Západočeská univerzita v Plzni

Fakulta aplikovaných věd

Katedra informatiky a výpočetní techniky

Počítačová grafika

DIPLOMOVÁ PRÁCE

Zobrazování povrchu scén definovaných pomocí implicitních funkcí a CSG stromů

Vypracoval:	Martin ČERMÁK
Vedoucí diplomové práce:	Prof. Ing. Václav Skala, Csc.

Rok a místo vydání:	Plzeň, 2001

Děkuji rodičům, bratrovi a přátelům za duševní i finanční pomoc během studia.

Prohlašuji, že jsem svou diplomovou práci vypracoval výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Plzni dne 22. 8. 2001 Martin Čermák

Obsah

1 Úvod

Většina programů umožňujících modelování 3D těles, využívá pro jejich reprezentaci jednoduchých primitiv jako je přímka, plocha, kvádr, atd. Taková reprezentace těles spočívá v popisu jejich povrchu, tedy v popisu množiny hraničních bodů (*Boundary Representation, B - rep*). Modelovat objekty hladkých organických tvarů je s takovouto reprezentací velmi obtížné ne-li nemožné a to i v případě, že použitými primitivy jsou kulové nebo beziérovy/spline plochy. Z těchto důvodů neustále vzrůstá popularita modelování objektů pomocí implicitních funkcí. Povrch takto definovaného objektu se stává *izoplochou*[1] v prostoru, který je danou funkcí ohodnocen.

Myšlenka vyjádření těles implicitními funkcemi není nová. Jim Blinn v roce 1982 (viz [Blin82]) navrhl chápat izoplochy vzniklé při modelování elektrického potenciálu elementárních částic jako objekty. Postupem času byla tato technika rozšiřována a tyto objekty dostávaly rozličné názvy (*Blobby objects* [Blin82], *Soft objects* [Wyvi86b], *Meta balls*). Jules Bloomenthal upozornil (viz [Bloo88]), že všechny tyto objekty patří do stejné kategorie a mohou proto být označovány jediným názvem *implicitní plochy* (*implicit surfaces*).

1.1 Implicitní plochy

Implicitní plocha je množina bodů $\boldsymbol{p}$, pro které platí $f(\boldsymbol{p}) = 0$, kde $\boldsymbol{p} \in \Re^3$, viz [Bloo95]. Taková plocha je známá pod názvem *zero set* funkce f a lze ji zapisovat jako $f^{-1}(0)$ nebo $Z(f)$. V souladu s větou o implicitních plochách lze psát: pokud je nulová hodnota regulární hodnotou funkce, potom množina nulových hodnot (*zero set*) je dvou-dimenzionální manifold.

Izoplocha je obdobná množina bodů $\boldsymbol{p}$, pro které platí $f(\boldsymbol{p}) = c$, kde c je hodnota vrstevnice (*iso-contour*) implicitní plochy.

Ve většině případů funkce f rozděluje prostor na část vnitřní a část vnější, tj. vnitřek funkce a okolí. Podle konvencí (viz [Bloo95]) je f obvykle zapisována jako $f(\boldsymbol{p}) < 0$, což označuje množství bodů (objem prostoru) uzavřených hranicí (povrchem funkce) $f(\boldsymbol{p}) = 0$. Tato schopnost uzavřít objem a schopnost reprezentace spojení objemových těles je základní výhodou v návrhu geometrických objektů pomocí implicitních funkcí.

---

[1] izoplocha je množina bodů se stejnou funkčí hodnotou

Používány jsou i jiné formy zápisu implicitních funkcí, např. *F-Rep* (funkcionální reprezentace), kde je funkce f zapisována jako $f(\boldsymbol{p}) \geq 0$. Podrobnější popis je uveden v kapitole 2.2.

Další výhodou implicitní reprezentace objektů je její mnohdy vyšší čitelnost než u parametrického tvaru. Například, uvažujme funkci pro kouli (*sphere*) se středem $\boldsymbol{c} = (c_x, c_y, c_z)$ a poloměrem r. Tato funkce může být vyjádřena parametricky jako množina bodů $\{\boldsymbol{P}\}$, pro které platí:

$$(p_x, p_y, p_z) = (c_x, c_y, c_z) + (r\cos\beta\cos\alpha, r\cos\beta\sin\alpha, r\sin\beta) \tag{1.1}$$

nebo v implicitní reprezentaci jako rovnice:

$$(p_x - c_x)^2 + (p_y - c_y)^2 + (p_z - c_z)^2 - r^2 = 0. \tag{1.2}$$

1.2 Abstract

This document introduces some principles of implicit modeling and visualization methods of implicit defined objects.

The theoretical part is directed at summary of methods for implicit surfaces modeling and visualization including its using, advantages and disadvantages.

The practical part includes methods for visualization of implicit objects. We aim one's attention to parallel polygonization and problem of disjoint objects in scene.

In the conclusion, the reached results are appraised and future work is presented as well.

1.3 Rozvržení dokumentu

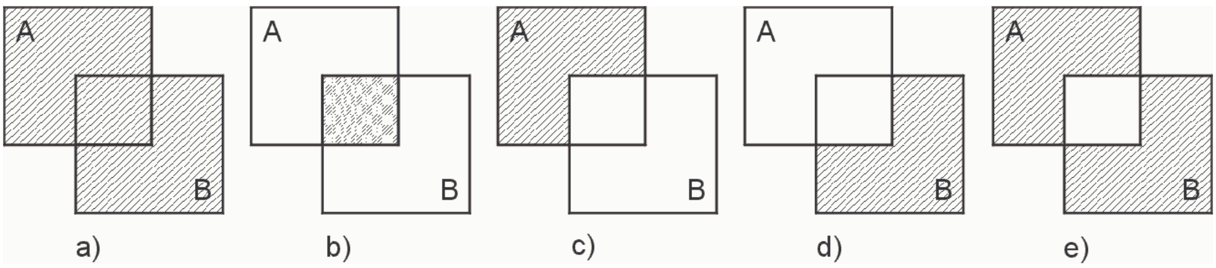
Tato diplomová práce je rozčleněna do sedmi kapitol. Po úvodní části následuje druhá kapitola, ve které jsou nastíněny metody používané pro modelování objektů pomocí implicitních funkcí. Ve třetí kapitole jsou uvedeny algoritmy pro vizualizaci povrchů takto definovaných objektů. Čtvrtá kapitola se zabývá konkrétními implementovanými algoritmy pro polygonizaci povrchů implicitních funkcí, které jsou v páté kapitole vyhodnoceny a testovány. Šestá kapitola je závěrem a zhodnocením dosažených výsledků společně s nastíněním budoucí práce. Sedmou kapitolou je seznam použitých zdrojů a literatury.

2 Modelovací techniky

V této kapitole budou vysvětleny základní principy a vlastnosti modelování objektů definovaných implicitními funkcemi. Bude představeno modelování implicitních těles pomocí CSG stromů, funkcionální reprezentace a modelování založené na kostře objektů.

2.1 Konstruktivní geometrie těles

V některých aplikacích, zejména v oblastech CAD systémů, jsou tělesa popisována způsobem, který odráží postupy používané konstruktérem při tvorbě tělesa. Metoda nazývaná *Konstruktivní Geometrie Těles*, zkráceně *CSG* (*Constructive Solid Geometry*), je založena na reprezentaci tělesa stromovou strukturou (CSG stromem), uchovávající historii dílčích konstrukčních kroků. Z jednoduchých geometrických objektů, tzv. CSG primitiv, je pomocí množinových operací a prostorových transformací vytvořen výsledný objekt. Na obrázku 2.1 je znázorněno 5 základních Booleovských množinových operací, které jsou použitelné pro objemové modelování.



Obrázek 2.1: Booleovské operace mezi dvěma tělesy; a) sjednocení A+B, b) průnik AB, c) rozdíl A-B, d) rozdíl B-A, e) symetrická diference (A-B)+(B-A).

Vnitřní uzly CSG stromu obsahují logické operace a v listech jsou uchovávány informace o CSG primitivech. Prostorové transformace (rotace, posun, změna měřítka) mohou být chápány jako CSG operace. Tehdy jsou transformační koeficienty umístěny do samostatných listů CSG stromu. Jiný přístup zaznamenává transformace ke každému primitivu, takže vnitřními uzly jsou pouze množinové operace. Mezi základní geometrická CSG primitiva patří kvádr, koule, válec, kužel, toroid a poloprostor. Poloprostor je praktický geometrický prvek s jehož pomocí lze v CSG stromu efektivně ořezávat operacemi průniku nebo rozdílu. Například kvádr lze definovat jako průnik šesti poloprostorů, rovina *xy* může být definována jako $f(x, y, z) = z$, atd.

Matematické vyjádření základních Booleovských operací pro objekty definované implicitními funkcemi f_1 a f_2 vypadá takto:

- sjednocení (*union*) $min\{\ f_1(\boldsymbol{p}),\ f_2(\boldsymbol{p})\ \}$, (2.1)
- průnik (*intersection*) $max\{\ f_1(\boldsymbol{p}),\ f_2(\boldsymbol{p})\ \}$,
- rozdíl (*differences*) $max\{\ f_1(\boldsymbol{p}), -f_2(\boldsymbol{p})\ \}$.

Poznámka:

Uvedené logické operace jsou platné pro definici implicitních funkcí podle [Bloo95], tj. $f(\boldsymbol{P}) < 0$. Pokud je funkce definována v *F-Rep* (viz následující kapitola), jsou operace *min*/*max* prohozeny.

2.2 Funkcionální modelování

Funkcionální reprezentace, zkráceně *F-rep* (*Function Representation*), definuje celý geometrický objekt pomocí jedné spojité reálné funkce několika proměnných $f(\boldsymbol{p}) \geq 0$ (2.2).

F-rep je dalším krokem pro obecnější modelování s využitím reálných funkcí. Na funkce nejsou kladeny složité omezovací podmínky, postačuje, jsou-li alespoň C^0 *spojité*[2]. Funkce mohou být definovány vzorcem nebo vyhodnocovány procedurou. V tomto smyslu je F-rep kombinací mnoha rozdílných modelovacích technik, jako je klasické implicitní modelování (*classic implicits*), modelování založené na kostře (*sceleton based implicits* [Bloo95], viz kapitola 2.3), *set-theoretic solids*, *sweeps*, *volumetric objects*, parametrické a procedurální modelování (*parametric and procedural models*).

Základními modelovacími technikami jsou množinové operace sjednocení, průniku, rozdílu, atd. podobně jako v případě CSG stromů. Hlavním omezením známých vztahů pro množinové operace mezi implicitními funkcemi (vzorce 2.1), které využívají *min*/*max* operace je jejich C^1 nespojitost, která může mít za následek neočekávané výsledky v dalších výpočtech s takto definovanými objekty. V F-rep jsou využívány pro reprezentaci Booleovských operací tzv. R-funkce (*R-functions*, definice viz [Rvachov] – definoval C^k spojitost). Před uvedením konkrétních vztahů pro množinové operace definujeme pojem *R - funkce* tak, jak je uveden v [Rvachov].

Reálná funkce reálných proměnných je nazývána R-funkcí, pokud platí, že funkce může měnit své znaménko právě když alespoň jeden z jejích argumentů změnil své znaménko.

---

[2] Spojitá funkce, která může obsahovat ostré hrany.

Nejjednodušší operace mezi dvěmi implicitními funkcemi f_1 a f_2 vypadají takto:

- sjednocení (*union*) $$f_1 \mid f_2 = f_1 + f_2 + \sqrt{f_1^2 + f_2^2}\,, \tag{2.3}$$
- průnik (*intersection*) $$f_1 \,\&\, f_2 = f_1 + f_2 - \sqrt{f_1^2 + f_2^2}\,,$$
- rozdíl (*differences*) $$f_1 \setminus f_2 = f_1 - f_2 - \sqrt{f_1^2 + f_2^2}\,.$$

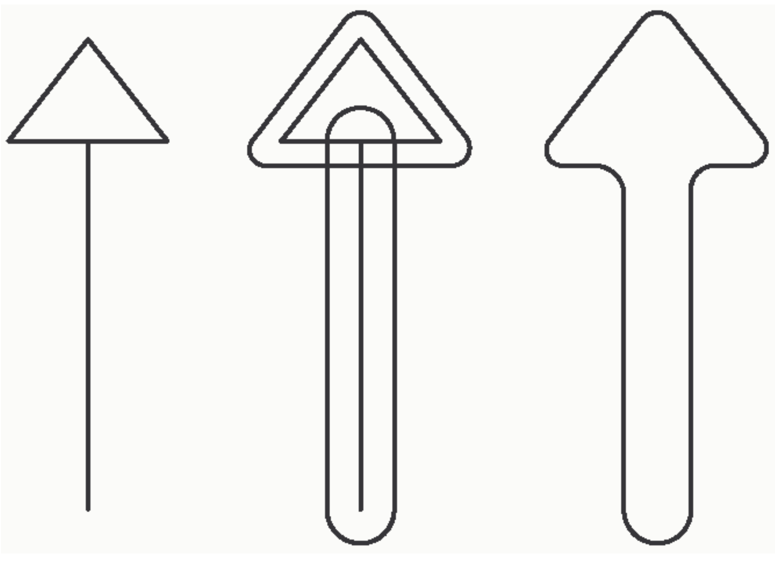
Logické operátory &, |, ~, \, atd. jsou nazývány *R-operátory*.

Mezi další operace, které je možné provádět s F-rep objekty patří:

- *Blending* - operace, která vyhlazuje hrany geometrických těles,
- *Offsetting* - vytváří rozšířenou verzi původního objektu,
- *Cartesian product* – operace, která převádí 2D objekt do 3D,
- *Bijective mapping* – modelovací operace, která slouží k deformaci původního tělesa,
- *Metamorphosis* – vytváří objekt, který je přechodem mezi dvěma původními objekty.

2.3 Modelování založené na kostře

Modelování založené na kostře objektu (*Skeleton based modeling*) je rozšířenou modelovací technikou v implicitní reprezentaci těles. Základem objektu je kostra, jejíž každá část definuje nějaký typ primitivního implicitního objektu. Výsledný modelovaný objekt vzniká sjednocením takto definovaných částí. Znázorněno na obrázku 2.2.



Obrázek 2.2: Kostra (vlevo) definuje dvě implicitní primitiva (uprostřed), která tvoří výsledný objekt (vpravo).

Kostra objektu může být složena z několika základních částí.

- Body – Středy jednoduchých kvadrik (koule, elispoid), nebo superkvadrik.
- Křivky – Množiny centrálních os válců s proměnným průměrem.
- Polygony – Síť polygonů a křivek se používá k vytvoření vyváženého povrchu.

Modelování objektů založené na kostře se často používá ve spojení s modelováním organických látek, lidských orgánů, atd. Na obrázku 2.3 je znázorněn model ruky, vytvořený pomocí kostry.

Obrázek 2.3: Model ruky vymodelovaný pomocí kostry.

3 Zobrazování povrchu implicitních funkcí

V této kapitole budou představeny základní metody pro zobrazování povrchu objektů definovaných implicitními funkcemi s jejich výhodami i nevýhodami. Budou vysvětleny některé algoritmy, které pro svou činnost vyžadují počáteční bod (*start point*) ležící na povrchu funkce a také metoda *Exhaustive search*, která takový bod nevyžaduje. Obě polygonizační metody budou porovnány s algoritmem *Ray-tracing*, který je určen pro přímé zobrazování povrchu implicitních funkcí.

Nejdříve je však nutné objasnit problémy, které s vizualizací implicitních funkcí souvisí a které jsou hlavní příčinou vzniku takového množství nejrůznějších algoritmů určených pro jejich zobrazování.

3.1 Problém nalezení kořene rovnice

Jedním z hlavních problémů při vizualizaci povrchu implicitních funkcí je obtížnost hledání kořene rovnice $f(\boldsymbol{p}) = 0$, tj. nalezení takového bodu $\boldsymbol{p}$, který rovnici splňuje. Zpravidla nelze z rovnice explicitně vyjádřit jednu ze souřadnic, jejíž hodnoty by mohly být vypočítávány standardními postupy. Souřadnice bodu $\boldsymbol{p}$ musí být proto voleny s využitím prostředků numerické matematiky. Problém nalezení kořene rovnice potom spočívá v hledání minima funkce f vhodnou volbou souřadnic bodu $\boldsymbol{p}$.

V následujících odstavcích budou přestaveny možné metody, které vedou k výsledkům, ale jejichž použití je vždy jistým způsobem omezeno, buďto rychlostí polygonizace nebo nemožností nalézt objekty sestávající se z více oddělených částí.

3.2 Polygonizační algoritmy

Polygonizační algoritmy aproximují povrch implicitních objektů polygonální sítí a jsou nejčastěji používány jako náhled pro interaktivní modelování. Vytvořené polygonální modely jsou zobrazitelné běžnými grafickými akcelerátory a převoditelné do datových struktur jiných aplikací, určených pro jejich další zpracování.

Aproximace matematicky definovaného tělesa polygonální sítí s sebou nutně nese určitou chybu, jejíž vyjádřením se také budeme zabývat v následujících odstavcích.

3.2.1 Metody založené na počátečním bodě

Metody tohoto typu se vyznačují vysokou rychlostí polygonizace povrchu funkce, což patří mezi jejich hlavní přednosti. Rychlost těchto metod spočívá v tom, že polygonizační algoritmus „cestuje“ pouze po povrchu implicitního objektu a zbytek prostoru nekontroluje. Vstupním prvkem takového algoritmu je bod na povrchu funkce, kterou chceme vizualizovat. S tímto bodem však souvisejí následující problémy.

Obecná scéna

Uvažujeme-li obecnou scénu definovanou implicitními funkcemi, ve které je umístěno více oddělených objektů a o které nejsou známy bližší informace, vyplynou na povrch následující otázky.

- Kolik oddělených objektů je obsaženo ve scéně?
- Jak nalézt počáteční bod pro každý z nich?

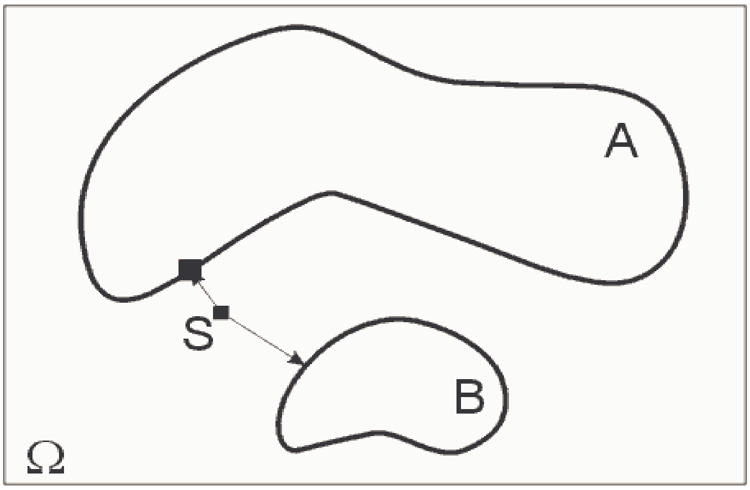
Ani na jednu z těchto otázek není známa uspokojivá odpověď, která by přišla v tak dobrém čase, aby stále platila zmíněná přednost, tj. vysoká rychlost polygonizace. Oblast použití těchto metod je v takových případech omezena na scény, které obsahují jeden spojitý objekt.

Scéna se známou definicí implicitní funkce

Máme-li k dispozici další informace o zobrazované scéně, jsou metody využívající počáteční bod, vhodné i pro zobrazování složitějších scén s více objekty v reálném čase (min. 10 snímků/sec). Autoři v [Triq01] využívají znalostí kostry, pomocí které definují vlastní scénu. Z každého bodu kostry (středu primitiva) hledají počáteční bod v jednotném směru a po jeho nalezení generují povrch. Tím je zajištěno bezpečné nalezení všech oddělených objektů ve scéně, neboť pro každý objekt existuje vlastní startovací bod.

Nalezení startovacího bodu

Bez jakýchkoliv dalších informací o vstupní funkci je nutné hledat startovací bod jiným způsobem než v předchozím odstavci a není zaručeno nalezení všech objektů. Na obrázku 3.1 je znázorněn možný způsob, jakým algoritmus pro hledání počátečního bodu, nalezne v oblasti Ω bod na povrchu jedné z funkcí A, B. Pro hledání počátečního bodu je možné využít metody *Random search* [Bloo94] o které se podrobněji zmíníme v kapitole 4.2.2, nebo numerické vyhledání povrchu implicitní funkce, kapitola 4.2.1.



Obrázek 3.1: Nalezení počátečního bodu na povrchu funkce numerickým výpočtem.

Pokud je nalezen příslušný počáteční bod, je možné implicitní funkci polygonizovat[3]. Budou představeny dvě metody pro polygonizaci povrchu funkce, známá metoda *Marching cubes* (kapitola 4.5) a méně známá metoda *Marching triangles* (kapitola 4.6).

3.2.2 Úplné prohledávání definované oblasti

Metody založené na úplném prohledávání oblasti prostoru se vyznačují schopností nalézt všechny oddělené části implicitně definovaného objektu. Metody tohoto typu jsou často používány v aplikacích pro zpracování objemových dat, kdy je v zadaném objemu vyhledávána izoplocha. Tato oblast musí být prohledána beze zbytku, neboť hledaná izoplocha se může vyskytovat kdekoliv.

Úplné prohledávání prostorové oblasti má za následek $O(n^3)$ složitost implementovaných algoritmů a tím i poměrnou pomalost výpočtů. Algoritmus určený pro implicitní funkce je uveden v kapitole 4.3 a jmenuje se *Exhaustive search*.

3.2.3 Chyba aproximace implicitního povrchu

V dostupné literatuře jsem nevyčetl odpověď na otázku: „Jak kvalitní je výsledná aproximace implicitní funkce?“, tj. jak přesně odpovídá výsledný polygonální objekt svému matematickému modelu. Z tohoto důvodu jsem navrhl a do svého modulu začlenil funkce pro zjišťování kvality aproximace.

[3] vytvářet polygonální síť, která aproximuje povrch objektu

Základ algoritmu je velice jednoduchý a vychází z definice implicitního povrchu, tj. z rovnice *f(X) = 0*. Máme-li vygenerován polygonální objekt, jehož všechny body leží přesně na povrchu implicitní funkce, musí pro součet funkčních hodnot ve všech těchto bodech platit:

$$E_{absolute} = \sum_{i=0}^{N-1} \left| \; f(x_i) \; \right| = 0, \tag{3.4}$$

kde N je celkový počet vygenerovaných bodů a $\boldsymbol{x}_i$ je bod na povrchu objektu.

V praktickém výpočtu je tento součet nenulový a odpovídá absolutní odchylce aproximace polygonálního objektu od matematického modelu. Tento součet se může lišit podle počtu vygenerovaných bodů, a proto je nutné vzorec (3.4) upravit do podoby:

$$E_{average} = \frac{\sum_{i=0}^{N-1} \left| \; f(x_i) \; \right|}{N}$$

(3.5). Tak získáme průměrnou odchylku každého bodu aproximovaného objektu od matematického modelu, která není závislá na počtu vygenerovaných bodů. Průměrná odchylka je zároveň nezávislá na velikosti objektu, což může v některých případech vést k jistému zkreslení. Pokud je např. průměrná odchylka rovna 10^{-2} u objektu *Sphere* o poloměru *1cm*, není objektivní závěr, že objekt *Sphere* o poloměru *1000cm* se stejnou průměrnou odchylkou 10^{-2} je aproximován se stejnou chybou. V takovém případě je nutné brát v úvahu i velikost měřeného objektu.

Jako jistý odhad velikosti měřeného objektu může sloužit jeho celkový obsah, popř. objem. Vzhledem k jednoduchosti výpočtu celkového obsahu objektu složeného z trojúhelníkových ploch, je v našem případě použit právě obsah, jako měřítko velikosti objektů.

