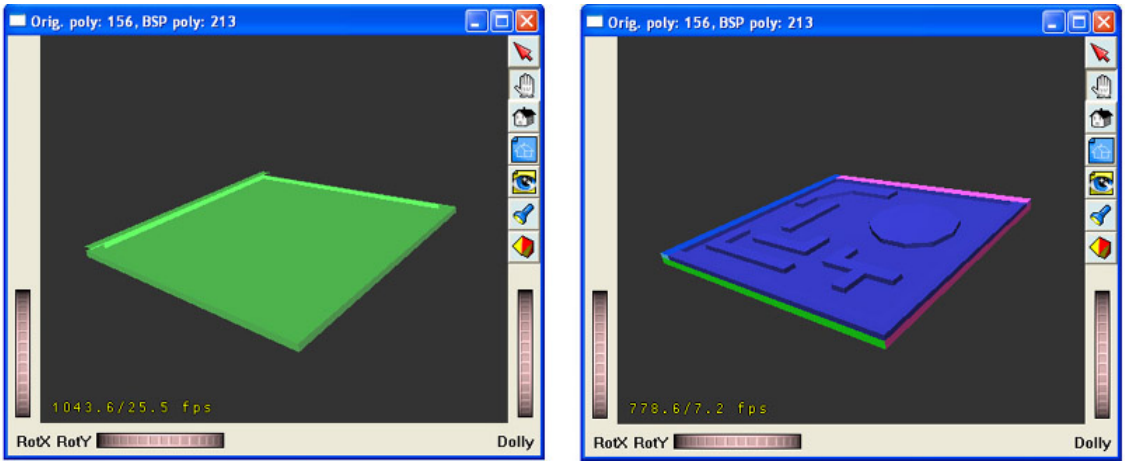
b) obrázky z aplikace



Rozdělení modelu na nekolik částí a následné vykreslení



Drátěný model po několika děleních(vlevo) a před dělením(vpravo)



Vlevo jsou zobrazeny dělící roviny, vpravo jsou vidět segmenty po rozdělení