Vzorec (3.5) je potom upraven na tvar:

$$E_{relative} = \frac{E_{average}}{P}$$

(3.6), kde P je celkový obsah polygonálního objektu, který je počítán podle vzorce (3.7).

$$P = \sum_{i=0}^{M} P_i = \frac{1}{2} \sum_{i=0}^{M} \left| \mathbf{u}_i \times \mathbf{v}_i \right|$$

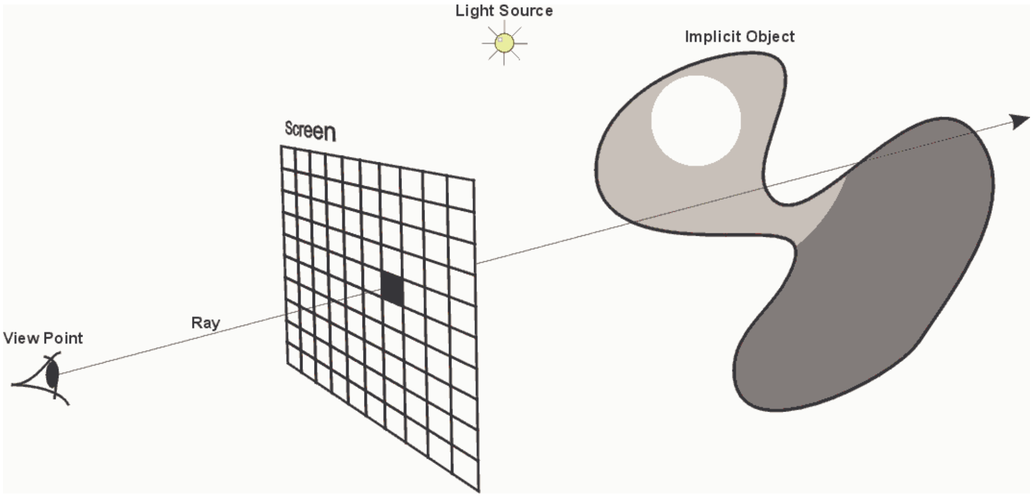
(3.7), kde M je celkový počet trojúhelníků, P_i je obsah jednotlivých trojúhelníků a $\boldsymbol{u}_i$ a $\boldsymbol{v}_i$ je dvojice vektorů vycházejících ze společného bodu v každém trojúhelníku.

Relativní chyba aproximace tak, jak byla definována, je použitelná pro posouzení kvality polygonálního modelu s přihlédnutím k jeho velikosti. V kapitole 5.3 jsou hodnoceny dosažené výsledky a porovnávána je i průměrná odchylka.

3.3 Ray tracing pro implicitní funkce

Algoritmus *Ray-tracing* je oproti polygonizačním technikám určen pro přímou vizualizaci povrchu implicitních funkcí. Vytváří digitální obrazy vysoké kvality, kterým také odpovídá doba výpočtu.

Základ algoritmu se nijak neliší od klasického *Ray-tracing* algoritmu pro zobrazování scén s polygonálně definovanými objekty, nebo od zobrazování volumetrických dat.



Obrázek 3.2: Princip algoritmu *Ray-tracing*.

Algoritmus (3.1)

Každým pixelem obrazového displeje je do zobrazované scény vyslán paprsek (*ray*), viz obrázek 3.2. Parametrickou rovnici paprsku ***r*** lze zapsat:

$$r(t) = a + t \cdot b, \tag{3.1}$$

kde ***a*** je zdroj paprsku, ***b*** je směrový vektor a *t* je parametr.

Pro hledaný průsečík s povrchem funkce platí rovnice:

$$f(\mathbf{r}) = f(\mathbf{a}, \mathbf{b}, t) = 0, \tag{3.2}$$

zkráceně

$$f(t) = 0. \tag{3.3}$$

Pro každý paprsek je prováděn test, zda nedošlo k průsečíku s tělesem v zobrazované oblasti. Pokud k průsečíku nedošlo, je pixel obarven barvou pozadí. Pokud došlo k protnutí tělesa, tj. v implicitní reprezentaci to znamená změnu znaménka funkce na dráze paprsku, je vypočítána přesná pozice průsečíku a jeho normálový vektor.

Přesné souřadnice průsečíku paprsku a tělesa je možné vypočítat numerickou metodou půlením intervalu (*binary subdivision*), popř. *regula falsi*. Složky normálového vektoru mohou být vypočítány podle vzorce 4.1.

Nejznámějším algoritmem pro vizualizaci implicitních funkcí, který je založen na sledování paprsku, je *Ray-marching*. Tento algoritmus řeší problém detekce průsečíku s objektem „brutální silou“ (*brute force*). Vyhodnocuje funkci $f(t)$ v každém kroku v celé délce paprsku. Průsečík je detekován při první změně znaménka funkce $f(t)$.

Alternativní algoritmy využívají různé metody odhadu intervalu paprsku, ve kterém je garantováno, že nedojde k průsečíku s tělesem, dělení prostoru, apod. Podrobnější informace jsou obsaženy v [Sher98] nebo [Cap99].

4 Implementace

Nyní se dostáváme ke konkrétním aplikačním řešením. V této kapitole budou představeny algoritmy konkrétněji tak, jak byly implementovány a testovány.

4.1 Datová struktura CSG stromu

Jedním z úkolů diplomové práce bylo navrhnout vnitřní datové struktury pro representaci implicitních funkcí a CSG stromů. Jak již bylo řečeno v kapitole 2.1, CSG strom obsahuje ve vnitřních uzlech logické operace a v listech CSG primitiva.

Datová struktura reprezentující implicitní funkci je navržena jako pointer jazyka C na funkci se třemi parametry, představujícími souřadnice bodu v prostoru. Návratovou hodnotou je funkční hodnota implicitní funkce v daném bodě.

Uzly CSG stromu nesou odkaz na přiřazené primitivum a každý z nich obsahuje trojici vektorů představující prostorové transformace posunu, rotace a změny měřítka. Tím je zajištěno, že uvedené transformace se mohou vztahovat jak na jednotlivá primitiva, tak na celé podstromy. Použitelnými primitivy jsou koule, válec, kužel, toroid a poloprostor. V každém uzlu je dále obsažena informace o požadované logické operaci mezi syny, tj. operaci sjednocení, průniku, rozdílu A-B, rozdílu B-A a symetrické diferenci (viz kapitola 2.1).

Navržená datová struktura umožňuje použití všech běžných modelovacích operací s CSG stromy v kombinaci s použitím implicitních funkcí jako CSG primitiv. Konkrétní implementace datových struktur je k nahlédnutí v příloze C diplomové práce.

4.2 Nalezení počátečního bodu

Počátečním bodem se rozumí jakýkoliv bod, který se nachází na povrchu generovaného objektu a který je zároveň prvním bodem výsledné polygonální sítě. Od tohoto bodu začíná proces polygonizace, který se dále šíří po celém povrchu funkce. Úspěšné nalezení takového bodu je důležitou součástí algoritmů uvedených v kapitole 3.2.1.

V této kapitole budou uvedeny dva algoritmy, použitelné pro nalezení počátečního bodu na povrchu objektu definovaného implicitní funkcí. Bude představen numerický postup, který cíleně směřuje k povrchu funkce a také alternativní metoda, využívající náhodného jevu.

4.2.1 Numerické vyhledávání

Algoritmus je založen na nalezení dvou různých bodů v prostoru, z nichž jeden leží uvnitř objektu a druhý vně. Řídíme-li se definicí implicitních objektů podle [Bloo95], tak pro vnitřní bod platí $f(\boldsymbol{p}_{in}) < 0$ a pro vnější bod $f(\boldsymbol{p}_{out}) > 0$. Uvažujeme-li *F-Rep*, jsou uvedené závislosti obrácené, tj. $f(\boldsymbol{p}_{in}) > 0$ a $f(\boldsymbol{p}_{out}) < 0$.

Obsahuje-li scéna více oddělených objektů, metoda nalezne nejbližší z nich vzhledem k náhodně zvolenému bodu $\boldsymbol{S}$ uvnitř definované oblasti Ω. (viz obrázek 4.1). Algoritmus zvolí bod uvnitř prohledávané oblasti a od tohoto bodu se přibližuje k povrchu funkce tak, že v každém kroku se přiblíží o δ ve všech třech osách.

Algoritmus: (4.1)

1. Náhodná volba souřadnic bodu $\boldsymbol{S}$ uvnitř definované oblasti Ω, $\boldsymbol{A}$, $\boldsymbol{B}$ jsou pomocné body, velikost proměnné δ je úměrná velikosti rastru (*cubesize*, viz obrázek 4.2).
2. Opakuj, dokud neplatí, že $f(\boldsymbol{A}) * f(\boldsymbol{B}) < 0$.
    a. $\boldsymbol{A}(x, y, z) = \boldsymbol{S}(x-\delta, y, z)$, $\boldsymbol{B}(x, y, z) = \boldsymbol{S}(x+\delta, y, z)$, test podmínky $f(\boldsymbol{A}) * f(\boldsymbol{B}) < 0$, pokud platí $|f(\boldsymbol{A})| < |f(\boldsymbol{B})|$ tak $\boldsymbol{S} = \boldsymbol{A}$, jinak $\boldsymbol{S} = \boldsymbol{B}$.
    b. $\boldsymbol{A}(x, y, z) = \boldsymbol{S}(x, y-\delta, z)$, $\boldsymbol{B}(x, y, z) = \boldsymbol{S}(x, y+\delta, z)$, test podmínky $f(\boldsymbol{A}) * f(\boldsymbol{B}) < 0$, pokud platí $|f(\boldsymbol{A})| < |f(\boldsymbol{B})|$ tak $\boldsymbol{S} = \boldsymbol{A}$, jinak $\boldsymbol{S} = \boldsymbol{B}$.
    c. $\boldsymbol{A}(x, y, z) = \boldsymbol{S}(x, y, z-\delta)$, $\boldsymbol{B}(x, y, z) = \boldsymbol{S}(x, y, z+\delta)$, test podmínky $f(\boldsymbol{A}) * f(\boldsymbol{B}) < 0$, pokud platí $|f(\boldsymbol{A})| < |f(\boldsymbol{B})|$ tak $\boldsymbol{S} = \boldsymbol{A}$, jinak $\boldsymbol{S} = \boldsymbol{B}$.
3. Platí-li $f(\boldsymbol{A}) < 0$, tak $\boldsymbol{p}_{in} = \boldsymbol{A}$, $\boldsymbol{p}_{out} = \boldsymbol{B}$, jinak $\boldsymbol{p}_{in} = \boldsymbol{B}$, $\boldsymbol{p}_{out} = \boldsymbol{A}$.
4. Přesné souřadnice počátečního bodu na povrchu funkce jsou vypočítány numerickou metodou půlení intervalu mezi body $\boldsymbol{p}_{in}$, $\boldsymbol{p}_{out}$.

4.2.2 Algoritmus Random search

Tato metoda je obdobou předchozího algoritmu. Nejdříve jsou hledány dva body, jeden ležící uvnitř objektu a druhý vně. Jak už název napovídá, hledání obou bodů je založeno na náhodné volbě souřadnic (viz [Bloo94]).

Algoritmus: (4.2)

1. Náhodná volba souřadnic bodu $\boldsymbol{A}$ uvnitř definované oblasti Ω, inicializuj proměnnou $range = \delta$.
2. Opakuj, dokud neplatí, že $f(\boldsymbol{A}) * f(\boldsymbol{B}) < 0$; Bod $\boldsymbol{B}$ je hledaný bod s opačným znaménkem funkční hodnoty než bod $\boldsymbol{A}$.
   a. Náhodná volba souřadnic bodu $\boldsymbol{B}$, $\boldsymbol{B} = \boldsymbol{B} * range$, $range = range * 1{,}0005$.
   b. Pokud je počet opakování větší než přijatelná hodnota, tak konec, bod nenalezen.
3. Pokud platí $f(\boldsymbol{A}) < 0$, tak $\boldsymbol{p}_{in} = \boldsymbol{A}$, $\boldsymbol{p}_{out} = \boldsymbol{B}$, jinak $\boldsymbol{p}_{in} = \boldsymbol{B}$, $\boldsymbol{p}_{out} = \boldsymbol{A}$.
4. Přesná souřadnice počátečního bodu na povrchu funkce je vypočítána numerickou metodou půlení intervalu mezi body $\boldsymbol{p}_{in}$, $\boldsymbol{p}_{out}$.

Poznámka:

- Proměnná *range* zajišťuje, že souřadnice bodu $\boldsymbol{B}$ budou voleny stále ve větší vzdálenosti od počátku souřadného systému.
- Hodnota konstanty 1,0005 je volena z důvodu pozvolného růstu proměnné *range*.

4.3 Exhaustive search

Tato metoda nevyžaduje pro svou činnost startovací bod, protože je založena na systematickém prohledávání předem definované oblasti v prostoru. V takové oblasti jsou nalezeny a následně polygonizovány všechny objekty, které s ní incidují. Další výhodou uvedeného algoritmu je snadná paralelizovatelnost, která je objasněna v kapitole 4.4.

Metoda využívá pro polygonizaci buněk (*krychlí*, *cubes*) algoritmus *Marching cubes*, který je podrobněji vysvětlen v kapitole 4.5.

Máme tedy k dispozici oblast prostoru, která je rozdělena v jednotlivých osách rastrem pevné velikosti (*cubesize*), jak je znázorněno na obrázku 4.2.

Algoritmus: (4.3)

1. Opakuj přes celou oblast Ω, indexy (i, j, k); i, j, k = 1, 2, … N-1.
   a. Urči funkční hodnotu $f(x_i, y_j, z_k)$ v aktuálním vrcholu mřížky (i, j, k).
   b. Urči funkční hodnotu $f(x_{i+1}, y_j, z_k)$ ve vrcholu mřížky ($i+1$, j, k). Pokud platí $f(x_i, y_j, z_k)^* f(x_{i+1}, y_j, z_k) < 0$, tak polygonizuj všechny buňky, které sdílí stejnou hranu mřížky, tj. hrana mezi vrcholy mřížky (i, j, k) a ($i+1$, j, k). Pro 3D případ to jsou čtyři okolní krychle[4].
   c. Urči funkční hodnotu $f(x_i, y_{j+1}, z_k)$ ve vrcholu mřížky (i, $j+1$, k). Pokud platí $f(x_i, y_j, z_k)^* f(x_i, y_{j+1}, z_k) < 0$, tak polygonizuj všechny buňky, které sdílí hranu mřížky mezi vrcholy (i, j, k) a (i, $j+1$, k).
   d. Urči funkční hodnotu $f(x_i, y_j, z_{k+1})$ ve vrcholu mřížky (i, j, $k+1$). Pokud platí $f(x_i, y_j, z_k)^* f(x_i, y_j, z_{k+1}) < 0$, tak polygonizuj všechny buňky, které sdílí hranu mřížky mezi vrcholy (i, j, k) a (i, j, $k+1$).

Poznámka:

Aby nedocházelo k opakovanému výpočtu funkčních hodnot v rozích buněk, popř. k opakované polygonizaci buňky se stejným indexem, jsou funkční hodnoty ukládány do tabulky využitím hashovací funkce a indexy již zpracovaných buněk jsou testovány pomocí bitové mapy. Tvar hashovací funkce byl převzat z [Bloo94].

4.4 Paralelní výpočet

Dalším úkolem diplomové práce bylo navrhnout paralelizaci zvoleného řešení. Metody založené na počátečním bodě by byly paralelizovatelné snadno pro implicitní funkce, které definují více oddělených objektů. Každý takový objekt by mohl být polygonizován samostatným výpočetním vláknem (*thread*) bez kritických sekcí v paralelním běhu. Urychlení výpočtu v takovém případě by bylo úměrné počtu dostupných procesorů a počtu objektů ve scéně. Jak už ale bylo uvedeno v kapitole 3.2.1, zjistit počet oddělených objektů a nalézt pro každý z nich odpovídající počáteční bod, je prozatím problém. Pro paralelní výpočet se tedy nabízí algoritmus metody *Exhaustive search*.

Do původního sekvenčního algoritmu 4.3 není potřeba příliš zasahovat. Před začátkem výpočtu je oblast Ω rozdělena na tolik částí, kolik je dostupných procesorů. Na obrázku 4.2 je

[4] Vznik případných duplicit je řešen ukládáním indexu již polygonizovaných buněk do bitové mapy.

znázorněno rozdělení oblasti pro 2 procesory. Část ***A*** je určena pro výpočet prvním procesorem a část ***B*** druhým procesorem. Rozdělení práce pro všechny procesory je tedy provedeno staticky před zahájením výpočtu.

Dynamické přidělování práce jednotlivým procesorům není nutné, neboť algoritmus „tráví“ nejvíce času na postupné kontrole znamének funkčních hodnot v rozích buněk (krychlí) a případné polygonizace buněk spotřebují relativně málo času. Tak je zajištěno, že procesory jsou vytíženy přibližně rovnoměrně i pokud je většina objektů ve scéně umístěna v oblasti jednoho procesoru. Tento předpoklad potvrzují dosažené výsledky v kapitole 5.2.3.

Předtím, než bude uveden algoritmus, který oblast rozděluje, definujme pojem *slab*.

Slab je plátek prostoru, který má ve dvou osách stejný rozměr jako oblast Ω a ve třetí ose je široký jako rozměr buněk (*cubesize*). Znázorněno na obrázku 4.2.

Počet slabů n_s, které mají být příslušným procesorem polygonizovány, je vypočítán podle následujícího algoritmu.

Algoritmus: (4.4)

Mějme oblast Ω rozdělenou na N slabů a k dispozici n_p volných procesorů.

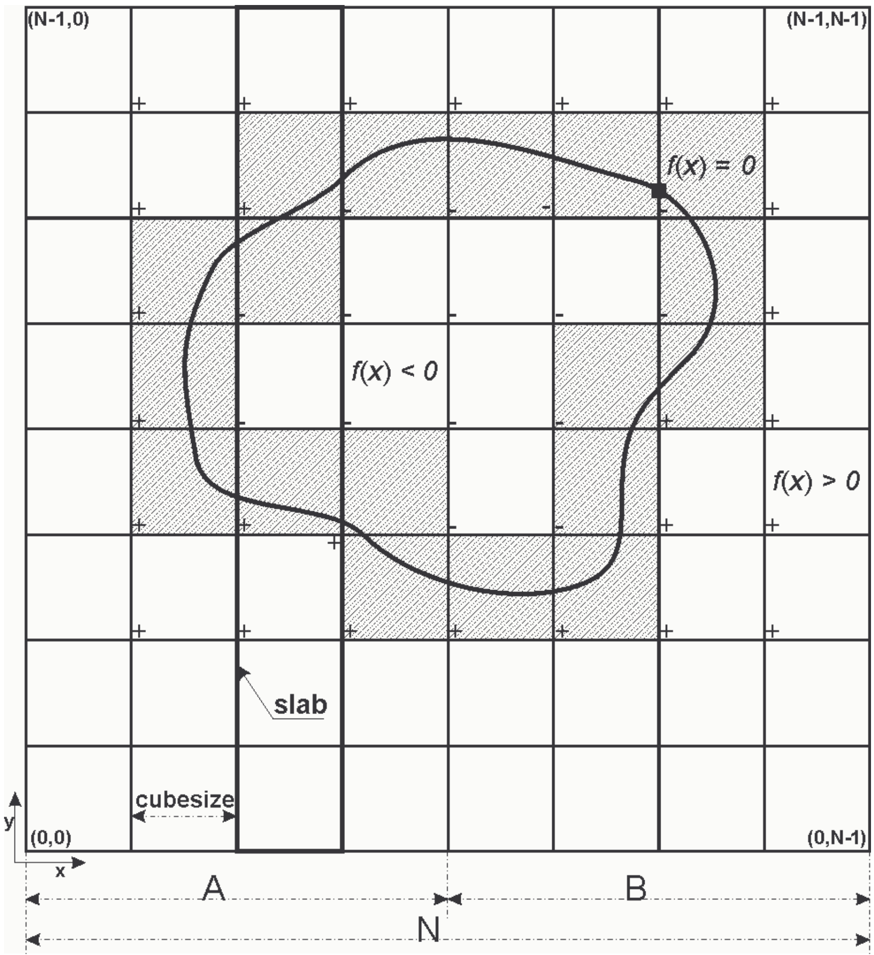
1. Vypočítejme $n_s = trunc\dfrac{N}{n_p}, n_z = \operatorname{mod}\dfrac{N}{n_p}$.
2. Prvních n_z vláken bude polygonizovat (n_s+1) slabů, ostatní vlákna budou polygonizovat n_s slabů.

Poznámka:

Operace *trunc* představuje převod reálného čísla na celé, odříznutím desetinné části.

Všechna výpočetní vlákna pracují nad stejnou datovou strukturou a využívají i stejné hashovací tabulky. Tím je zajištěno, že nedochází k opakované polygonizaci buněk na hranicích oblastí jednotlivých vláken a na konci výpočtu není nutné napojování více trojúhelníkových sítí.

Naproti tomu, práce vláken nad stejnými datovými strukturami je příčinou vzniku kritických sekcí, které jsou implementačně řešeny semafory jazyka Visual C++. Kritické sekce se při běhu programu na dvou procesorech příliš neprojevují (viz kapitola 5.2.3) a jejich vliv na urychlení a efektivitu paralelního algoritmu při běhu programu na víceprocesorovém stroji nebyl prokázán, protože příslušné testy nebyly provedeny. Důvodem byla nedostupnost takového počítače.



Obrázek 4.2: Oblast prostoru rozdělená rastrem. Buňky, které jsou protnuty povrchem objektu, jsou zvýrazněny.

4.5 Algoritmus Marching cubes

Marching cubes je nejznámějším a současně nejpoužívanějším algoritmem pro převod izoploch v objemové reprezentaci[5] na trojúhelníkové sítě. Použitelnost metody je nadále rozšířená o polygonizaci určité oblasti ohodnocené implicitní funkcí.

Výstupem algoritmu je izoplocha ve formě trojúhelníkové sítě, která je zobrazitelná klasickými metodami počítačové grafiky s využitím existujících grafických akcelerátorů.

Metoda produkuje velké množství dat (z jedné buňky mohou vzniknout až 4 trojúhelníky), která jsou charakteristická pravidelnou mřížkou a množstvím „úzkých“ trojúhelníků.

V následujících podkapitolách bude popsán postup, kterým algoritmus polygonizuje povrch implicitního tělesa od počátečního bodu až k finálnímu výsledku.

4.5.1 Detekce buněk protnutých povrchem funkce

V tomto odstavci bude zmíněn princip, kterým metoda *Marching cubes* rozpoznává buňky, které jsou protnuty povrchem objektu a jsou určeny pro polygonizaci, převzato z [Bloo94].

Detekované buňky jsou ukládány do tzv. *Seznamu aktivních buněk*. Tento seznam obsahuje jen ty buňky, které budou v následujících krocích polygonizovány.

Pokud byl úspěšně nalezen počáteční bod ležící na povrchu objektu (viz kapitola 3.2.1), je možné zahájit polygonizaci funkce následujícím způsobem.

Algoritmus: (4.5)

1. Umísti střed první buňky (krychle, *cube*) s indexy (*0*, *0*, *0*) na počáteční bod.
2. Vypočítej funkční hodnoty v rozích první krychle a vlož ji do seznamu aktivních buněk.
3. Opakuj, dokud není seznam aktivních buněk prázdný.
    a. Polygonizuj aktuální buňku a vyřaď ji ze seznamu aktivních buněk.
    b. Vypočítej funkční hodnoty v rozích sousedních[6] buněk a vlož do seznamu aktivních buněk ty, u kterých se liší hodnoty znamének.
    c. Vyber další aktuální buňku ze seznamu aktivních buněk.

Poznámka:

Aby nedocházelo k opakovanému výpočtu funkčních hodnot ve vrcholech mřížky, popř. k opakované polygonizaci buňky se stejným indexem, jsou funkční hodnoty a indexy již zpracovaných buněk ukládány stejným způsobem jako v odstavci 4.3.

---

[5] např. tomografické snímky (CT), magnetická rezonance (MR)

[6] Sousední buňky jsou uvažovány pro 2D případ ve smyslu čtyř-okolí (viz obrázek 4.2), tzn. šesti-okolí ve 3D případě.

4.5.2 Polygonizace buňky

Máme-li připravenu ke zpracování polygonizační buňku (krychli) s ohodnocenými rohy, je její polygonizace prováděna podle následujících kroků.

Algoritmus: (4.6)

1. Sestavení indexu do tabulky možných polarit rohů.
2. Nalezení seznamu hran, které jsou protnuty povrchem objektu.
3. Výpočet souřadnic vrcholů trojúhelníků na protnutých hranách.
4. Výpočet normál ve vrcholech trojúhelníků.

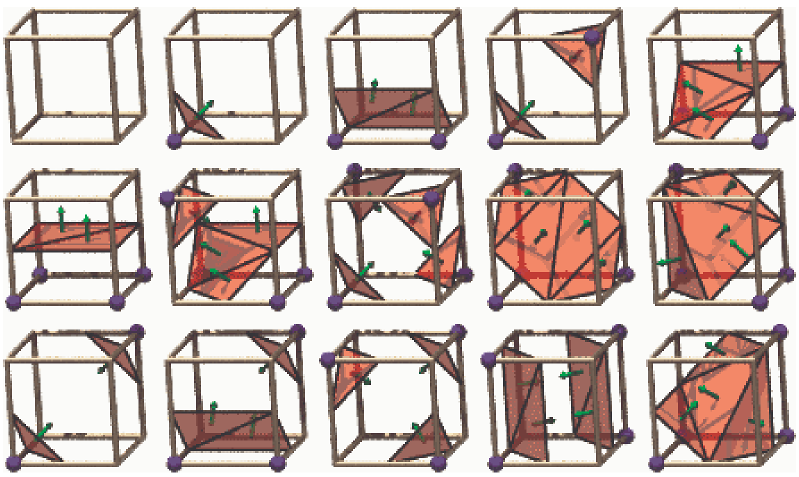
4.5.3 Sestavení indexu a nalezení protnutých hran

Tabulka možných polarit rohů krychle obsahuje 256 záznamů (řádků), kde každý z nich obsahuje seznam hran protnutých povrchem. Díky tomu, že krychle je značně symetrické geometrické těleso, je z těchto 256 možných případů jen 15 základních, které jsou znázorněny na obrázku 4.3. Ostatní případy vzniknou rotacemi a inverzí nul a jedniček v indexu.

Polaritu každého rohu lze vyjádřit jedním bitem, který nabývá hodnoty

- „*1*“, pokud platí $f(\boldsymbol{c}) > 0$,
- „*0*“, pokud platí $f(\boldsymbol{c}) < 0$, kde $\boldsymbol{c}$ je pozice příslušného rohu.

Vzniklých 8 binárních hodnot, představuje konkrétní index do tabulky polarit.



Obrázek 4.3: Patnáct základních případů polarit krychle pro metodu *Marching cubes*.

4.5.4 Výpočet souřadnic vrcholů trojúhelníků

Při použití metody *Marching cubes* v aplikacích s naměřenými objemovými daty, je nutné provádět výpočet souřadnice vrcholu lineární interpolací hodnot v rozích buněk.

Stejný postup je použitelný i zde, ale aproximace povrchu implicitní funkce tím ztrácí na přesnosti. Lepší alternativou je využití vlastností implicitních funkcí, které ohodnocují prostor ve všech bodech, tzn. že souřadnice vrcholů lze vypočítat s určitou vyšší přesností ($\pm\,\varepsilon$).

Z tabulky polarity buněk máme k dispozici seznam protnutých hran. Přesnou souřadnici vrcholu, v rámci každé protnuté hrany, pak můžeme získat numerickým výpočtem, který konverguje k povrchu objektu. Možnými numerickými metodami je binární dělení (*binary subdivision*), popř. *regula falsi*. V tomto případě je použit algoritmus binárního dělení, protože je v případě některých složitých implicitních funkcí, numericky stabilnější (viz [Bloo94]).

4.5.5 Výpočet normál ve vrcholech

Výpočet jednotkového normálového vektoru ve vrcholech je prováděn podle vzorce:

$$n = {}^{\nabla f(v)}\!\Big/\!_{\|\nabla f(v)\|}, \tag{4.1}$$

kde v je příslušný vrchol a ∇f je gradient funkce.

Výpočet gradientu může být prováděn buď symetrickou nebo asymetrickou diferencí. Protože implicitní funkce ohodnocuje prostor v každém bodě, nemusí být užíváno jen symetrického vztahu pro diferenci, jako je tomu v případě navzorkovaných volumetrických dat, viz [Zar98a], kde se asymetrická diference nepoužívá z důvodů větší nepřesnosti.

Symetrická diference: (4.2)

$$\nabla f(x,y,z) = \begin{bmatrix} f(x+\delta,y,z) - f(x-\delta,y,z) \\ f(x,y+\delta,z) - f(x,y-\delta,z) \\ f(x,y,z+\delta) - f(x,y,z-\delta) \end{bmatrix}$$

Asymetrická diference: (4.3)

$$\nabla f(x,y,z) = \begin{bmatrix} f(x+\delta,y,z) - f(x,y,z) \\ f(x,y+\delta,z) - f(x,y,z) \\ f(x,y,z+\delta) - f(x,y,z) \end{bmatrix},$$

kde hodnota δ je úměrná rozměru buňky (*cubesize*) a platí relace δ << *cubesize*.

4.6 Algoritmus Marching triangles

Tato metoda patří mezi nové metody generující polygonální síť z implicitních funkcí, a proto jí budeme věnovat větší pozornost. Algoritmus *Marching triangles* přistupuje k polygonizaci implicitní funkce zcela odlišným způsobem a výsledná polygonální síť se vyznačuje trojúhelníky s průměrnou velikostí minimálního úhlu 60°. Jedná se tedy o kvalitnější trojúhelníkovou síť, než jaká vzniká metodou *Marching cubes*. Konkrétní výsledky a porovnání jsou uvedeny v kapitole 5.2.1.

Algoritmus vyžaduje pro své výpočty startovací bod ležící na povrchu generovaného objektu, podobně jako předchozí metoda.

Základ algoritmu, který byl uveden v [Hart98], je popsán v odstavci 4.6.3. Urychlení původního algoritmu je objasněno v odstavcích 4.6.5 a 4.6.6. Nejdříve však bude uvedena procedura *Surface point*, na kterou je v algoritmu 4.8 odkazováno a původní datové struktury.

4.6.1 Procedura Surface point

Procedura *Surface point* slouží k výpočtu přesné pozice bodu $\boldsymbol{p}$ na povrchu objektu definovaného implicitní funkcí, k výpočtu jeho normálového vektoru $\boldsymbol{n}$ a dvou tečných vektorů $\boldsymbol{t_1}$ a $\boldsymbol{t_2}$. Trojice ortogonálních vektorů ($\boldsymbol{t_1}$, $\boldsymbol{t_2}$, $\boldsymbol{n}$) představuje lokální souřadnicový systém bodu $\boldsymbol{p}$.

Mějme implicitní funkci $\Phi : f(x) = 0$, pro kterou existuje nenulový gradient ∇f v každém uvažovaném bodě, a bod $\boldsymbol{q}$ ležící v blízkosti povrchu objektu.

Algoritmus: (4.7)

1. Inicializuj pomocný bod $\boldsymbol{u_0} = \boldsymbol{q}$.
2. Opakuj, dokud není $f(u_{k+1}) < \varepsilon$ nebo $f(u_k) \cdot f(u_{k+1}) < 0$.
   - $$u_{k+1} = u_k - \frac{f(u_k)}{\nabla f(u_k)^2} \nabla f(u_k). \qquad (4.4)$$
   - $u_k = u_{k+1}$.
3. Pokud platí, že $f(u_k) \cdot f(u_{k+1}) < 0$, tak použij metodu půlení intervalu dokud neplatí $f(u_{k+1}) < \varepsilon$.
4. Bod na povrchu $\boldsymbol{p} = \boldsymbol{u_{k+1}}$.

5. Jednotkový normálový vektor bodu ***p***,

$$n = \nabla f(p) \big/ \left\| \nabla f(p) \right\| .$$

6. Tečné vektory $\boldsymbol{t_1}$ a $\boldsymbol{t_2}$ lze určit:
   - pokud $\boldsymbol{n_x} > 0,5$ nebo $\boldsymbol{n_y} > 0,5$, potom $\boldsymbol{t_1} = (n_y, -n_x, 0)$, jinak $\boldsymbol{t_1} = (-n_z, 0, n_x)$,
   - $t_2 = n \times t_1$, kde $\boldsymbol{n} = (n_x, n_y, n_z)$.

Poznámka:

- Gradient funkce je počítán podle vzorce (4.3).
- V algoritmu je kombinována Newtonova metoda výpočtu kořene implicitní rovnice s metodou půlení intervalu z důvodu vyšší numerické stability v bodech C^1 nespojitosti implicitní funkce.

4.6.2 Datové struktury

V tomto odstavci budou představeny základní datové struktury pro *bod*, *trojúhelník* a *front polygon*, v původní podobě tak, jak byly popsány v [Hart98].

Pro každý bod je nutné uchovávat jeho souřadnice v prostoru, normálový vektor a dva tečné vektory a trojici příznaků platnosti. Jednotlivé trojúhelníky jsou chápány klasickým způsobem, tj. jako trojice indexů do pole vrcholů. Poněkud neobvyklou datovou strukturou je *Front polygon*, který představuje seznam vrcholů, které se aktuálně podílejí na polygonizaci povrchu, tj. následující triangulace bude prováděna pouze *před* body *Front* polygonu. Tento seznam je realizován jako pole indexů do pole bodů a má jednu základní vlastnost, musí zachovávat pro každý bod relace levý soused, pravý soused. Front polygon si je možné představit jako uzavřenou křivku, ležící na povrchu objektu. Protože je křivka uzavřená, je také datová struktura cyklická, tj. pravý soused pro poslední prvek seznamu je první prvek a naopak. Konkrétní implementace datových struktur je uvedena v příloze C diplomové práce.

4.6.3 Originální algoritmus Marching triangles

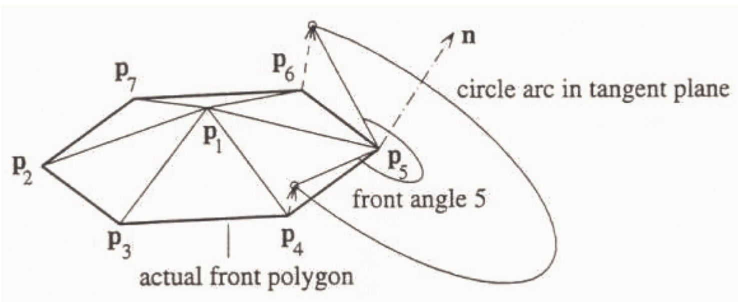
V této kapitole je představen původní algoritmus, který byl uveden v [Hart98].

Algoritmus: (4.8)

1. Mějme startovacího bod $\boldsymbol{q_1}$ v blízkosti povrchu objektu. Pro bod $\boldsymbol{q_1}$ je volána procedura *Surface point* (popsána v odstavci 4.6.1), která vypočítá bod na povrchu $\boldsymbol{p_1}$, jeho normálu $\boldsymbol{n_1}$ a dva tečné vektory $\boldsymbol{t_{11}}$ a $\boldsymbol{t_{12}}$. Bod $\boldsymbol{p_1}$ je pak ohraničen šestiúhelníkem[7] v jeho tečné rovině - body $\boldsymbol{q_2}\ldots\boldsymbol{q_7}$, podle vzorce:

$$q_{i+2} = p_1 + \delta_t \cos(i\pi/3)t_{11} + \delta_t \sin(i\pi/3)t_{12}, i = 0,...,5\,. \tag{4.5}$$

   Pro každý z bodů $\boldsymbol{q_2}\ldots\boldsymbol{q_7}$ je volána procedura *Surface point*, která vypočítá odpovídající body $\boldsymbol{p_2}\ldots\boldsymbol{p_7}$, které leží na povrchu objektu. Trojúhelníky na povrchu hexagonu jsou prvními šesti trojúhelníky triangulace ($\boldsymbol{p_1}$, $\boldsymbol{p_2}$, $\boldsymbol{p_3}$), ($\boldsymbol{p_1}$, $\boldsymbol{p_3}$, $\boldsymbol{p_4}$), ($\boldsymbol{p_1}$, $\boldsymbol{p_4}$, $\boldsymbol{p_5}$), ($\boldsymbol{p_1}$, $\boldsymbol{p_5}$, $\boldsymbol{p_6}$), ($\boldsymbol{p_1}$, $\boldsymbol{p_6}$, $\boldsymbol{p_7}$), ($\boldsymbol{p_1}$, $\boldsymbol{p_7}$, $\boldsymbol{p_2}$). Seřazené pole bodů $\boldsymbol{p_2}\ldots\boldsymbol{p_7}$ se nazývá *Actual front polygon* Π_0. První šestiúhelník je znázorněn na obrázku 4.4.



Obrázek 4.4: První šestiúhelník se zvýrazněným front úhlem u bodu $\boldsymbol{p_5}$.

2. Pro každý bod aktuálního front polygonu Π_0 určíme úhel oblasti, která se bude triangulovat. Tyto úhly nazveme *Front angles* (znázorněn na obrázku 4.4 u bodu $\boldsymbol{p_5}$). Dále určíme sousedy bodu $\boldsymbol{p_{0i}}$.
   - Nechť bod $\boldsymbol{v_1}$ je levým sousedem (na obrázku 4.4 je levým sousedem bodu $\boldsymbol{p_5}$ bod $\boldsymbol{p_4}$), který má v lokálním souřadnicovém systému bodu $\boldsymbol{p_{0i}}$ ($\boldsymbol{t_{i1}}$, $\boldsymbol{t_{i2}}$, $\boldsymbol{n_i}$) souřadnice (ξ_1, η_1, ζ_1) a
   - bod $\boldsymbol{v_2}$ pravým sousedem se souřadnicemi (ξ_2, η_2, ζ_2).

[7] Tvar šestiúhelníka je volen proto, aby vzniklá polygonální síť byla tvořena trojúhelníky, jejichž průměrný vnitřní úhel je 60°, tj. π/3. Takové trojúhelníky jsou přibližně rovnostranné a výsledná síť je kvalitnější.

Potom ω_1 = polární úhel souřadnic (ξ_1, η_1) a ω_2 = polární úhel souřadnic (ξ_2, η_2).

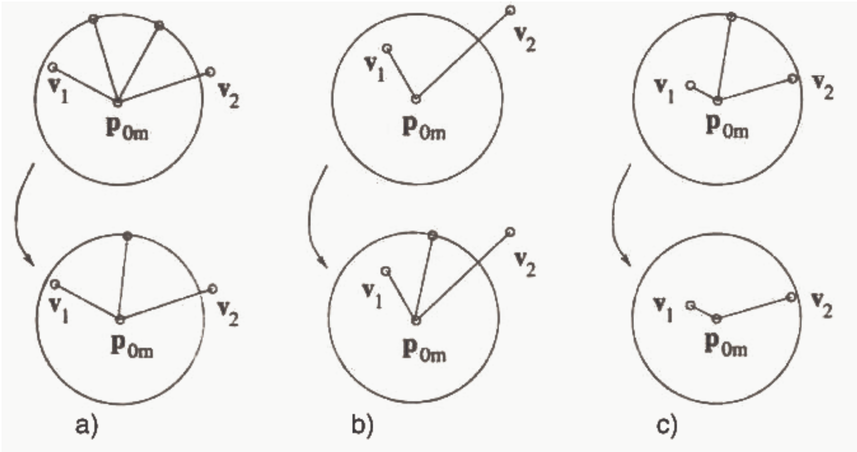
Výsledný úhel je určen jako:

Front angle = $\omega_1 - \omega_2$, pokud $\omega_1 > \omega_2$, jinak *Front angle* = $\omega_1 - \omega_2 + 2\pi$.

3. Test který zabraňuje překrývání nových trojúhelníků s již triangulovanou částí.
   - Testujeme vzdálenost dvojic bodů aktuálního front polygonu Π_0.
   - Testujeme vzdálenost bodů aktuálního front polygonu Π_0 s body ostatních front polygonů Π_k, kde $k > 0$.

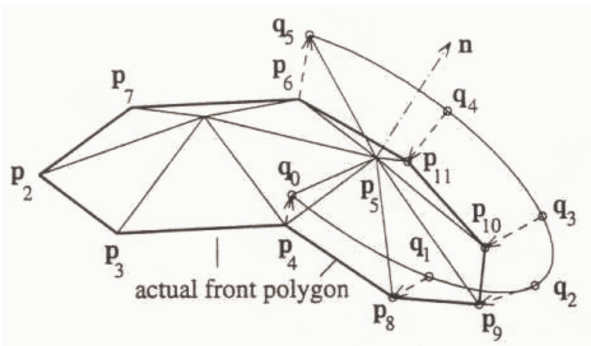
   Poznámka:
   - Před aplikací obou testů by měli být triangulovány všechny body jejichž *front angle* je menší než 60°.
   - Těmito testy se budeme zabývat podrobněji v následujících odstavcích, protože jsou kritickým místem, kde algoritmus „tráví“ nejvíce času.

4. Vlastní triangulace. Nechť $\boldsymbol{p_{0m}}$ je bod aktuálního front polygonu Π_0 s minimálním front úhlem ω. Triangulace okolí bodu $\boldsymbol{p_{0m}}$ se skládá z následujících kroků.
   - Určení levého $\boldsymbol{v_1}$ a pravého $\boldsymbol{v_2}$ souseda bodu $\boldsymbol{p_{0m}}$.
   - Určení počtu trojúhelníků n_t, které budou generovány, podle vzorce:
     - $n_t = trunc(3\omega/\pi) + 1, \Delta\omega = \omega/n_t$. (4.6)
   - Oprava hodnoty Δω pro extrémní případy, které jsou znázorněny na obrázku 4.5, podle následujících pravidel.
     - Pokud je $\Delta\omega < 0,8$ a $n_t > 1$, potom $n_t = n_t - 1, \Delta\omega = \omega/n_t$. Obrázek 4.5a.
     - Pokud je $n_t = 1$ a $\Delta\omega > 0.8$ a $\| v_1 - v_2 \| > 1.2\delta_t$, potom $n_t = 2, \Delta\omega = \Delta\omega/2$. Obrázek 4.5b.
     - Pokud je $\omega < 3$ a ($\| v_1 - p_{0m} \| \le 0.5\delta_t$ nebo $\| v_2 - p_{0m} \| \le 0.5\delta_t$), potom $n_t = 1$. Obrázek 4.5c.



Obrázek 4.5: Oprava hodnot $\Delta\omega$ pro extrémní případy.

- Generování nových trojúhelníků v okolí bodu $\boldsymbol{p_{0m}}$.
    - Pokud je $n_t = 1$, je generován pouze jeden nový trojúhelník ($\boldsymbol{v_1}$, $\boldsymbol{v_2}$, $\boldsymbol{p_{0m}}$).
    - V ostatních případech nechť $\boldsymbol{q_0}$ a $\boldsymbol{q_{nt}}$ jsou ortogonální projekce bodů $\boldsymbol{v_1}$, $\boldsymbol{v_2}$ do tečné roviny bodu $\boldsymbol{p_{0m}}$ a nechť $\boldsymbol{q_i}$, $i = 1,\ldots, n_t-1$, jsou body v tečné rovině vzniklé rotací bodu $\boldsymbol{b}$ kolem normálového vektoru $\boldsymbol{n}$ bodu $\boldsymbol{p_{0m}}$ o úhel $\Delta\omega$ (znázorněno na obrázku 4.6). Souřadnice pomocného bodu $\boldsymbol{b}$ lze vypočítat podle vzorce: $b = p_{0m} + \delta_t (q_0 - p_{0m}) / \| q_0 - p_{0m} \|$. (4.7)



Obrázek 4.6: Triangulace okolí bodu $\boldsymbol{p_5}$.

Aplikací procedury *Surface point* na body $\boldsymbol{q_i}$, $i = 1,\ldots, n_t-1$, získáme odpovídající nové body $\boldsymbol{p_{N+i}}$, $i = 1,\ldots, n_t-1$, kde N je celkový počet již dříve vygenerovaných bodů (znázorněno na obrázku 4.6).
Z nových bodů vzniknou následující trojúhelníky:

($\boldsymbol{v_1}$, $\boldsymbol{p_{N+1}}$, $\boldsymbol{p_{0m}}$), ($\boldsymbol{p_{N+1}}$, $\boldsymbol{p_{N+2}}$, $\boldsymbol{p_{0m}}$),…, ($\boldsymbol{p_{N+nt-1}}$, $\boldsymbol{v_2}$, $\boldsymbol{p_{0m}}$).

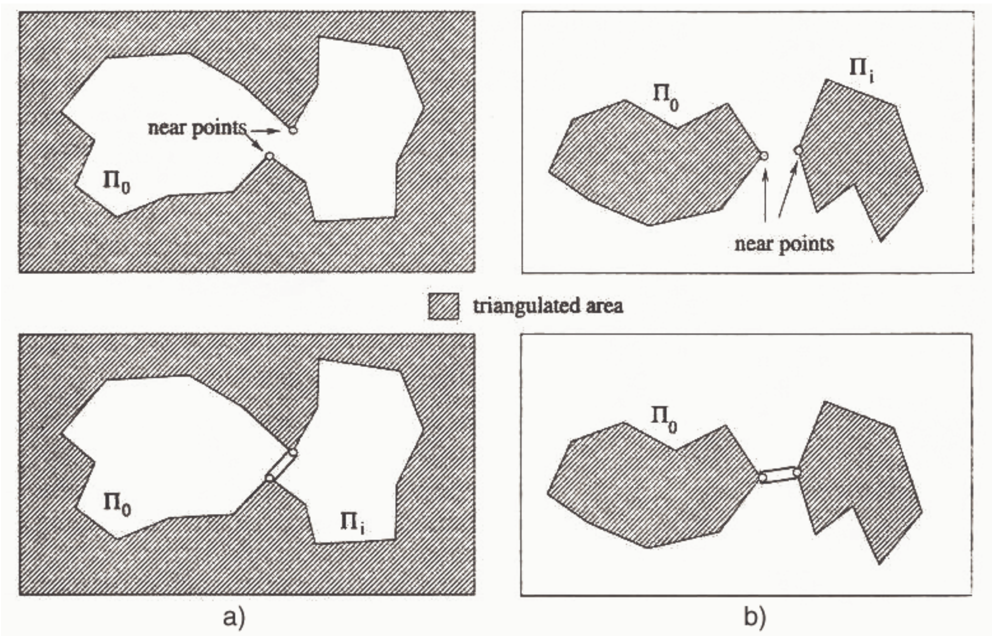
5. Aktualizace aktuálního front polygonu.
   - Vyřazení bodu $\boldsymbol{p_{0m}}$ z aktuálního front polygonu.
   - Pokud je $n_t > 1$, vlož do aktuálního front polygonu body $\boldsymbol{p_{N+1}}, \dots, \boldsymbol{p_{N+nt-1}}$.
   - Nastav příznak *angle_changed* u bodů $\boldsymbol{v_1}$, $\boldsymbol{v_2}$, $\boldsymbol{p_{N+1}}, \dots, \boldsymbol{p_{N+nt-1}}$, z důvodu aktualizace jejich *front angle*.

6. Opakuj kroky 2-5, dokud aktuální front polygon Π_0 neobsahuje poslední 3 body, které společně tvoří trojúhelník. Pokud existuje nějaký další (neprázdný) front polygon, stane se novým aktuálním front polygonem Π_0 a kroky 2-5 se opakují. Pokud již není k dispozici žádný jiný front polygon, končí proces polygonizace.

4.6.4 Testy vzdálenosti

V tomto odstavci budou vysvětleny původní testy vzdálenosti bodů představené v [Hart98]. Testy vzdálenosti (algoritmus 4.8, krok 3) slouží k detekci překrývání nových trojúhelníků s již triangulovanou částí a patří mezi časově nejnáročnější operace originálního algoritmu. Používány jsou dva následující testy:

1. Test vzdálenosti dvojic bodů aktuálního front polygonu Π_0.
2. Test vzdálenosti bodů aktuálního front polygonu Π_0 s body ostatních front polygonů Π_k, kde $k > 0$.



Obrázek 4.7: Test vzdálenosti bodů aktuálního front polygonu – rozpojení a), test vzdálenosti bodů aktuálního front polygonu s jiným front polygonem – spojení b).

1. test

Testujeme vzdálenost dvojic bodů aktuálního front polygonu $\boldsymbol{p_{0i}}, \boldsymbol{p_{0j}} \in \Pi_0$. Pokud existuje bod $\boldsymbol{p_{0j}}$, který není pro bod $\boldsymbol{p_{0i}}$ přímým sousedem ani sousedem souseda a platí:

$\| p_{0i} - p_{0j} \| < \delta_t$, kde δ_t představuje průměrnou délku hran vznikajících trojúhelníků.

V takovém případě je aktuální front polygon s celkovým počtem N bodů rozdělen na dvě části (viz obrázek 4.7a).

- Nový aktuální front polygon ($\boldsymbol{p_{01}}, \ldots, \boldsymbol{p_{0i}}, \boldsymbol{p_{0j}}, \ldots, \boldsymbol{p_{0N}}$) s novým počtem bodů $N_{new} = N - (j-i-1)$.
- Zbývající front polygon ($\boldsymbol{p_{0i}}, \ldots, \boldsymbol{p_{0j}}$) s počtem bodů $N_{new} = j-i+1$.

Body $\boldsymbol{p_{0i}}, \boldsymbol{p_{0j}}$ nesmí být zúčastněny dalších testů vzdálenosti, musí být nastaven jejich příznak *border_point*.

2. test

Testujeme vzdálenost bodů aktuálního front polygonu Π_0 s body ostatních front polygonů Π_k, $k > 0$. Pokud existují body $\boldsymbol{p_{0i}} \in \Pi_0$ a $\boldsymbol{p_{mj}} \in \Pi_m$, pro které platí:

$\| p_{0i} - p_{mj} \| < \delta_t$,

potom se původní aktuální front polygon ($\boldsymbol{p_{01}}, \ldots, \boldsymbol{p_{0N}}$) a jiný front polygon ($\boldsymbol{p_{m1}}, \ldots, \boldsymbol{p_{mM}}$) spojí do nového aktuálního front polygonu (znázorněno na obrázku 4.7b):

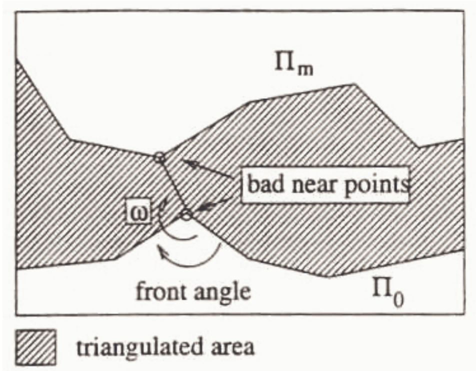
($\boldsymbol{p_{01}}, \ldots, \boldsymbol{p_{0i}}, \boldsymbol{p_{mj}}, \ldots, \boldsymbol{p_{mM}}, \boldsymbol{p_{m1}}, \ldots, \boldsymbol{p_{mj}}, \boldsymbol{p_{0i}}, \ldots, \boldsymbol{p_{0N}}$) s počtem bodů $N_{new} = N + M + 2$.

Body $\boldsymbol{p_{0i}}$ a $\boldsymbol{p_{mj}}$ jsou vloženy dvakrát. Před dalším výpočtem je nutné přepočítat *front angle* pro první výskyt těchto bodů a jejich okolí triangulovat. Tím dojde k odstranění jejich prvního výskytu v aktuálním front polygonu.

Body $\boldsymbol{p_{0i}}, \boldsymbol{p_{mj}}$ nesmí být zúčastněny dalších testů vzdálenosti, musí být nastaven jejich příznak *border_point*.

Poznámka:

Předtím než dojde ke spojení obou front polygonů, je nutné přezkoumat dvojici bodů $\boldsymbol{p_{0i}}, \boldsymbol{p_{mj}}$, zda se nejedná o blízké body přes již triangulovanou oblast. Takové body nazveme *Bad near points*. Situace je znázorněna na obrázku 4.8.



Obrázek 4.8: Blízké body přes již triangulovanou oblast.

4.6.5 Urychlení původního algoritmu

V původním algoritmu popsaném v kapitole 4.6.3 je několik míst, která nejsou realizována efektivně. Z těchto míst jsou časově nejnáročnější právě oba testy vzdáleností.

První z testů, který porovnává vzdálenosti bodů aktuálního front polygonu je složitosti $O(\frac{n(n+1)}{2})$, kde $n = m - 3$, je-li m aktuální počet bodů v aktuálním front polygonu. Konstanta *3* vychází z požadavku, že testované body nejsou sousedy ani sousedy sousedů.

Druhý test, který porovnává vzdálenosti bodů aktuálního front polygonu se všemi ostatními front polygony je složitosti $O(n \cdot m)$, kde n je počet bodů aktuálního front polygonu a m je celkový počet bodů ve všech ostatních front polygonech.

Oba testy jsou vykonávány v každém kroku algoritmu 4.8, tj. výsledná složitost je dále násobena počtem průchodů algoritmem.

Množství bodů ve front polygonech je úměrné tvaru objektu a velikosti δ_t (průměrná délka hran trojúhelníků) a platí, že $n_{pf} << n_p$, kde n_{pf} je průměrný počet bodů ve front polygonu a n_p je celkový počet generovaných bodů. I přes uvedenou relaci, je časová složitost algoritmu příliš vysoká.

Abychom odstranili redundantní kontroly vzdáleností v bodech, ve kterých nemůže dojít k rozdělení, resp. spojení front polygonů, provedl jsem následující změnu v algoritmu. Test vzdálenosti je prováděn přímo ve 4. kroku algoritmu 4.8, tj. pouze pro nově vložené body do aktuálního front polygonu. V jiných bodech ke změně dojít nemůže.

Pak je pro každý nově vložený bod aktuálního front polygonu provedena:

- kontrola vzdálenosti s body aktuálního front polygonu se složitostí $O(n)$, kde n je počet bodů aktuálního front polygonu,
- kontrola vzdálenosti s body jiných front polygonů se složitostí $O(m)$, kde m je celkový počet bodů ve všech ostatních front polygonech.

Další místo v algoritmu, které není prováděno efektivně, je výpočet *front angles* u bodů aktuálního front polygonu, které mají nastaven příznak *angle_changed* = *TRUE* (2. krok algoritmu 4.8). Tento krok je složitosti $O(n)$, kde n je počet bodů aktuálního front polygonu, protože je nutné procházet všechny body.

Z tohoto důvodu byl výpočet *front_angles* také přemístěn do kroku 4 a je prováděn pouze pro nově přidané body a jejich sousedy.

K dalšímu zefektivnění metody *Marching triangles* vedla změna datové struktury reprezentující *front polygon*. Protože je v každém *front polygonu* nutné zachovávat relace levý soused, pravý soused, znamenalo každé vložení/odstranění bodu do/ze středu pole posun a přeindexování celého zbytku tabulky. Datová struktura pro *front polygon* byla proto změněna z tabulky na oboustranně zřetězený seznam, ve kterém zůstávají zachovány obě požadované relace a vložení bodu do středu seznamu znamená pouhé přesměrování sousedních ukazatelů.

Výsledky porovnání naměřených časů původního algoritmu a upravené verze jsou uvedeny v kapitole 5.2.2.

4.6.6 Jiný způsob urychlení výpočtu

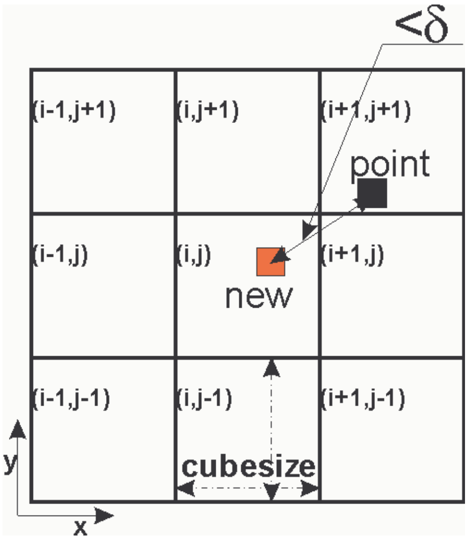
Upravená verze algoritmu *Marching triangles* uvedená v předcházejícím odstavci je stále málo efektivní. Příčinou jsou opět zmiňované testy vzdálenosti. Přestože jsou testovány pouze nově přidávané body, stále jsou prováděny testy se všemi body front polygonů, z nichž většina je od těchto nových příliš vzdálena a k rozdělení/spojení front polygonů dojít nemůže.

K vyřešení tohoto problému jsem použil *HASH* funkci, která každý přidávaný bod zařadí zároveň do tabulky na místo, které odpovídá podoblasti (*cell*, *cube*), ve které se příslušný bod nachází. Každá podoblast tak obsahuje oboustranně zřetězený seznam bodů (podobná datová struktura jako u front polygonu), které v ní leží. Počet těchto bodů je v každém záznamu *HASH* tabulky mnohonásobně menší než počet bodů ve front polygonu a testovány jsou opravdu jen nejbližší body, ve kterých může dojít k rozdělení/spojení.

Ve společné *HASH* tabulce jsou uloženy body ze všech front polygonů a každý z těchto bodů s sebou nese informaci, ke kterému front polygonu náleží. Kontrola vzdálenosti potom není vykonávána zvlášť pro aktuální front polygon a zvlášť pro ostatní polygony, ale je provedena v jednom kroku a to jen pro body, které leží ve stejné oblasti (nebo jedné ze sousedních) jako nově přidávaný bod. Na obrázku 4.9 je znázorněno osmi-okolí oblasti, ve které leží testovaný bod.

Poznámka:

Pro názornost je uveden dvourozměrný případ, v trojrozměrné reprezentaci se jedná o 26-ti okolí.



Obrázek 4.9: Test vzdálenosti nově přidávaného bodu jen s body z nejbližšího okolí.

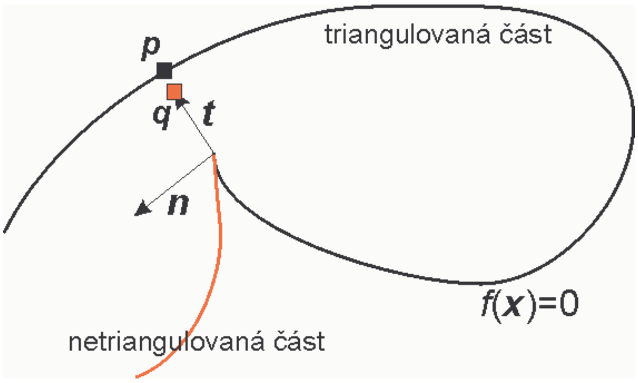
Porovnání rychlosti jednotlivých implementací metody *Marching triangles* je provedeno v kapitole 5.2.2. Konkrétní tvar *HASH* funkce je uveden v příloze *C* diplomové práce.

4.6.7 Omezení metody

Algoritmus metody *Marching triangles* určuje směr generování povrchu objektu na základě výpočtu správné souřadnice bodu na povrchu a také v závislosti na výpočtu normálového vektoru. Pokud dojde k chybnému výpočtu normály v bodě, dojde i k chybnému výpočtu tečných vektorů $\boldsymbol{t}_1$ a $\boldsymbol{t}_2$, což by mohlo mít za následek obrácení směru polygonizace zpět do triangulované části.

Popisovaná situace může nastat v bodech C^1 nespojitosti, tj. u funkcí, které obsahují ostré hrany. V těchto místech se mění skokově směr normálového vektoru, jehož výpočet je zatížen chybou, jejíž velikost je závislá na volbě δ (viz kapitola 4.5.5).

Jedna z možných variant chybného výpočtu normály a tím i celé polygonizace, je znázorněna na obrázku 4.10.



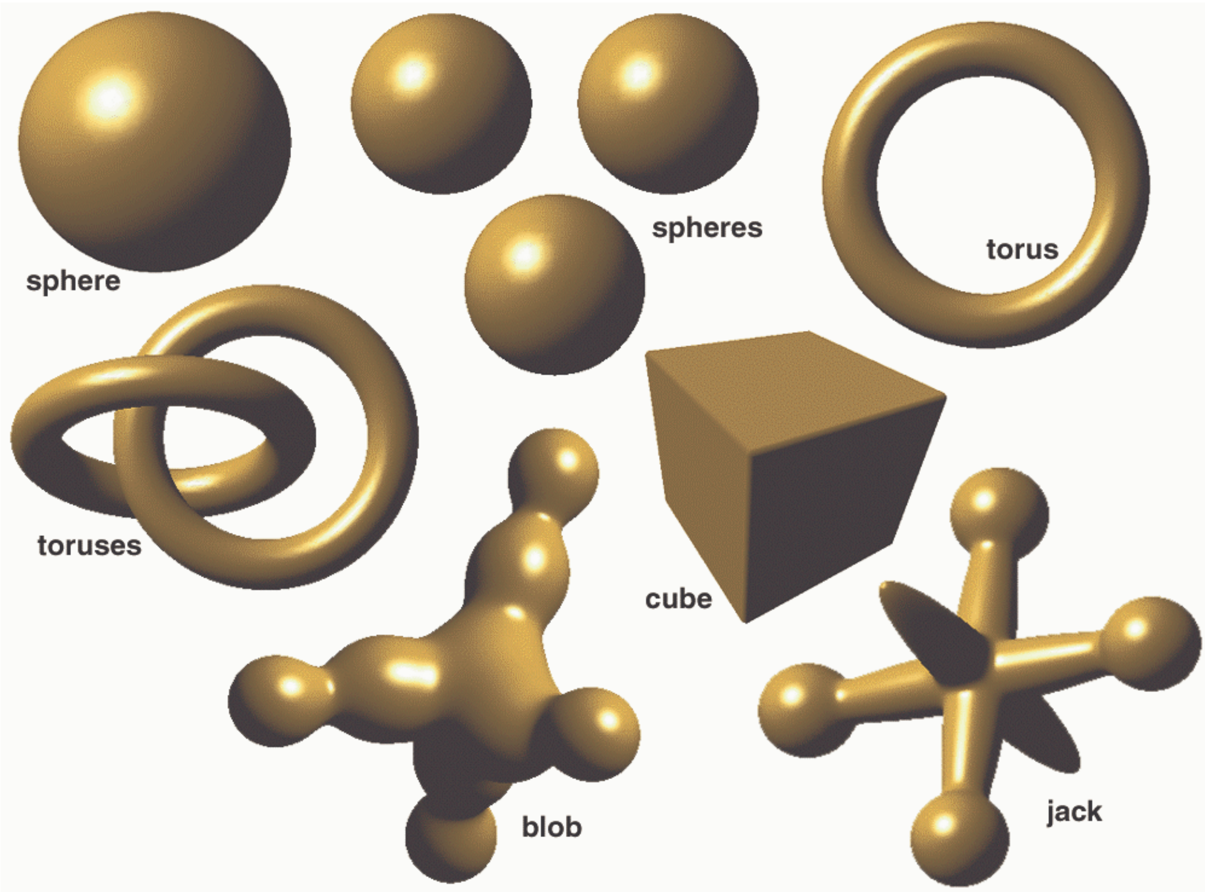
Obrázek 4.10: Chybný výpočet směru normály má za následek chybné určení pozice bodu ***q*** a tím i chybné vypočítání pozice bodu ***p*** na povrchu objektu v již triangulované oblasti.

Tento problém algoritmu *Marching triangles* se zatím nepodařilo úspěšně vyřešit, proto je popisovaná metoda funkční jen pro implicitní funkce C^1 spojité, které jsou charakteristické oblými tvary. V kapitole 5 je proto metoda *Marching triangles* porovnávána jen na funkcích s hladkým povrchem.

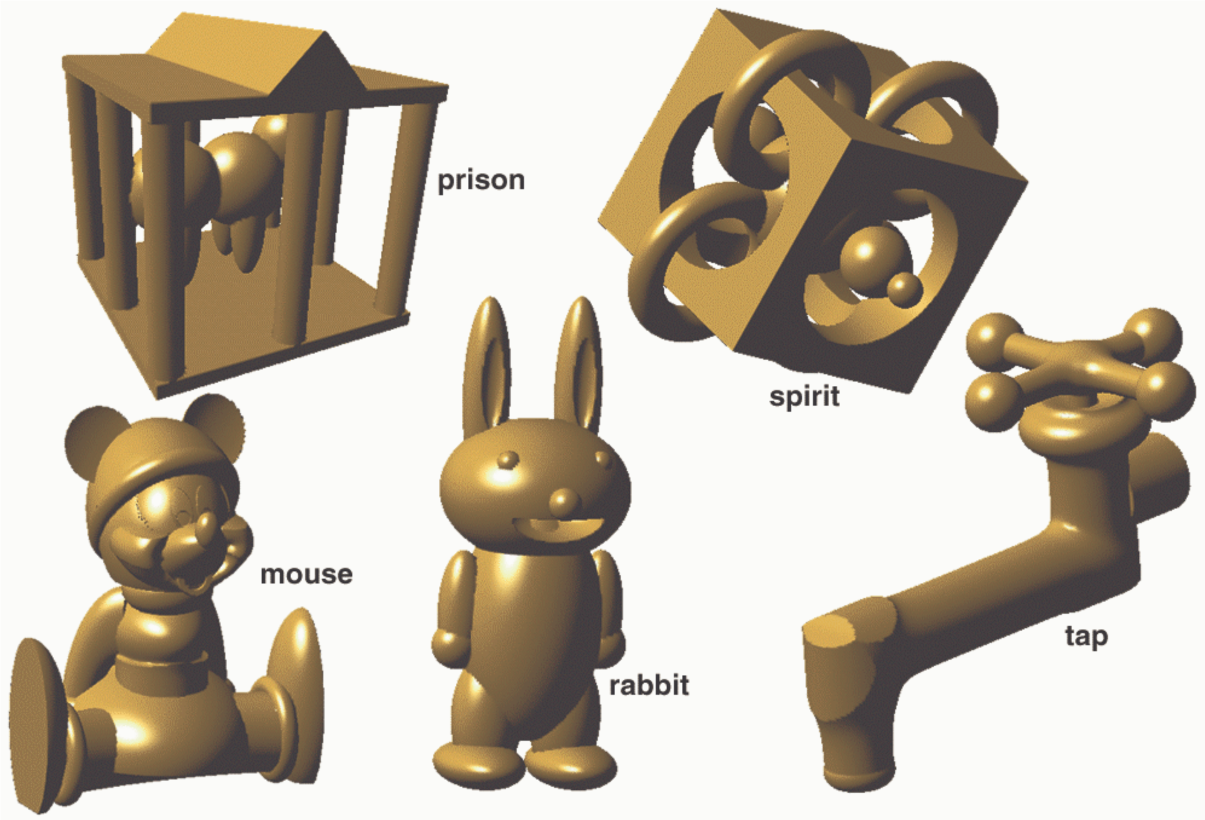
5 Dosažené výsledky

V předchozích kapitolách bylo představeno několik metod určených k polygonizaci objektů definovaných implicitními funkcemi. V této kapitole budou porovnány naměřené výsledky. Na obrázku 5.1 jsou zobrazeny tvary implicitních funkcí předdefinovaných v modulu *Polyg_editor* a na obrázku 5.2 jsou objekty modulu *Modeler* kolegy Karla Uhlíře (kuhlir@students.zcu.cz).

Realizované testy byly prováděny přímo v prostředí MVE a k vizualizaci všech funkcí byl použit modul *Renderer* kolegy Marka Krejzy (mkrejza@students.zcu.cz).



Obrázek 5.1: Předdefinované objekty modulu *Polyg_editor*.



Obrázek 5.2: Objekty modulu *Modeler*.

5.1 Časová a paměťová složitost

Jednotlivé implementované algoritmy se vzájemně odlišují, jak v přístupu k polygonizaci, což se projevuje hlavně rozdílnými časy výpočtu, tak i použitými datovými strukturami.

5.1.1 Časová složitost

V tomto odstavci porovnáme časové složitosti jednotlivých algoritmů.

Hledání počátečního bodu

Obě metody, numerické vyhledávání i metoda *Random search*, mají složitost $O(N)$, kde N je počet kroků potřebných k nalezení počátečního bodu.

Algoritmus Marching cubes

Algoritmus prochází jen buňky, které jsou protnuty povrchem funkce. Počet takových buněk je úměrný ploše funkce. Např. pro kouli je počet buněk roven $M = 4 \cdot \pi \cdot r^2$, kde r je poloměr koule. Odhad složitosti metody *Marching cubes* je $O(N^2)$, kde N je určitá míra velikosti objektu (viz [Bloo94]).

Exhaustive search

Algoritmus prochází celou oblast v E^3, která je rozdělena v každé ose na N částí. Složitost algoritmu je $O(N^3)$.

Marching triangles

Algoritmus prochází v každém kroku $A*N$ bodů, kde A je počet testovaných sousedních oblastí (1-26 ve smyslu 26-ti okolí testované oblasti) a N je průměrný počet bodů v každé z oblastí. Počet kroků K je úměrný počtu generovaných bodů. Výsledná složitost algoritmu je $O(A*K*N)$.

5.1.2 Paměťová složitost

Metody *Marching cubes* a *Exhaustive search* mají stejnou paměťovou složitost, protože využívají stejné datové struktury. Každý vrchol si uchovává informace o jeho souřadnicích v prostoru (3*8 Byte) a normálovém vektoru (3*8 Byte). Trojúhelník je tvořen třemi indexy do pole vrcholů (3*4 Byte). V paměti jsou dále během výpočtu uloženy hashovací tabulky CentreList, EdgeList a CornerList, které slouží k uchovávání již vypočtených hodnot a tím k urychlení výpočtu. Tabulka CentreList uchovává indexy buněk (3*4 Byte) ve kterých již proběhl výpočet. Tabulka EdgeList uchovává indexy dvojice rohů krychle, které společně tvoří hranu, tj. šestice indexů (6*4 Byte). Tabulka CornerList uchovává index již spočítaných rohů (3*4 Byte) i s jejich funkční hodnotu (8 byte).

Metoda *Marching triangles* využívá stejné datové struktury pro trojúhelník, ale odlišné pro vrchol, neboť vyžaduje uchovávat některé dodatečné informace. Pro každý vrchol je uložena jeho souřadnice v prostoru (3*8 Byte), normálový vektor (3*8 Byte) a dva tečné vektory (6*8 Byte), úhel pro polygonizaci – FrontAngle (8 Byte) a tři příznaky platnosti bodu – AngleChanged, BorderPoint a OutPoint (3*4 Byte). Pro urychlení výpočtu je v paměti uložena jedna hashovací tabulka, jejíž každá položka obsahuje spojový seznam indexů bodů (n*4 Byte).

5.2 Naměřené časy

Po uvedení časových a paměťových složitostí jednotlivých implementovaných metod, můžeme přistoupit ke konkrétním naměřeným hodnotám. Pro testování algoritmů byly použity předdefinované funkce modulu *Polyg_editor* a funkce definované v modulu *Modeler* kolegy Uhlíře (kuhlir@students.zcu.cz), obrázky 5.1 a 5.2.

Všechny následující testy byly realizovány v univerzitní laboratoři UL407 na pracovní stanici DELL Precision 410, 2x procesor Intel Pentium III, 1GB RAM, síťové označení NYMPH6.

5.2.1 Porovnání metod Marching cubes a Marching triangles

Pro testy byly použity funkce modulu Polyg_editor: *Sphere*, *Torus*, *Blob* a *Jack*. Srovnávána je převážně rychlost polygonizace, množství vygenerovaných dat a kvalita výsledné polygonální sítě. Naměřené hodnoty jsou znázorněny v tabulce 5.1.

Funkce	Metoda	N	160	240	400	630	1000
Sphere	Marching Cubes	Triangles:	3728	8432	23408	58472	147368
		Vertices:	1866	4218	11706	29238	73686
		Time [ms]:	94	156	375	937	2266
	Marching Triangles	Triangles:	3291	7445	20567	50496	127131
		Vertices:	1647	3724	10285	25249	63567
		Time [ms]:	47	141	500	1562	5110
Torus	Marching Cubes	Triangles:	520	1280	3568	8352	22912
		Vertices:	256	640	1784	4176	11456
		Time [ms]:	63	62	93	172	562
	Marching Triangles	Triangles:	515	1159	3229	7891	20025
		Vertices:	258	579	1615	3945	10012
		Time [ms]:	15	31	78	172	500
Blob	Marching Cubes	Triangles:	12704	28600	79348	197180	496636
		Vertices:	6354	14298	39676	98592	248320
		Time [ms]:	469	969	2531	6016	14000
	Marching Triangles	Triangles:	11283	24869	68807	170753	430251
		Vertices:	5643	12434	34405	85376	215127
		Time [ms]:	390	859	2718	8000	25969
Jack	Marching Cubes	Triangles:	16984	38680	106888	264728	667144
		Vertices:	8494	19342	53446	132366	333574
		Time [ms]:	703	2172	4047	7610	20828
	Marching Triangles	Triangles:	15647	34707	96051	237405	597965
		Vertices:	7825	17355	48025	118704	298982
		Time [ms]:	469	1063	3344	10485	29578

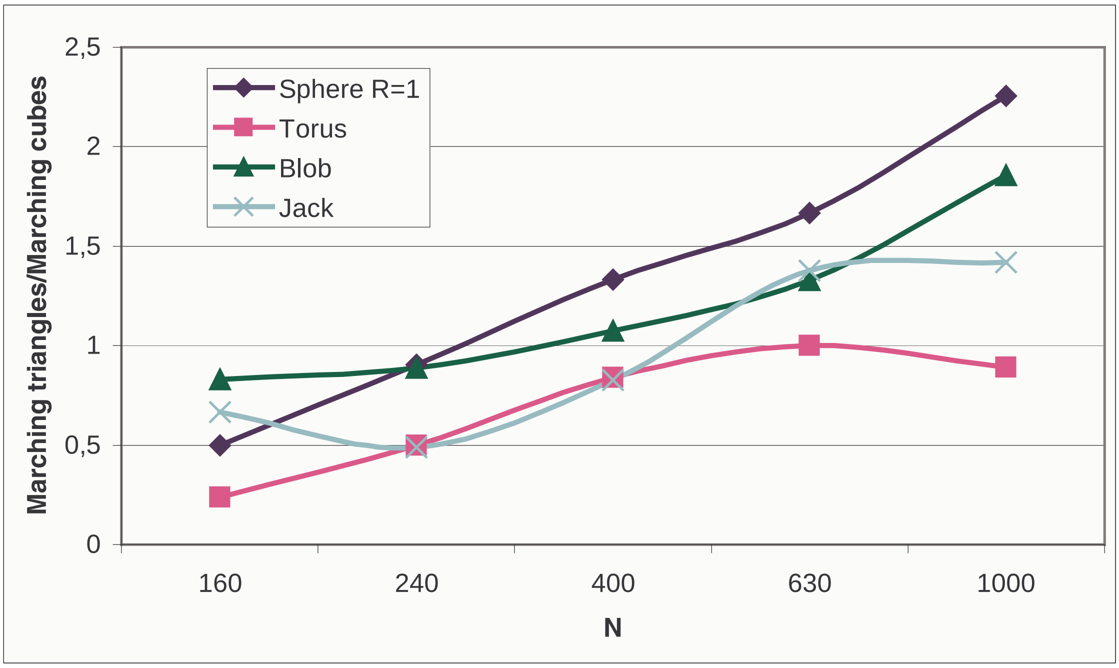
Tabulka 5.1: Porovnání metod *Marching cubes* a *Marching triangles* na objektech *Sphere*, *Torus* , *Blob* a *Jack*.

V průběhu měření byl nastaven parametr *Area size* podle tabulky 5.2.

	min	max
x	-8	8
y	-8	8
z	-8	8

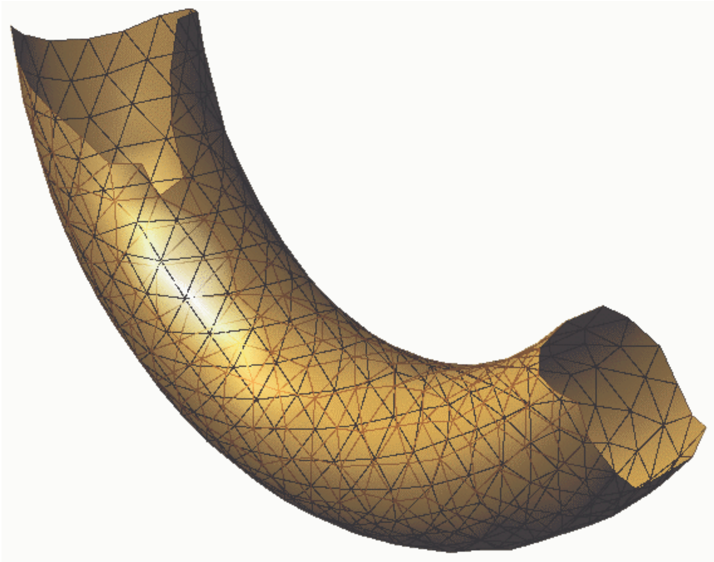
Tabulka 5.2: Oblast *Area size* během výpočtů.

V grafu 5.1 je znázorněn poměr naměřených časů metod *Marching triangles* a *Marching cubes*.



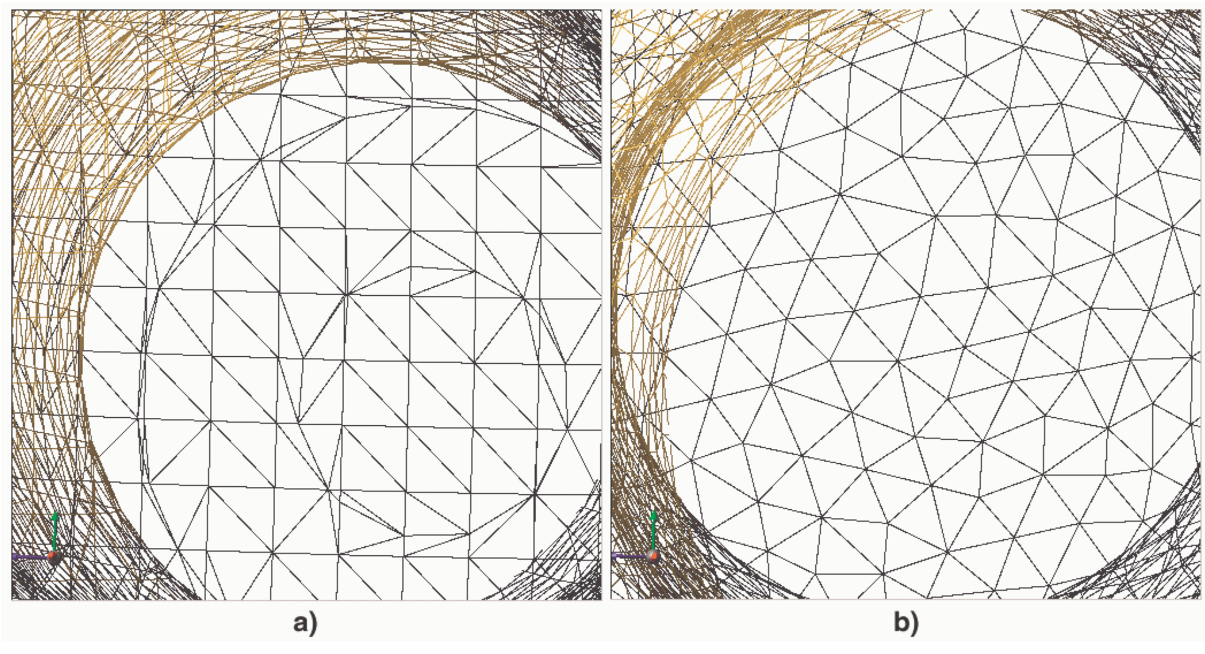
Graf 5.1: Porovnání rychlosti algoritmů *Marching cubes* a *Marching triangles*.

Z grafu 5.1 vyplývá, že ve většině případů metoda *Marching triangles* polygonizuje povrch funkce pomaleji se vzrůstajícím dělením prostoru. To je způsobeno tím, že v jednotlivých záznamech hashovací tabulky (viz kapitola 4.6.6) je větší výskyt synonym a je nutné testovat více bodů na vzdálenost (kapitola 4.6.4). Výjimkou je funkce *Torus*, jejíž prstencovitý tvar způsobuje, že polygonizační proces postupuje po vnějším obvodu funkce. Vzniklé front polygony pak obsahují malý počet bodů a testy vzdálenosti probíhají rychleji. Situace je znázorněna na obrázku 5.3. Podobný efekt vzniká u funkcí válcovitého tvaru.



Obrázek 5.3: Znázornění dvou front polygonů (hranice objektu) v jejichž směru pokračuje polygonizační proces, část objektu *Torus*.

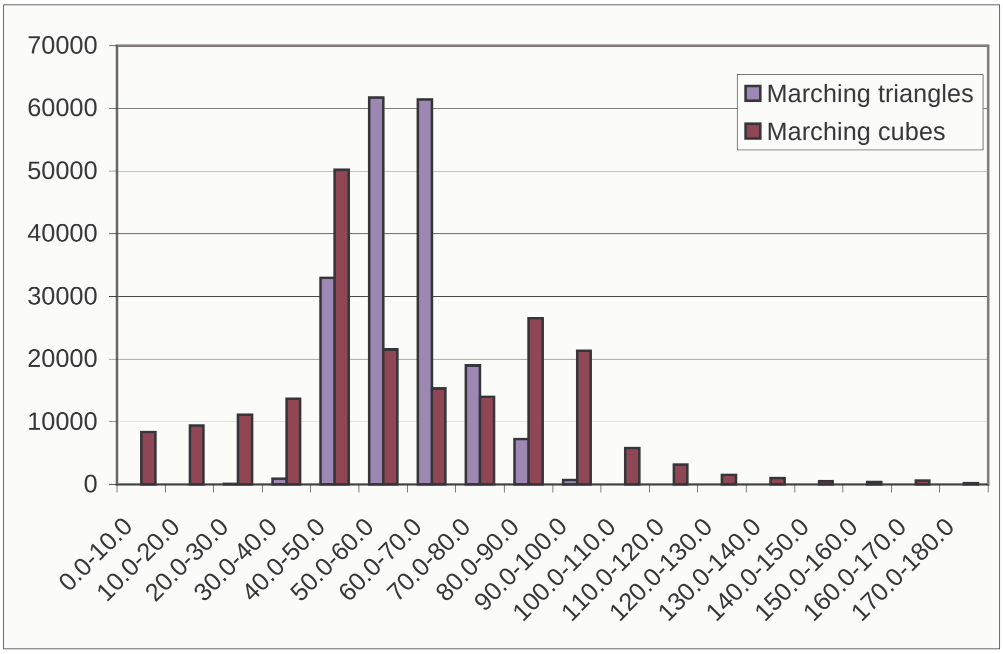
Vizuální porovnání kvality polygonální sítě generované oběma metodami je dobře znázorněno na následujícím obrázku.



Obrázek 5.4: Detailní pohled na funkci *Jack* pro vizuální porovnání kvality polygonální sítě metod *Marching cubes* (a) a *Marching Triangles* (b).

Z obrázku 5.4 vyplývá, že metoda *Marching triangles* produkuje kvalitnější polygonální síť, ve které nevznikají „úzké“ trojúhelníky, a na které není patrná pravidelná mřížka velikosti původního dělení prostoru.

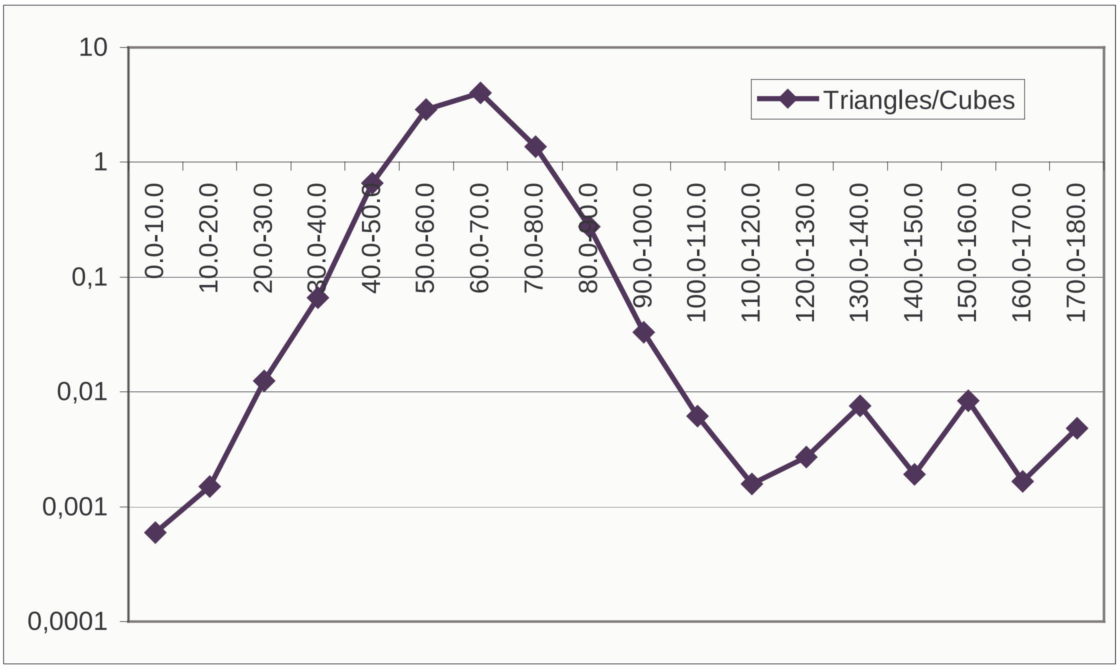
Pro jiné než vizuální posouzení kvality vznikající trojúhelníkové sítě je v grafu 5.2a zobrazen histogram četnosti jednotlivých úhlů v triangulaci (na ose X jsou znázorněny úhlové intervaly a na ose Y počet úhlů v těchto intervalech).



Graf 5.2a: Histogram četnosti jednotlivých úhlů v trojúhelníkové síti generované metodou *Marching cubes* a *Marching triangles*, funkce *Jack*.

Z histogramu je patrné, že v polygonizaci metodou *Marching triangles* převládají trojúhelníky s úhly v intervalu (50°, 70°).

V grafu 5.2b je znázorněn poměr množství úhlů v jednotlivých intervalech ve tvaru Marching triangles/Marching cubes.



Graf 5.2b: Poměr četnosti jednotlivých úhlů v trojúhelníkové síti generované metodou *Marching cubes* a *Marching triangles*, funkce *Jack*.

5.2.2 Porovnání jednotlivých implementací algoritmu Marching triangles

V tomto odstavci bude kladen důraz na porovnání rychlosti původního algoritmu *Marching triangles* a obou upravených verzí, které byly popsány v kapitolách 4.6.5 a 4.6.6.

Následující test byl proveden na objektu *Sphere*, *Torus* a *Blob* a byla sledována závislost doby výpočtu na dělení prostoru. Dělení prostoru bylo voleno podle řady *E5*, tj. v násobcích *1,6* ; *2,4* ; *4,0* ; *6,3* ; *10* .

N		160	240	400	630	1000
		Naměřený čas [ms]				
Sphere	Originál	63	234	1715	10656	82094
	1. Úprava	31	46	156	422	1500
	2. Úprava	16	32	109	265	844
Torus	Originál	78	78	406	1406	100843
	1. Úprava	31	62	141	297	1468
	2. Úprava	16	31	63	188	469
Blob	Originál	641	2578	17797	122984	---
	1. Úprava	125	297	922	2844	11750
	2. Úprava	94	218	609	1610	4594

Tabulka 5.3: Porovnání rychlosti jednotlivých verzí algoritmu *Marching triangles* na funkcích *Sphere*, *Torus* a *Blob* v závislosti na dělení prostoru.

Poznámka:

Pro originální algoritmus nebyla změřena hodnota odpovídající dělení os $N = 1000$, protože doba výpočtu byla velmi dlouhá.

Nastavení oblasti v průběhu měření je znázorněno v tabulce 5.4.

	min	max
x	-8	8
y	-8	8
z	-8	8

a)

	min	max
x	-16	16
y	-16	16
z	-16	16

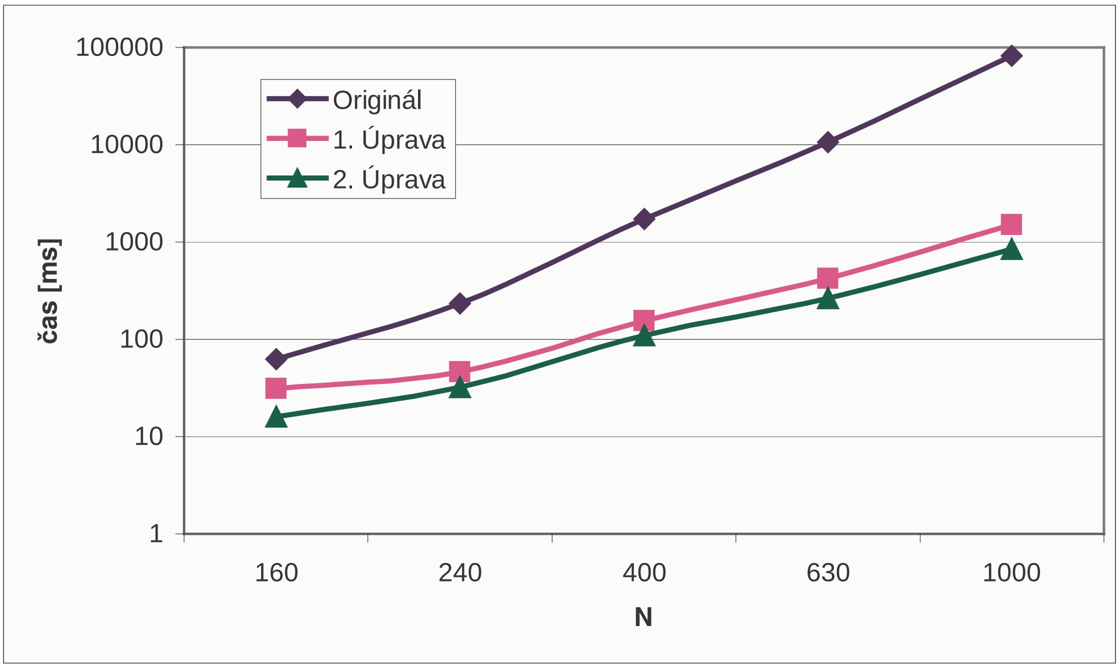
b)

Tabulka 5.4: Parametr *Area size* během měření funkce a) *Sphere* a *Torus*, b) *Blob*.

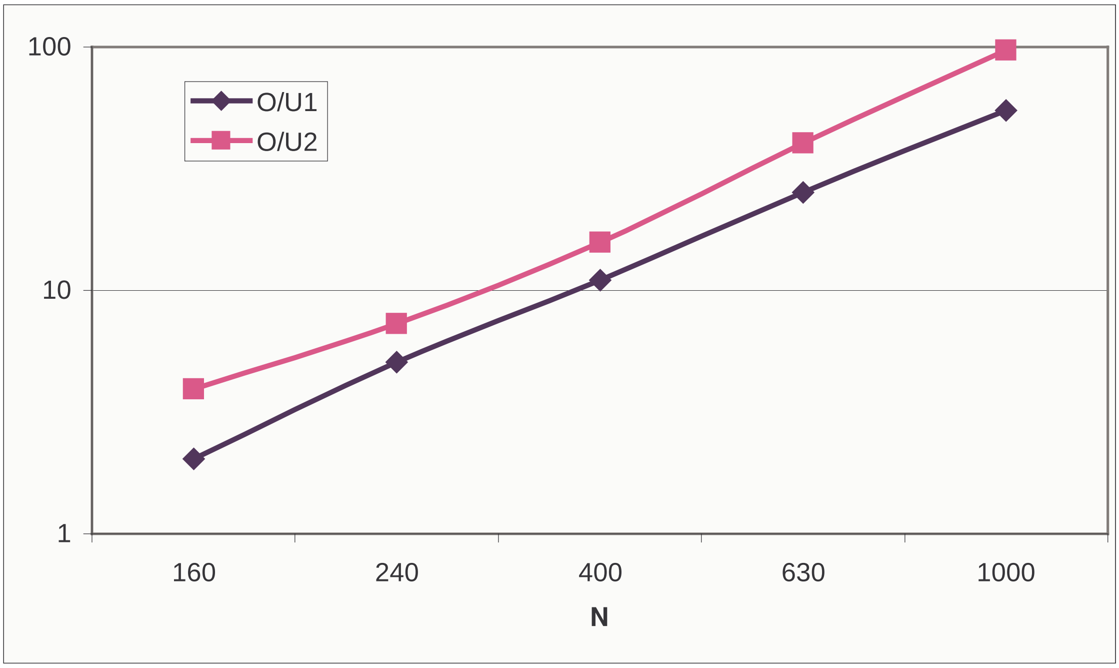
Poznámka:

V případě měření funkce *Blob* byla sledovaná oblast zvětšena na dvojnásobek z důvodu snížení detailů (zvětšení mřížky, tzn. parametr *cubesize*), aby měřené časy originálního algoritmu nebyly příliš dlouhé.

Výsledky uvedené v tabulce 5.3 jsou graficky zobrazeny v grafech 5.3-5.11. Pro větší názornost, jsou zvlášť uváděny grafy s poměry rychlostí výpočtu $q = {}^{1.metoda}\!/_{2.metoda}$.

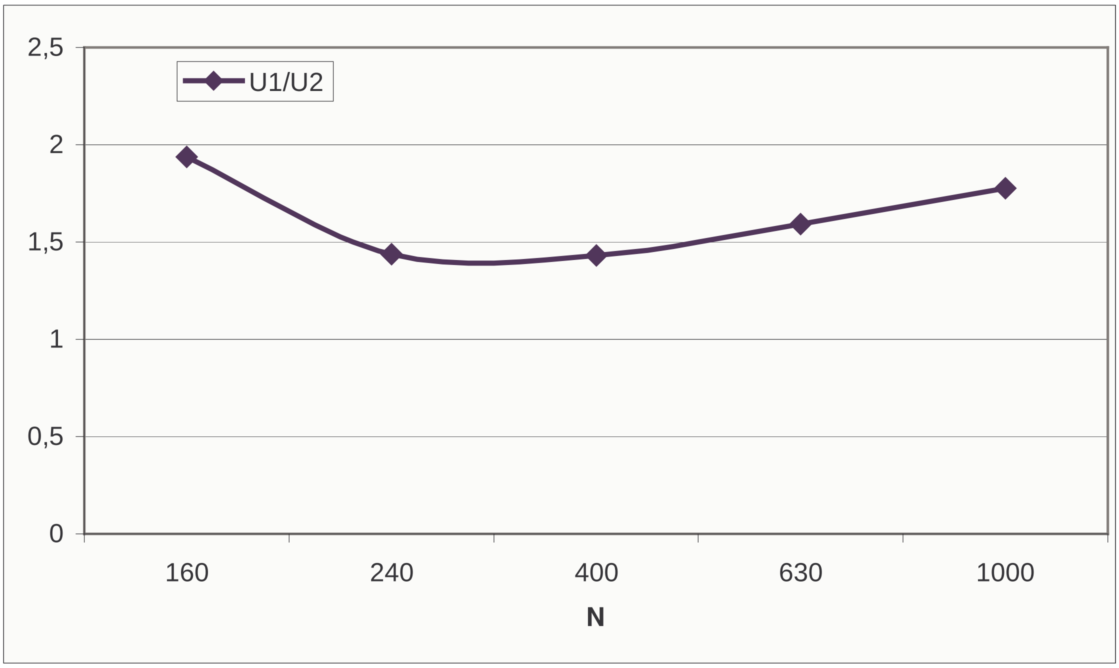


Graf 5.3: Grafické porovnání rychlosti jednotlivých verzí algoritmu *Marching triangles*, funkce *Sphere*.

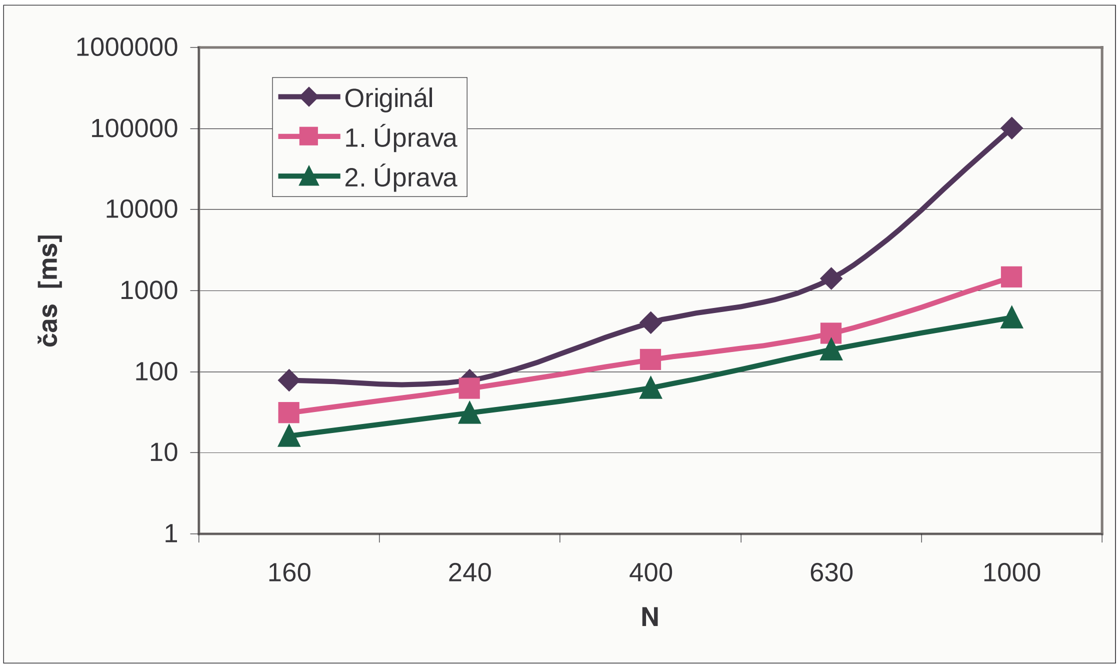


Graf 5.4: Grafické znázornění poměru rychlostí originálního algoritmu a upravených verzí, funkce *Sphere*.

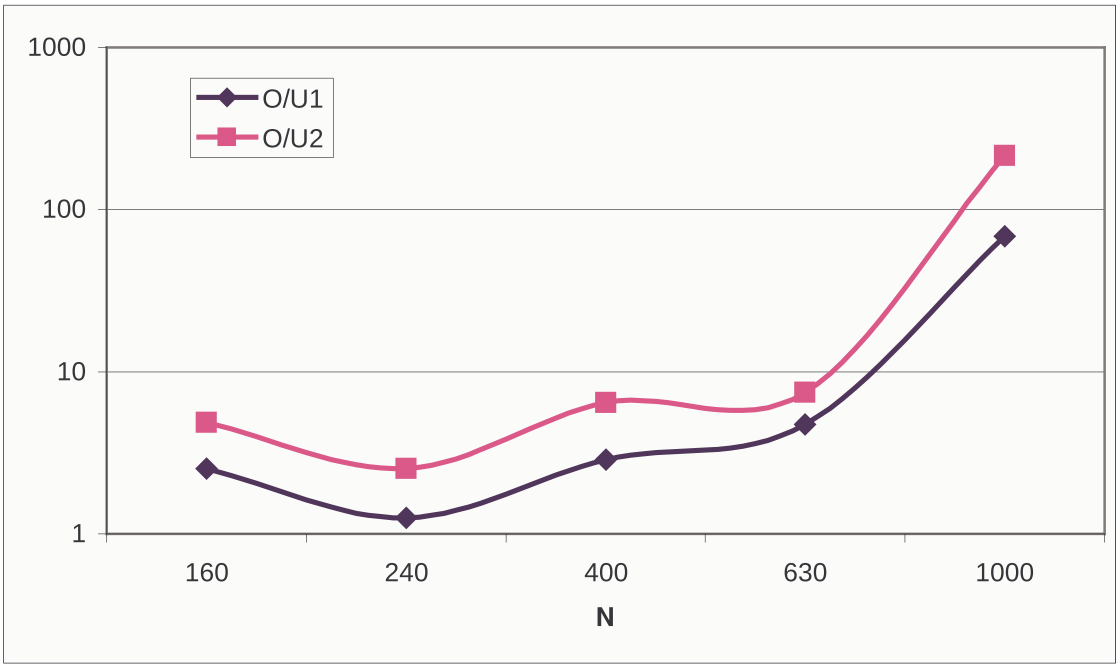
Porovnání nových verzí algoritmu je zobrazeno v grafu 5.5.



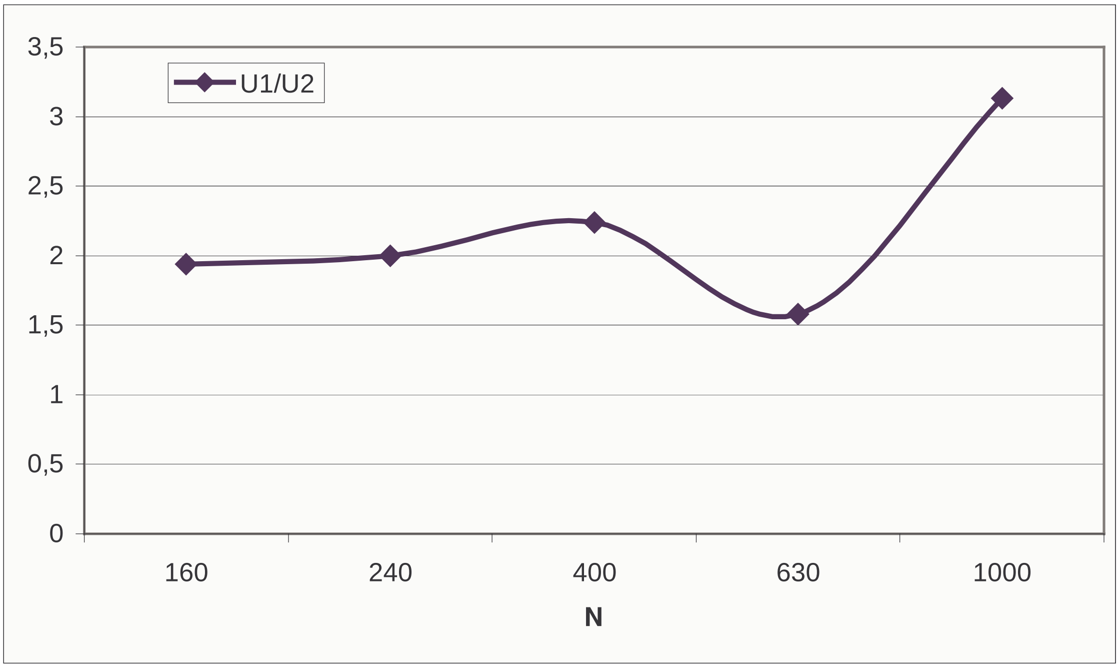
Graf 5.5: Grafické znázornění poměru upravených verzí algoritmu, funkce *Sphere*.



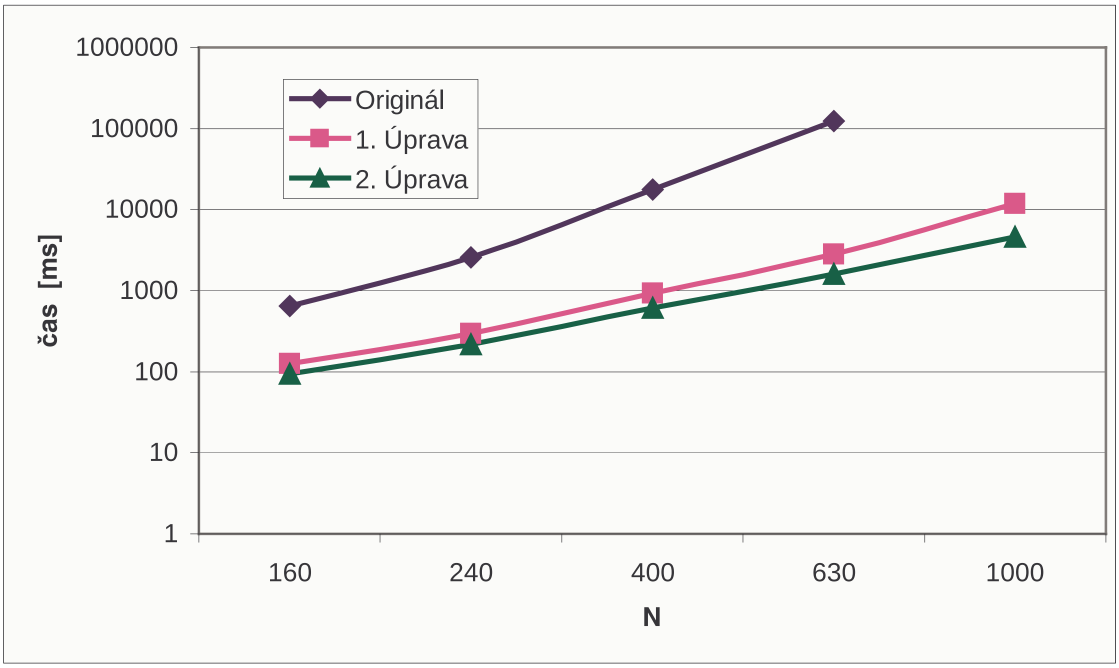
Graf 5.6: Grafické porovnání rychlosti jednotlivých verzí algoritmu *Marching triangles*, funkce *Torus*.



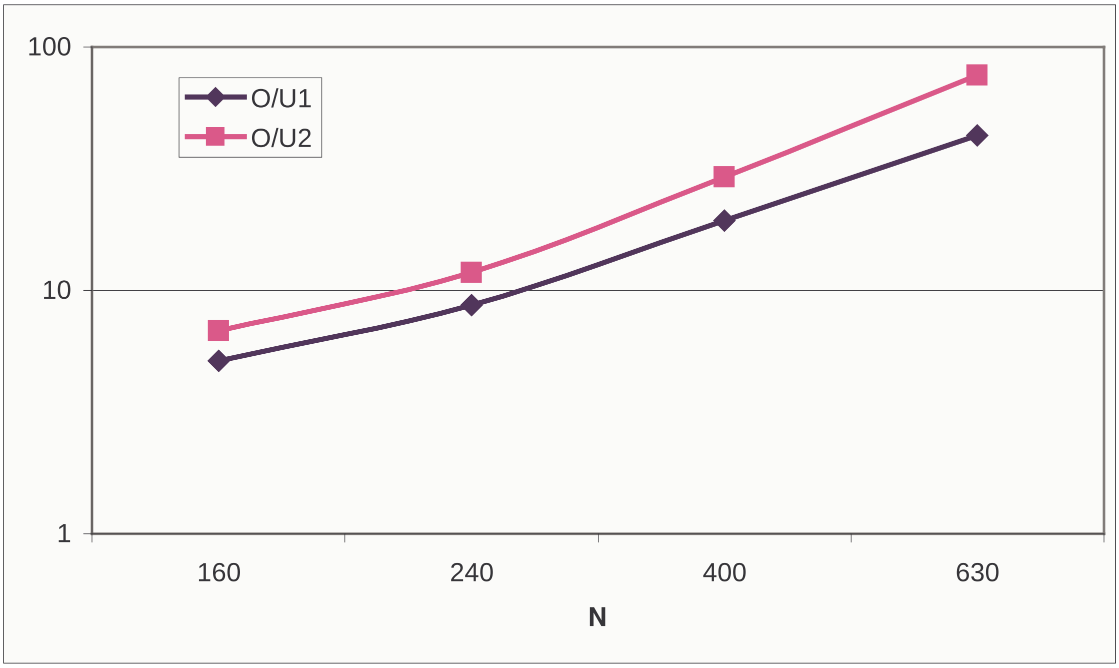
Graf 5.7: Grafické znázornění poměru originálního algoritmu a upravených verzí, funkce *Torus*.



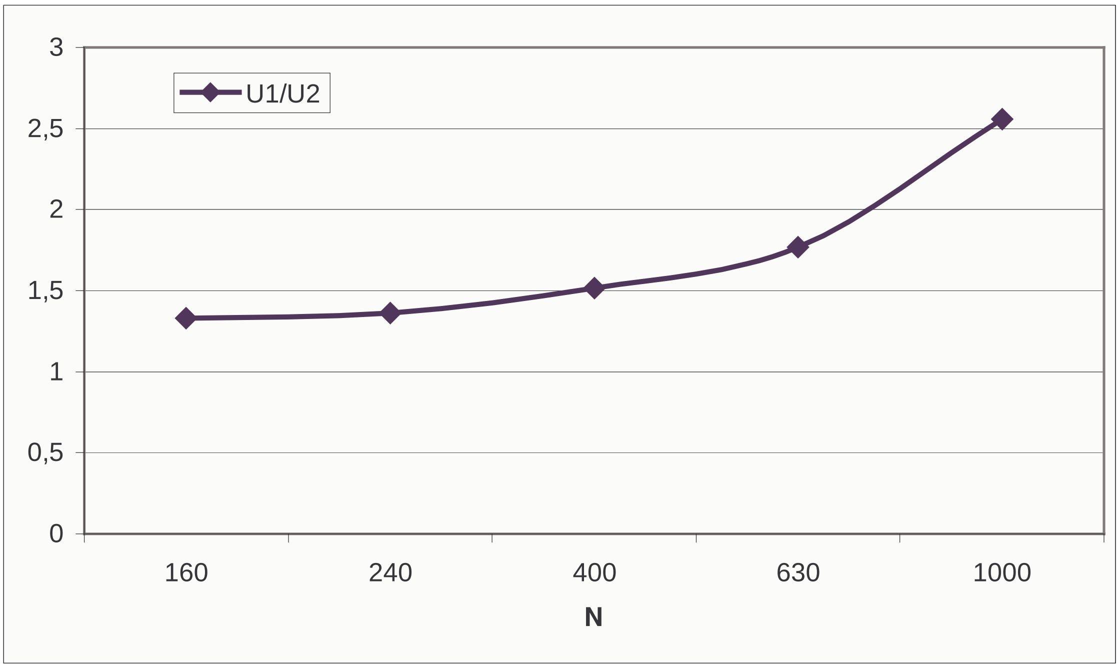
Graf 5.8: Grafické znázornění poměru upravených verzí algoritmu, funkce *Torus*.



Graf 5.9: Grafické porovnání rychlosti upravených verzí algoritmu *Marching triangles*, funkce *Jack*.



Graf 5.10: Grafické znázornění poměru originálního algoritmu a upravených verzí, funkce *Jack*.



Graf 5.11: Grafické znázornění poměru upravených verzí algoritmu, funkce *Jack*.

Z naměřených hodnot vyplývá, že výpočet upravených verzí algoritmu *Marching triangles* je mnohonásobně rychlejší (v některých případech až stonásobně, viz grafy 5.5 a 5.7) než původní algoritmus, přičemž rozdíl doby výpočtu mezi upravenými algoritmy také není zanedbatelný.

5.2.3 Paralelní běh

Jak bylo uvedeno v kapitole 4.4, paralelně implementovanou metodou je metoda *Exhaustive search*, proto bude v následujících odstavcích podrobena několika testům.

Pro určování kvality paralelního algoritmu je nezbytná znalost pojmů *Urychlení* a *Efektivita* paralelního výpočtu.

Urychlení

Urychlení (angl. *speedup*) ***s*** paralelního algoritmu je obvykle vyhodnocováno jako podíl

$$s = \frac{t_1}{t_n} \qquad (5.1),$$

kde t_1 je čas výpočtu na jednom procesoru a t_n je čas výpočtu na n procesorech.

Efektivita

Efektivita, resp. účinnost (angl. *efficiency*) ***e***, je *urychlení* dělené počtem použitých procesorů ***p***. Zapsáno vzorcem:

$$e = \frac{s}{p} = \frac{t_1}{p \cdot t_n} \qquad (5.2).$$

V tabulce 5.5 jsou uvedeny naměřené časy při běhu programu na počítači se dvěma procesory.

	N	20	40	60	80	100
Funkce	Počet procesorů	Naměřený čas [ms]				
Spheres	1	62	265	765	1735	3360
	2	78	172	422	953	1781
Blob	1	94	438	1375	3156	5937
	2	78	296	734	1640	3172
Jack	1	94	438	1172	2765	5078
	2	94	265	672	1484	2844
Tap	1	547	3844	12391	28719	55609
	2	375	2172	6750	15313	29266
Spirit	1	609	3625	9609	23046	42844
	2	500	2407	5406	12735	23328

Tabulka 5.5: Naměřené časy při paralelním běhu metody *Exhaustive search*.

Nastavení oblasti v průběhu měření je znázorněno v tabulce 5.6.

a)

	min	max
x	-2	2
y	-2	2
z	-2	2

b)

	min	max
x	-4	4
y	-4	4
z	-4	4

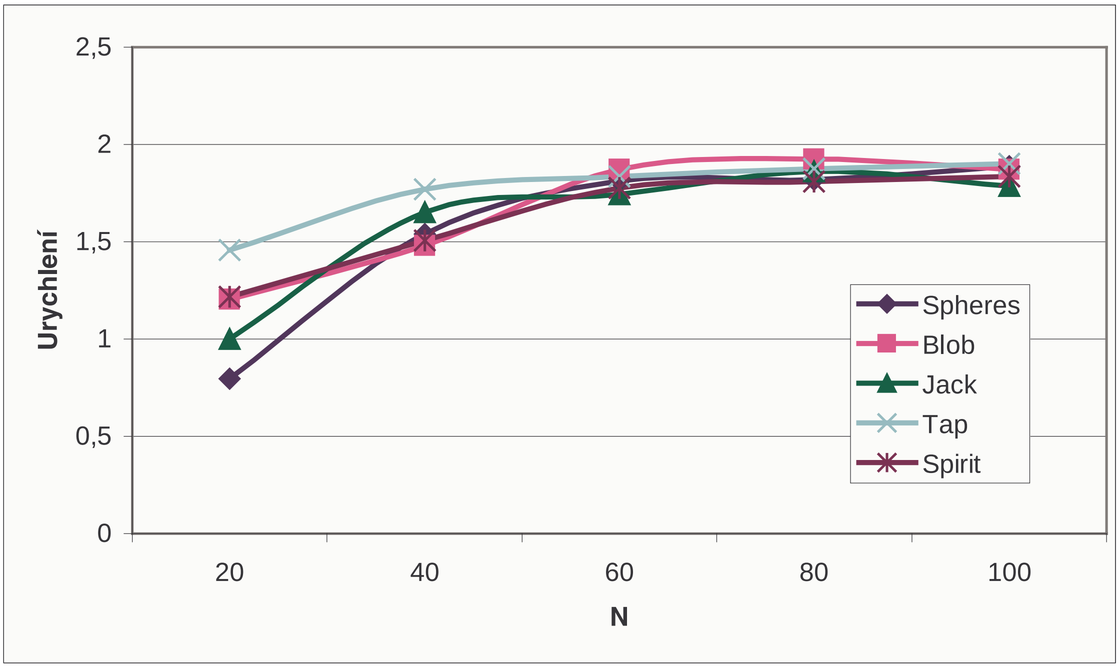
c)

	min	max
x	-12	12
y	-12	12
z	-12	12

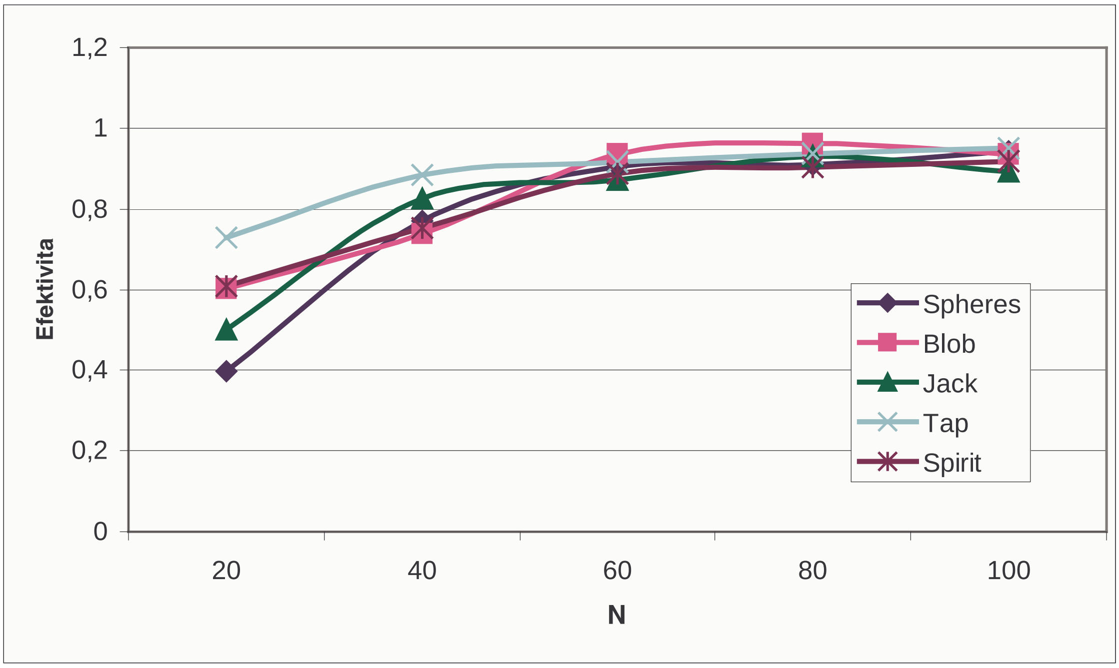
Tabulka 5.6: Parametr *Area size* pro funkce a) *Spheres*, b) *Blob* a *Jack*, c) *Tap* a *Spirit*.

Parametr *Area size* byl v průběhu měření měněn pouze z důvodu velikosti jednotlivých funkcí, aby byly vygenerované objekty celé.

Závislost urychlení a efektivity na typu objektu je znázorněna v grafech 5.12 a 5.13.



Graf 5.12: Závislost urychlení paralelního algoritmu na typu objektu pro dva procesory.



Graf 5.13: Závislost efektivity paralelního algoritmu na typu objektu pro dva procesory.

Vypočítané hodnoty urychlení a efektivity paralelního algoritmu jsou pro krátké časy (malá dělení prostoru) více vzdáleny očekávaným výsledkům. Důvodem je určitá časová

„režie“, která je nezbytná pro rozběhnutí vláken. Pro delší doby výpočtu se již dosažené výsledky přibližují ideálním hodnotám.

Potenciálně paralelní část kódu

Urychlení výpočtu paralelního algoritmu je omezeno tzv. Amdahlovým zákonem. Tento zákon bere v úvahu, že výpočet zpravidla nelze paralelizovat úplně, tj. určitá část musí být provedena sekvenčně.

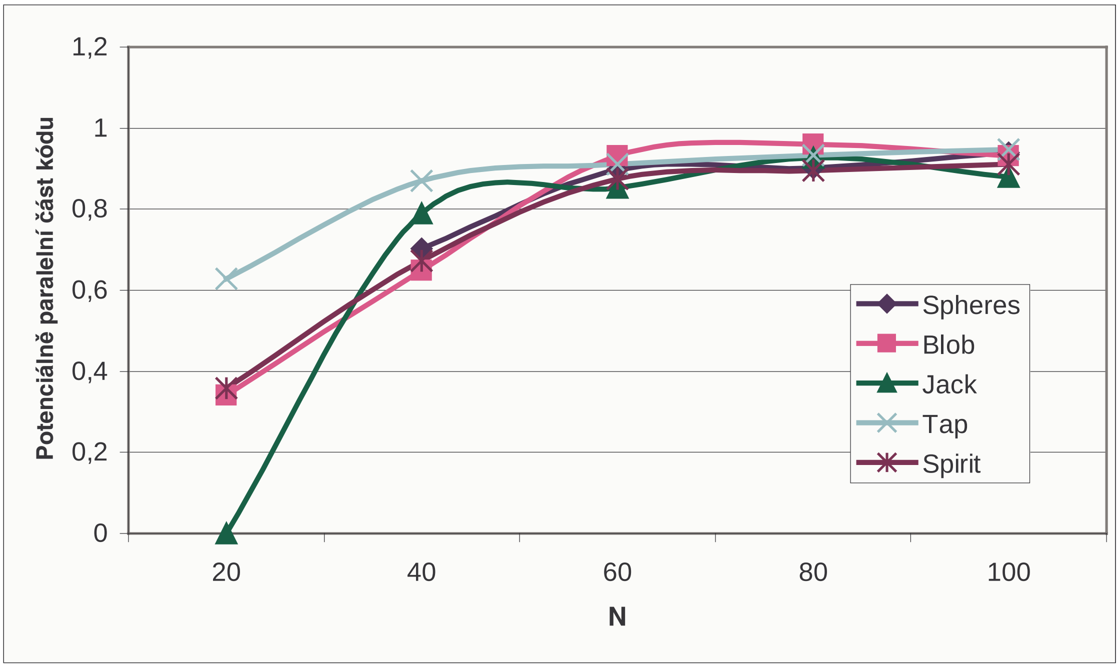
Amdahlův zákon lze vyjádřit vztahem:

$$s \le \frac{1}{f + \frac{1-f}{p}} \le \frac{1}{f} \qquad (5.3),$$

kde ***s*** je urychlení, f je sekvenční část paralelního kódu a ***p*** je počet procesorů.
Z Amdahlova zákona můžeme vyjádřit koeficient $q = 1 - f$ představující potenciálně paralelní část kódu:

$$q = \frac{p \cdot (s_{teor} - 1)}{s_{teor} \cdot (p - 1)} \qquad (5.4),$$

kde s_{teor} je teoreticky dosažitelné urychlení vyjádřené z Amdahlova zákona.
Závislost potenciálně paralelní části kódu na typu objektu a dělení oblasti je zobrazena v grafu 5.14. V našem testu bylo pro určení koeficientu q použito vypočítané urychlení s z naměřených hodnot v tabulce 5.5.



Graf 5.14: Závislost potenciálně paralelní části kódu na typu objektu a dělení prostoru pro dva procesory.

Výpočet potenciálně paralelní části kódu je, podobně jako efektivita a urychlení, ovlivněn režií vláken a pro krátké časy výpočtu jsou dosažené hodnoty vzdálené od očekávaných. Pro funkci *Spheres* vyšel koeficient q dokonce záporný, neboť naměřený čas běhu programu na dvou procesorech vyšel delší než při běhu na jednom procesoru. Z tohoto důvodu nebyla do grafu 5.13 zanesena hodnota pro dělení prostoru N = 20 (funkce *Sphere*), protože potenciálně paralelní část kódu nemůže být menší než nulová.

5.3 Porovnání kvality aproximace

Následující pokus byl proveden na funkci *Sphere* (koule) o poloměru $R=1$, kde byla sledována závislost přesnosti aproximace na normalizaci implicitní funkce. Testy byly provedeny na metodách *Marching triangles* a *Marching cubes*.

Všechny použité funkce v této kapitole jsou předdefinovány v modulu *Polyg_editor*.

Implicitní funkce pro kouli: $$p_x^{\ 2} + p_y^{\ 2} + p_z^{\ 2} - r^2 = 0. \tag{5.5}$$

Normalizovaný tvar: $$\sqrt{p_x^{\ 2} + p_y^{\ 2} + p_z^{\ 2}} - r = 0. \tag{5.6}$$

Normalizovaný tvar zaručuje, že se numerickým výpočtem přibližujeme k povrchu objektu po přímce a ne po parabolické křivce jako v prvním případě. Výpočet pozice každého vrcholu je zastaven po splnění podmínky $f(\boldsymbol{X}) < \varepsilon$, kde $\boldsymbol{X}$ je hledaný bod na povrchu. Konstanta ε je v modulu *Polygonizer* nastavena na hodnotu 10^{-5}.

Nastavení oblasti v průběhu měření je znázorněno v tabulce 5.7 a naměřené hodnoty v tabulkách 5.8 a 5.9.

	min	max
x	-4	4
y	-4	4
z	-4	4

Tabulka 5.7: Parametr *Area size* během výpočtů.

	N	160	240	400	630	1000
Normalizovaný tvar	Triangles:	13205	29681	81682	203133	509459
	Vertices:	6604	14842	40842	101568	254731
	Time [ms]:	266	1500	2796	9188	33953
	Total deviation:	0,04730	0,10616	0,29780	0,70550	1,51291
	Average deviation:	7,16291E-06	7,15293E-06	7,29139E-06	6,94610E-06	5,93926E-06
	Relative deviation:	5,70238E-07	5,69325E-07	5,80265E-07	5,52768E-07	4,72638E-07
	Surface area:	12,56126	12,56388	12,56563	12,56602	12,56618
Nenormalizovaný tvar	Triangles:	13141	29585	81217	202130	509851
	Vertices:	6572	14794	40610	101066	254927
	Time [ms]:	250	719	2828	10094	33422
	Total deviation:	0,04174	0,10736	0,29403	0,69446	1,66140
	Average deviation:	6,35088E-06	7,25686E-06	7,24040E-06	6,87139E-06	6,51715E-06
	Relative deviation:	5,05591E-07	5,77598E-07	5,76214E-07	5,46821E-07	5,18624E-07
	Surface area:	12,56129	12,56385	12,56548	12,56607	12,56623

Tabulka 5.8: Naměřené hodnoty odchylek aproximací normalizovaného a nenormalizovaného tvaru funkce *Sphere* generované metodou *Marching triangles*.

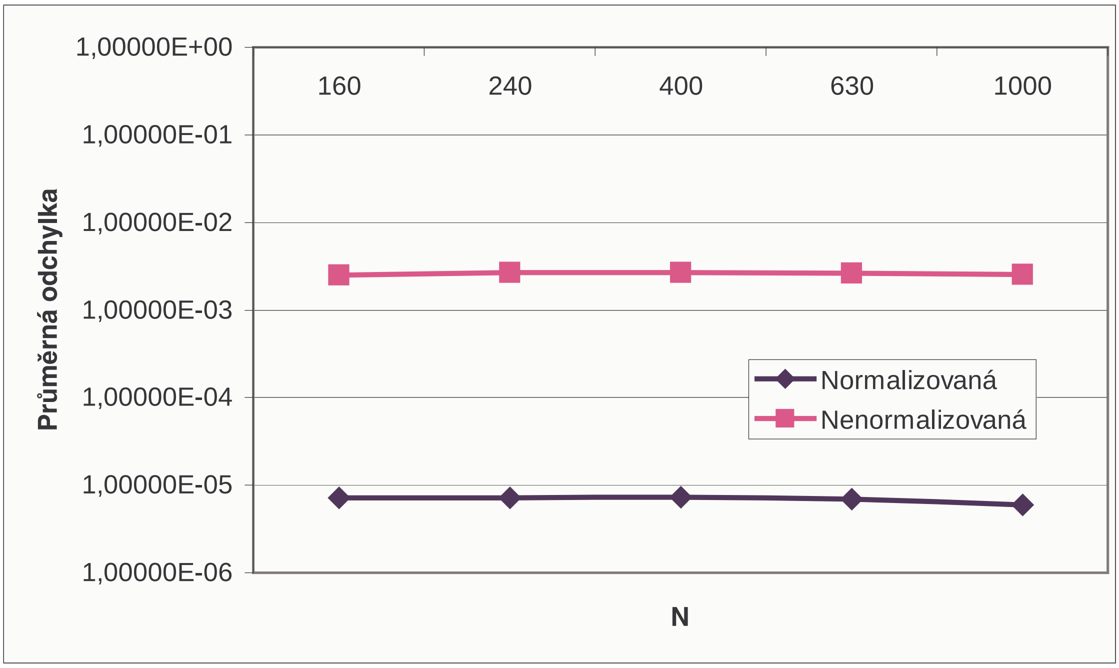
	N	160	240	400	630	1000
Normalizovaný tvar	Triangles:	14984	33752	93992	233720	588728
	Vertices:	7494	16878	46998	116862	294366
	Time [ms]:	766	1796	3735	5109	13813
	Total deviation:	0,03360	0,07506	0,20303	0,50700	1,32124
	Average deviation:	4,48368E-06	4,44696E-06	4,32002E-06	4,33842E-06	4,48843E-06
	Relative deviation:	3,57013E-07	3,53972E-07	3,43809E-07	3,45253E-07	3,57182E-07
	Surface area:	12,55889	12,56301	12,56518	12,5659	12,56623
Nenormalizovaný tvar	Triangles:	14984	33752	93992	233720	588728
	Vertices:	7494	16878	46998	116862	294366
	Time [ms]:	250	796	1734	3719	9672
	Total deviation:	0,03046	0,07426	0,21062	0,50627	1,28977
	Average deviation:	4,06433E-06	4,39965E-06	4,48145E-06	4,33223E-06	4,38153E-06
	Relative deviation:	3,23622E-07	3,50207E-07	3,56656E-07	3,44761E-07	3,48676E-07
	Surface area:	12,55891	12,56301	12,56517	12,56589	12,56619

Tabulka 5.9: Naměřené hodnoty odchylek aproximací normalizovaného a nenormalizovaného tvaru funkce *Sphere* generované metodou *Marching cubes*.

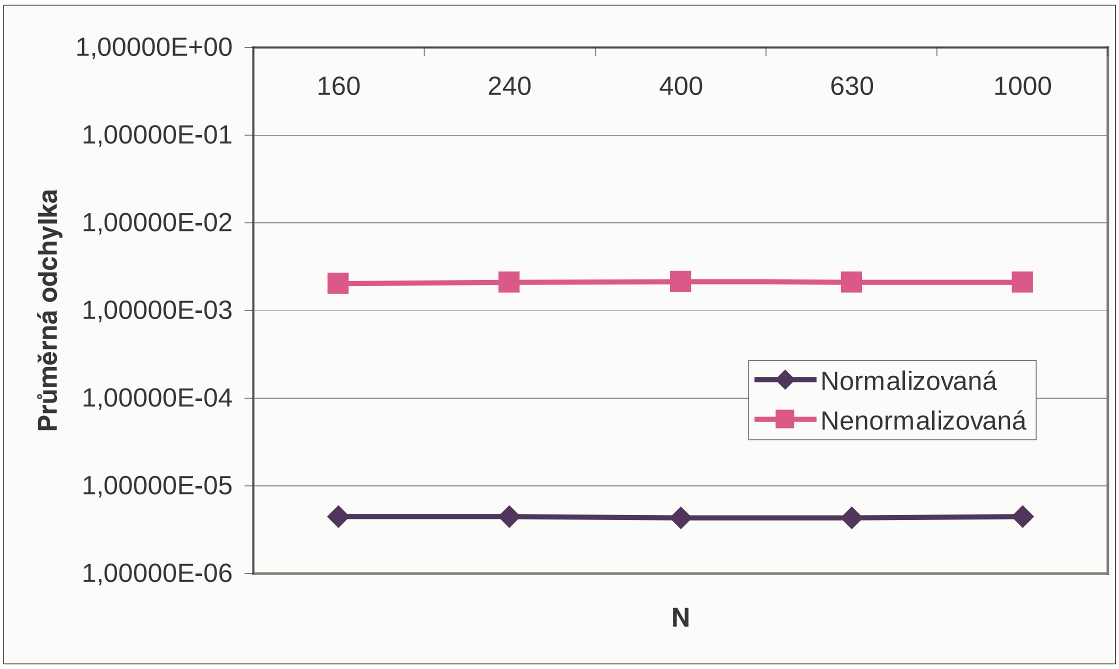
V grafech 5.15 a 5.16 je znázorněna závislost kvality aproximace polygonálního objektu normalizované a nenormalizované funkce *Sphere* (*R* = *1*).

Poznámka:

Nenormalizovaný tvar funkce (rovnice 5.5) navrací hodnotu, jejíž závislost je kvadratická. Naměřené odchylky v takovém případě odpovídají druhé mocnině skutečných hodnot, tj. před vlastním porovnáním s funkcí v normalizovaném tvaru (rovnice 5.6), je nutné získané výsledky nejprve odmocnit.



Graf 5.15: Porovnání kvality aproximace implicitní funkce pro normalizovaný a nenormalizovaný tvar funkce *Sphere* polygonizované metodou *Marching triangles*.



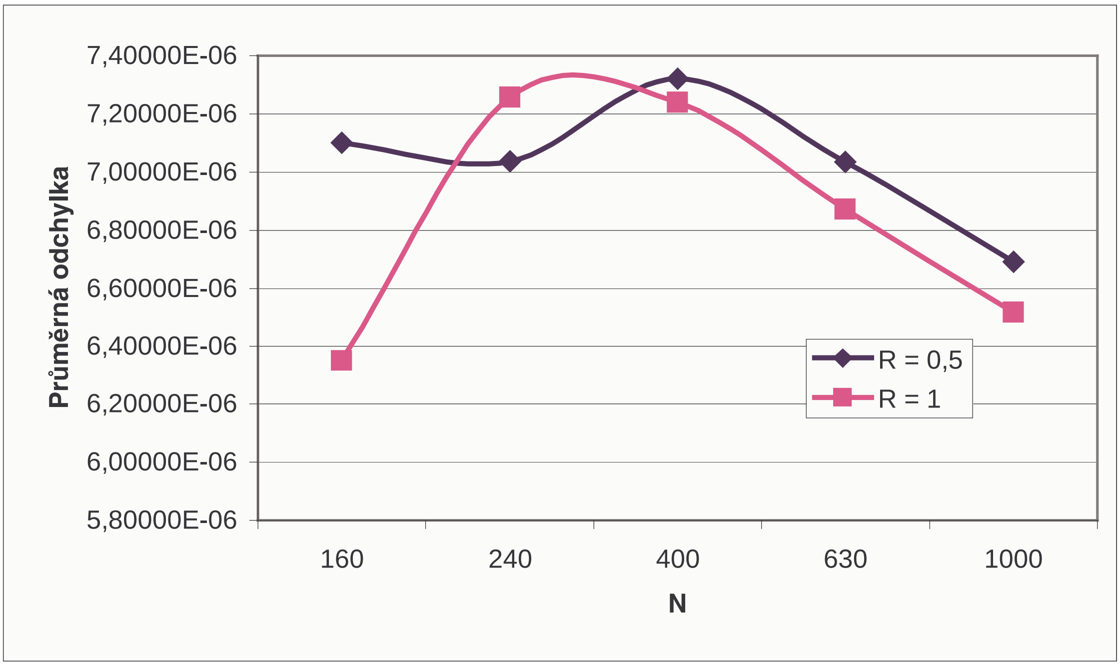
Graf 5.16: Porovnání kvality aproximace implicitní funkce pro normalizovaný a nenormalizovaný tvar funkce *Sphere* polygonizované metodou *Marching cubes*.

Z obou grafů vyplývá, že porovnávané metody aproximují povrch objektů se srovnatelnou odchylkou. Funkce, která není v normalizovaném tvaru je aproximována s větší chybou, neboť výpočet souřadnice každého bodu triangulace je zastaven dříve, něž je dosaženo požadované přesnosti $f(\mathbf{X}) < \varepsilon$ vlivem druhé mocniny návratové hodnoty funkce.

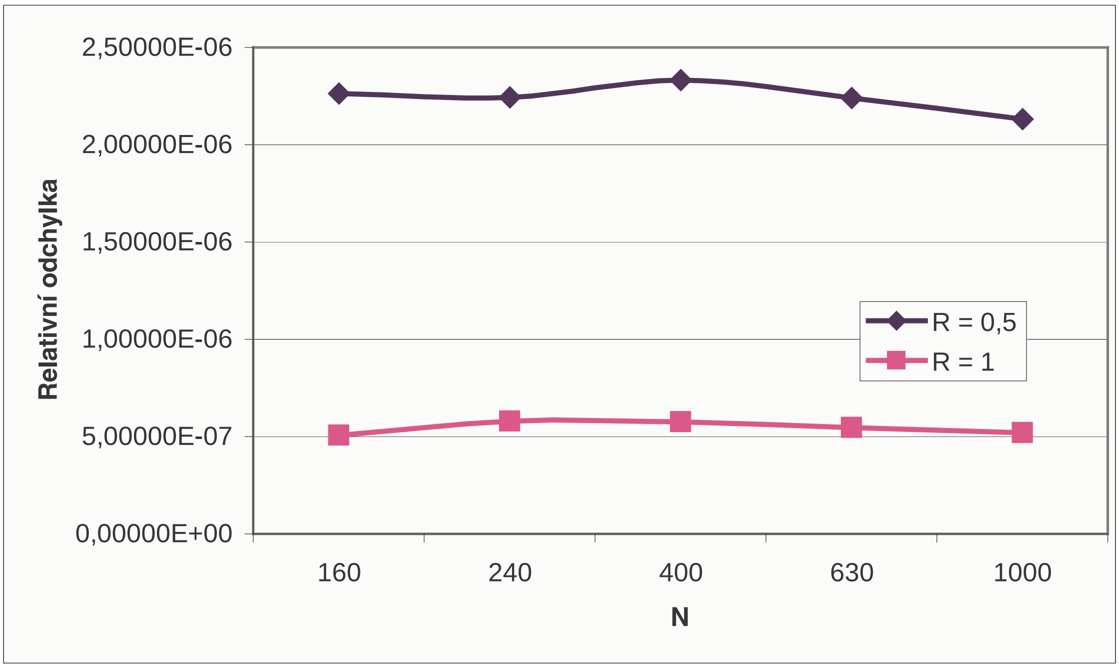
Nyní se budeme soustředit na porovnání vlivu velikosti měřeného objektu na velikost průměrné a relativní chyby. Naměřené hodnoty jsou uvedeny v tabulce 5.7.

	N	160	240	400	630	1000
R = 0,5	Triangles:	3323	7415	20423	50611	127257
	Vertices:	1663	3709	10213	25307	63630
	Time [ms]:	30	70	220	641	2464
	Total deviation:	0,01181	0,02610	0,07477	0,17803	0,42580
	Average deviation:	7,09962E-06	7,03777E-06	7,32089E-06	7,03468E-06	6,69178E-06
	Relative deviation:	2,26393E-06	2,24202E-06	2,33098E-06	2,23947E-06	2,13014E-06
	Surface area:	3,135971	3,139031	3,140695	3,141223	3,141474
R = 1	Triangles:	13141	29585	81217	202130	509851
	Vertices:	6572	14794	40610	101066	254927
	Time [ms]:	250	719	2828	10094	33422
	Total deviation:	0,04174	0,10736	0,29403	0,69446	1,66140
	Average deviation:	6,35088E-06	7,25686E-06	7,24040E-06	6,87139E-06	6,51715E-06
	Relative deviation:	5,05591E-07	5,77598E-07	5,76214E-07	5,46821E-07	5,18624E-07
	Surface area:	12,56129	12,56385	12,56548	12,56607	12,56623

Tabulka 5.10: Naměřené hodnoty odchylek aproximací funkce *Sphere* o poloměru 0,5 a 1 generované metodou *Marching triangles*.



Graf 5.17: Porovnání vlivu velikosti objektu *Sphere* na průměrnou chybu aproximace.



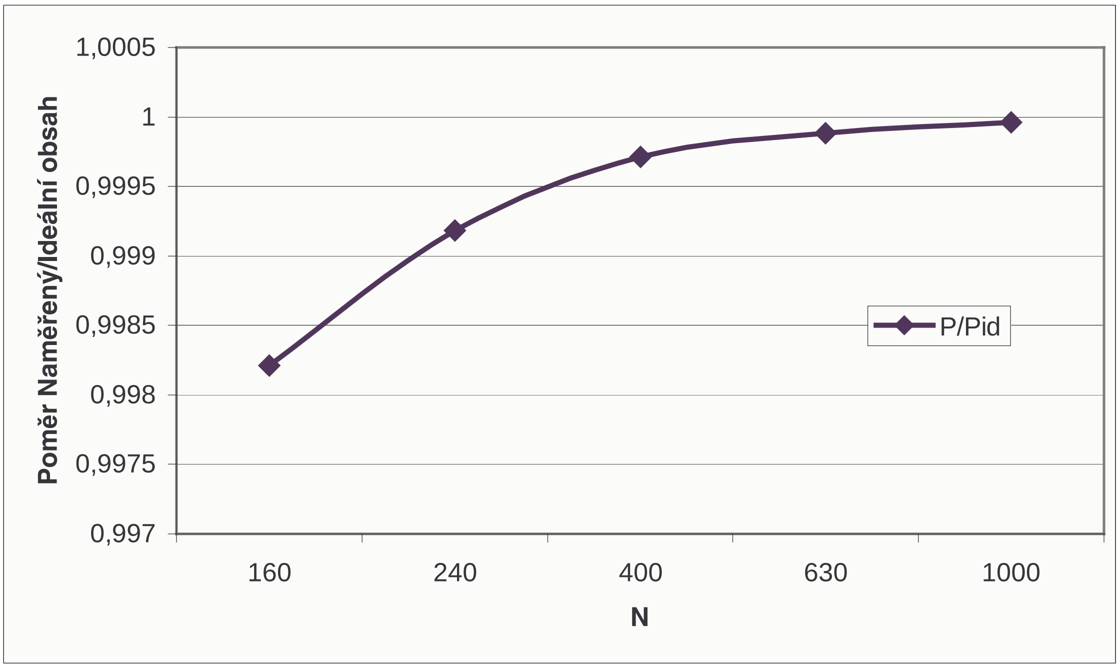
Graf 5.18: Porovnání vlivu velikosti objektu *Sphere* na relativní chybu aproximace.

Z grafů vyplývá, že průměrná chyba aproximace je pro obě velikosti objektu *Sphere* (koule) řádově srovnatelná, tj. neakceptuje vliv velikosti objektu na výslednou přesnost. Namísto toho relativní chyba aproximace je tím menší, čím je objekt větší, tj. průměrná chyba aproximace každého bodu je více zanedbatelná.

V následujícím grafu (5.19) budeme sledovat závislost celkového naměřeného obsahu *P* objektu *Sphere* (koule) s poloměrem *R = 0,5* na dělení prostoru. Sledován bude poměr

q = Naměřený obsah / *Ideální obsah*.

Ideální obsah, tj. obsah koule, je počítán podle vzorce $P = 4 \cdot \pi \cdot r^2$. Pro námi zvolený poloměr je tedy ideální obsah $P_{id} = 4 \cdot \pi \cdot 0.5^2 = \pi = 3{,}14159265358979323846264338327 95$.



Graf 5.19: Závislost poměru naměřeného obsahu objektu *Sphere* (*R=0,5*) k ideálnímu obsahu na dělení oblasti.

Z grafu vyplývá, že výpočet obsahu objektu je přesnější se vzrůstajícím počtem dělení oblasti. S rostoucím dělením oblasti roste i počet vygenerovaných bodů a trojúhelníků což má za následek zvýšení detailů na výsledném objektu. Takový objekt je pak aproximován přesněji, vzhledem k jeho matematickému modelu.

6 Závěr a zhodnocení

V diplomové práci byly popsány některé techniky pro modelování a pro zobrazování povrchu implicitně definovaných objektů. Byla podstatně urychlena stávající metoda *Marching triangles*, která začíná být dobrou alternativou známého algoritmu *Marching cubes*, ale která zatím dobře neřeší polygonizaci složitých a C^1 *nespojitých* objektů. Tento problém je způsoben numerickou nestabilitou výpočtu normálových a tečných vektorů v bodech nespojitosti a bude předmětem další práce.

Byl představen návrh paralelního algoritmu, který byl s úspěchem otestován na dvouprocesorovém počítači. Dosažené výsledky jsou velice uspokojivé, ale schází jim záruka, že se budou opakovat i na víceprocesorových strojích. Tento test bude proveden, jakmile bude k dispozici odpovídající hardwarové vybavení.

Dále byla navržena metoda, která jistým způsobem charakterizuje kvalitu aproximačního algoritmu a jejíž výsledky odpovídají průměrné, popř. relativní (vzhledem k velikosti objektu) odchylce polygonizovaného objektu od jeho matematického modelu.

6.1 Budoucí práce

Být předmětem dalšího úsilí si jistě zaslouží prezentovaný algoritmus Marching triangles. Kvalita výsledné sítě je velice dobrá, je však nutné nalézt numericky stabilnější cestu pro výpočet souřadnic a tečných vektorů.

Také paralelní část je možné rozšířit i do metod založených na počátečním bodě a tím ji podstatně urychlit. Jak bylo nastíněno v kapitole 4.4 hlavním problémem je nalezení oddělených částí objektů. Pokud by byly všechny části nalezeny, vytvoření paralelního algoritmu by nebyl tak velký problém. Zatím však je nalezení všech objektů ve scéně předmětem zkoumání.

Literatura

[Bloo88] Bloomenthal,J.: Polygonization of Implicit Surfaces, Computer Aided Geometric Design, 4(5):341-355, 1988.

[Bloo94] Bloomenthal,J.: Graphics Gems IV, Academic Press, 1994.

[Bloo95] Bloomenthal,J.: Skeletal Design of Natural Forms, Ph.D. Dissertation.

http://www.unchainedgeometry.com/jbloom/ … home pages.

[Blin82] Blinn,J.F.: Fractional Invisibility. Computer Graphics and Applications, November 1988.

[Cap99] Caprani,O., Hvidegaard,L., Mortensen,M., Schneider,T.: Robust and Efficient Ray Intersection of Implicit Surfaces. Aarhus University, Denmark 1999.

[Fran00] Franc,M.: Triangular Mesh Simplification Methods, Diplomová práce, Západočeská univerzita, Plzeň 2000.

[Hart98] Hartmann,E.: A marching method for the triangulation of surfaces, The Visual Computer (14), pp. 95-108, 1998.

[Krej00] Krejza,M.: Generating iso-surface from volumetric data, Diplomová práce, Západočeská univerzita, Plzeň 2000.

[Rac97] Ježek, K., Matějovic, P., Racek, S.: Paralelní architektury a programy, Vydavatelství ZČU, Plzeň 1997.

[Ross97] Rossignac,J.R., Requicha,A.A.G.: Solid modeling, 1997.

[Rou00] Roušal,M.: Prostředí pro modulární vizualizaci dat (MVE), Diplomová práce, Západočeská univerzita, Plzeň 2000.

[Rvachov] Rvachov,A.M.: Definition of R-functions.

http://www.mit.edu/~maratr/rvachev/p1.htm

[Sher98] Sherstyuk,A.: Convolution Surfaces in Computer Graphics, Ph.D. dissertation, 1998.

http://www.ugcs.caltech.edu/~andrei/thesis/

[Triq01] Triquet,F., Meseure,F., Chaillou,Ch.: Fast Polygonization of Implicit Surfaces, WSCG conference - report 162, West Bohemia University in Pilsen, 2001.

[Uhlir01] Uhlíř,J.: Interaktivní systém pro generování implicitních funkcí a jejich modelování, Diplomová práce, Západočeská univerzita, Plzeň 2001.

[Wyv88] Wyvill,B., Jevans,D.: Ray tracing implicit surfaces, Computer Science Technical Report, The University of Calgary, 1988.

http://pharos.cpsc.ucalgary.ca/Dienst/UI/2.0/Describe/ncstrl.ucalgary_cs/1988-292-04.

[Zar98a] Žára,J., Beneš,B., Felkel,P.: Moderní počítačová grafika, Computer Press, 1998.

Příloha A

User's guide

This document serves as a user guide for the MVE modules *Polygonizer* (`polygonizer.dll`), *Polyg_editor* (`polyg_editor.dll`) and the self-run application `MVE_Polygonizer.exe` that has been created for work without MVE.

Polygonizer module

The Polygonizer module is a part of MVE (*Modular Visualisation Environment*) system that is developed by Computer Graphics Group of West Bohemia University in Pilsen. The module is realized as DLL library `Polygonizer.dll` that is linked to MVE system by the standard interface.

The module input is an implicit function and module output is a triangle mesh. The basic scheme is in Figure A.1.

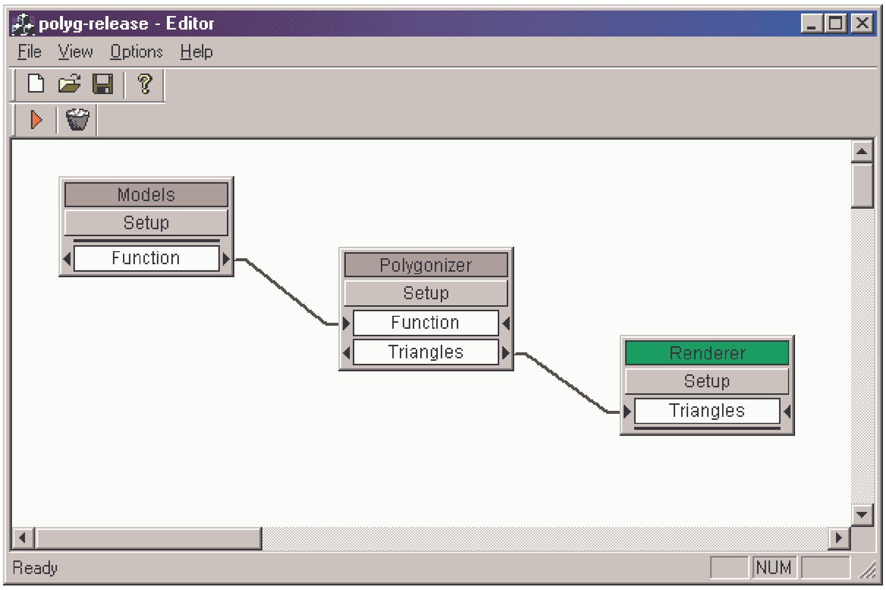


Figure A.1: The basic scheme for *Polygonizer* module in MVE.

The main *Polygonizer dialog* (Figure A.2) is visualized after clicking to *Setup* button.

The Polygonizer module is a triangle mesh generator. The triangle net can be directly visualized by *Renderer* module (mkrejza@students.zcu.cz) or it can be sent to other modules in MVE.

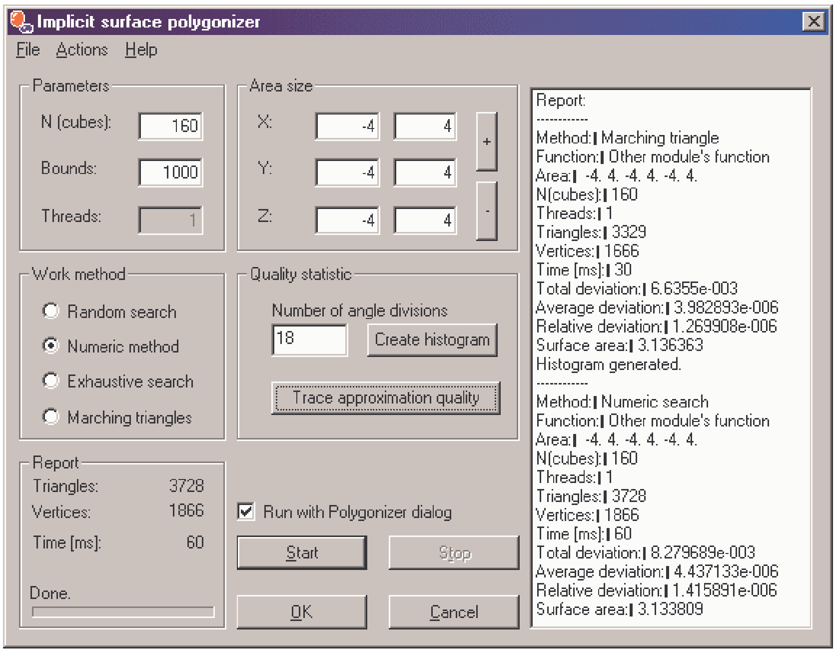


Figure A.2: Setup dialog of *Polygonizer* module.

Main menu

- **File**
  - *Save Triangles* – Saving a triangle net to disk in TRI format (triangle file – definition by Marek Krejza, mkrejza@students.zcu.cz).
  - *Save Results* – Saving the *Report* list box to disk in RSL text file.
  - *Save Histogram* – Saving *Histogram* to disk in HIST text file.
  - *Exit* – Closing dialog without saving of setup.
- **Actions** - *Start* – Starting of computation.
- **Help** - *About* – The basic information dialog about actual module’s version and author.

Buttons description

- **Parameters**
  - *N (cubes)* – Partitioning of defined area (*Area size*) in all axes (x, y, z).
  - *Bounds* – Bounding box, the maximal number of indexes of cells (cubes) in all axes.
  - *Threads* – The number of working threads in parallel computation (*Exhaustive search* method only).
- **Work method** – The implemented methods for visualization of implicit surfaces.
  - *Random search*
  - *Numeric method*
  - *Exhaustive search*
  - *Marching triangles*
- **Report** – Two boxes with actual information about computing (name of method and function, area size and its partitioning in axes, number of threads, number of triangles and vertices, time of computation).
- **Area size** – Size of area in 3D.
- **Quality statistic**
  - *Number of angle divisions* – Number of angle‘s intervals for histogram (π/n).
  - *Create histogram* – Creating of histogram of angle’s frequency in angle’s intervals.
  - *Trace approximation quality* – The detection of approximation quality. The results are written to *Report* list box.
- **Run with Polygonizer dialog** – Running of program with polygonizer setup dialog. It is advantageous for measuring of parameters of functions from other modules or for visualisation of suspended polygonization by *Stop* button.
- **Start** – Starting of computation.
- **Stop** – Stopping of computation.
- **OK** – Saving of module’s setup or sending of triangle net to the next MVE module[*].
- **Cancel** – Closing dialog without saving of setup.

---

[*] Only while running of module with check Run with polygonizer dialog.

Polyg_editor module

The *Polyg_editor* module is a simple module with some predefined functions only. This module collaborates with *MVE_Polygonizer* application and it can be used in MVE too. This module has no input and an implicit function is its output. The module's *Setup dialog* is in figure A.3.

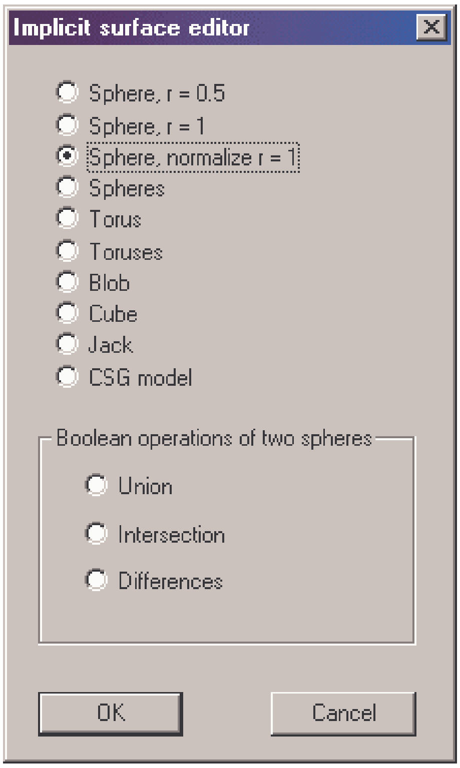


Figure A.3: The setup dialog of *Polyg_editor* module.

The application MVE_Polygonizer.exe

You do not have to use MVE to polygonization of implicit function and its visualization. You can use the `MVE_Polygonizer.exe` application which collaborate with DLL libraries `polyg_editor.dll`, `polygonizer.dll` , `triangle_modules.dll`. The library `triangle_modules.dll` contains the *Renderer* module for visualization of resultant triangle net.

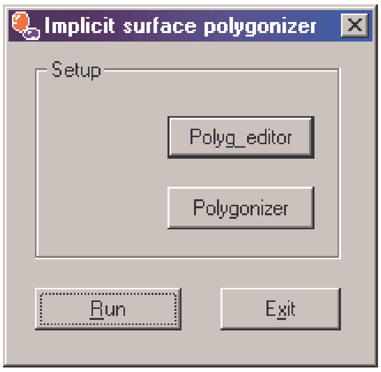


Figure A.4: The main dialog of application *MVE_Polygonizer.exe*.

Buttons description

The buttons of Setup area serves for setting of module‘s attributes. After click of them, the *Setup dialog* will appear.

The *Run* button starts computing and *Exit* button closes the application.

Příloha B

Installation guide

The *Polygonizer* module and the *Polyg_editor* module do not require an special installation. Just to copy these libraries to MVE modules directory where are stored the others member modules. See the MVE documentation [Rou00] for details.

Application MVE_Polygonizer.exe installation

The application uses functions from the `Polygonizer.dll`, `Polyg_editor.dll` and `Triangle_Modules.dll` dynamic linked libraries. Therefore, the application and these libraries have to be in the same directory. The MVE_Polygonizer is a Win32 application, it does work in Microsoft corporation's operating systems Win9x/Me/NT/2000.

Příloha C

Programmer's guide

The program is written in the C++ language with using the MFC library. The all dynamic linked libraries and the self-run application are built in Microsoft Visual C++ programming tool.

Polygonizer implemented model

- `Polygonizer.cpp`, `Polygonizer.h` – These files contain the basic functions which are required for integration to MVE (details in [Rou00]).
- `PolygonizerDlg.cpp`, `PolygonizerDlg.h` – The functions and classes which are necessary to displaying of dialog, module setup saving and starting of computation.
- `Functions.cpp`, `Functions.h` – contain the predefined implicit functions.
- `Implicit1.cpp`, `Implicit1.h` – implementation of *Marching cubes* method.
- `Implicit2.cpp`, `Implicit2.h` – implementation of *Exhaustive search* method.
- `Implicit3.cpp`, `Implicit3.h` – implementation of *Marching triangles* method.

Data types definitions files

- `MVE_Include.h` – contain data types which are required for integration to MVE.
- `Polyg_types.h` – data types of *Polygonizer* module.
- `Triangle_types.h` – data types for representation of triangle nets in MVE. (definition by Marek Krejza, mkrejza@students.zcu.cz).
- `Types.h` – The basic data types of MVE.

Common files of Visual C++ projects.

Data structures

CSG data structures

```
// pointer to implicit function with 3 coordinates (x, y, z)
typedef double (*PFunction)(double, double, double);
// CSG tree
typedef struct tcsg {
               PFunction function;    // implicit function pointer, =NULL if is not a leaf
               int operation;         // ID of operation, -1 leaf of tree, 0 union,
                                      // 1 intersection, 2 differences A-B, 3 differences B-A,
                                      // 4 symetric differences
               POINT1 move;           // vector of moving (x, y, z)
               POINT1 scale;          // scaling
               POINT1 rotate;         // rotation
               struct tcsg *left;     // left son
               struct tcsg *right;    // right son
} TCSG;
```

Module setup

```
typedef double (*PFunction)(double, double, double);
typedef struct tsetup {              // dat. structure for SETUP polygonizer
       PFunction function;           // pointer to implicit function
       double area[6];               // area size {xmin,xmax,ymin,ymax,zmin,zmax}
       int generated;                // indication of data generated
       int show_dialog;              // indication to view dialog during run application
       int bBox;                     // bounding box, bounds in dialog
       UINT methodID;                // ID of method (Start point, Exhaustive, ...)
       UINT funcID;                  // ID of function
       UINT nCubes;                  // N (cubes) in dialog
       UINT nThreads;                // number of threads
       UINT numberAngles;            // number of angle's intervals for histogram
} TSETUP;
```

Main process

```
typedef struct tmainprocess {
       TSETUP        *setup;         // module setup
       PROCESS       *p;             // method marching cubes
       PROCESS_TR    *ptr;           // method marching triangles
} TMainProcess;
```

Marching cubes and Exhaustive search

Both of these methods use the same data structures.

The main data structure.

```
typedef struct process {             // data structure for marching cubes
       PFunction     function;       // implicit function
       double        CubeSize;       // size of polygonization cell, cube size
       double        delta;          // delta for computing normal vector
       int           bBox;           // bounding box
       POINT1        start;          // start point on surface
       VERTICES      vertices;       // array of points
       TTRIANGLES    triangles;      // array of triangles
       CUBES         *cubes;         // list of active cubes for polygonization process
       CENTERLIST    **centers;      // HASH table for indexes of cubes
       CORNERLIST    **corners;      // HASH table for indexes of corners
       EDGELIST      **edges;        // HASH table for indexes of edges
} PROCESS;
```

Vertex data structure.

```
typedef struct vertex {              // surface vertex
       POINT1 position;              // coordinates
       POINT1 normal;                // surface normal
} VERTEX;
```

Array of vertices.

```
typedef struct vertices {            // list of vertices in polygonization
       int count, max;               // # vertices, max # allowed
       VERTEX *ptr;                  // dynamically allocated
} VERTICES;
```

Triangle data structure.

```
typedef struct Ttriangle {           // triangle
       int i1, i2, i3;               // indexes to vertices array
} TTRIANGLE;
```

Array of triangles.

```
typedef struct Ttriangles {          // list of triangles
       int count, max;               // # triangles, max # allowed
       TTRIANGLE *ptr;               // dynamically allocated
} TTRIANGLES;
```

Coordinates definition in 3D.

```
typedef struct point1 {              // a three-dimensional point or vector
       double x, y, z;               // its coordinates
} POINT1;
```

Data structure of polygonizing cell.

```
typedef struct cube {                 // partitioning cell (cube)
       int i, j, k;                   // lattice location of cube
       CORNER *corners[8];            // eight corners
} CUBE;

typedef struct corner {               // corner of a cube
       int i, j, k;                   // (i, j, k) is index within lattice
       double x, y, z, value;         // location and function value
} CORNER;
```

The list of active cubes.

```
typedef struct cubes {                // linked list of cubes acting as stack
       CUBE cube;                     // a single cube
       struct cubes *next;            // remaining elements
} CUBES;
```

HASH tables.

```
typedef struct centerlist {           // list of cube locations
       int i, j, k;                   // cube location (index)
       struct centerlist *next;       // remaining elements
} CENTERLIST;

typedef struct cornerlist {           // list of corners
       int i, j, k;                   // corner id
       double value;                  // corner value
       struct cornerlist *next;       // remaining elements
} CORNERLIST;

typedef struct edgelist {             // list of edges
       int i1, j1, k1, i2, j2, k2;    //  corner ids
       int vid;                       // vertex id
       struct edgelist *next;         // remaining elements
} EDGELIST;
```

CubeTable, a table that contains one entry for each of the possible configurations of the cell vertex polarities, it is a 256 entry table.

```
static INTLISTS *cubetable[256];      // 256 pointers to INTLIST

typedef struct intlist {              // list of integers
       int i;                         // an integer
       struct intlist *next;          // remaining elements
} INTLIST;

typedef struct intlists {             // list of list of integers
       INTLIST *list;                 // a list of integers
       struct intlists *next;         // remaining elements
} INTLISTS;
```

Marching triangles

The main data structure.

```
typedef struct process_tr {          // data structure for Marching triangles
       PFunction function;           // pointer to implicit function
       double dt;                    // general length of edge of triangles
       double dt2;                   // dt*dt for faster comparison without sqrt function
       double delta;                 // delta for computing normal vector
       TPOINTS points;               // array of points
       TTRIANGLES triangles;         // array of triangles
       TFPOLYGONS *front_polygon;    // list of front polygons
       TPointAreaLists *hash_table;  // HASH table for speed up control distances
} PROCESS_TR;
```

Vertex data structure.

```
typedef struct point_tr {            // surface point for marching triangles
       POINT1  coord;                // coordinates
       POINT1  n;                    // surface normal
       POINT1  t1;                   // 1. tangent vector
       POINT1  t2;                   // 2. tangent vector
       double  front_angle;          // actual front angle
       int     angle_changed;        // indication of change angle
       int     border_point;         // indication of point in border
       int     out_point;            // out of bounding box
} POINT_TR;
```

Array of vertices.

```
typedef struct tpoints {             // list of vertices in polygonization
       int count, max;               // # vertices, max # allowed
       POINT_TR *ptr;                // dynamically allocated
} TPOINTS;
```

Front polygon data structure.

```
typedef struct tfpolygons {          // front polygon
       int count;                    // # vertices in front polygon
       TLIST *ip;                    // list of vertices in front polygon
       struct tfpolygons *next;      // pointer to next front polygon
} TFPOLYGONS;

typedef struct tlist {               // list of vertices in front polygon
       int index;                    // index of point in point array
       struct tlist *left;           // left neighbour
       struct tlist *right;          // right neighbour
       struct pointarealist *p_element_hash_table; // pointer to the same point in HASH table
} TLIST;
```

HASH table.

```
typedef struct pointarealists {      // HASH table index
       int count;                    // # vertices in every HASH index
       TPointAreaList *list;         // list of vertices in HASH table index
} TPointAreaLists;

typedef struct pointarealist {       // list of vertices in HASH table
       int indexp;                   // index of point in front polygon
       TLIST *ip;                    // pointer to the same point in front polygon
       TFPOLYGONS *front_polygon;    // pointer to it's front polygon
       struct pointarealist *left;   // pointer to left point in the same area
       struct pointarealist *right;  // pointer to right point in the same area
} TPointAreaList;
```

Functions description

HASH function

The all implemented algorithms use this HASH function for their speedup. The hash index computation is described as C++ macro (viz [Bloo94]).

```
#define HASHBIT     (6)
#define HASHSIZE    (size_t)(1<<(3*HASHBIT))        /* hash table size (262144) */
#define MASK        ((1<<HASHBIT)-1)                // 00111111
#define HASH(i,j,k) ((( ( ((i)&MASK)<<HASHBIT)|((j)&MASK))<<HASHBIT)|((k)&MASK))
```

The i, j, k numbers are integer indexes in x, y, z axes. The hash index is in Figure A.5.

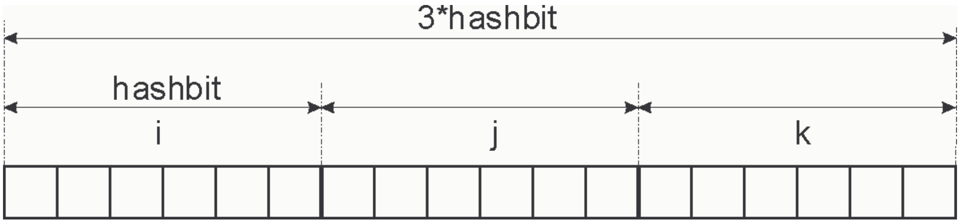


Figure A.5: The resulting hash index for i, j, k indexes in x, y, z axes.

Note:

In the Marching triangles method, the indexes i, j, k are computed from coordinates of point $\boldsymbol{p}$, as:

```
i = (int)(p.x / dt) - middle[0];
j = (int)(p.y / dt) - middle[1];
k = (int)(p.z / dt) - middle[2];
```

where $dt = cubesize = \delta_t$ and middle is array which contents indexes of start point.