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 ......................................................................................................................... 1
  1.1 Implicitní plochy ................................................................................................ 1
  1.2 Abstract.............................................................................................................. 2
  1.3 Rozvržení dokumentu.......................................................................................... 2

2 Modelovací techniky ..................................................................................3
  2.1 Konstruktivní geometrie těles ........................................................................... 3
  2.2 Funkcionální modelování .................................................................................. 4
  2.3 Modelování založené na kostře ....................................................................... 5

3 Zobrazování povrchu implicitních funkcí ................................................. 7
  3.1 Problém nalezení kořene rovnice ..................................................................... 7
  3.2 Polygonizační algoritmy ................................................................................... 7
    3.2.1 Metody založené na počátečním bodě ......................................................... 8
    3.2.2 Úplné prohledávání definované oblasti ....................................................... 9
    3.2.3 Chyba aproximace implicitního povrchu ................................................... 9
  3.3 Ray tracing pro implicitní funkce .................................................................... 11

4 Implementace .................................................................................................. 13
  4.1 Datová struktura CSG stromu ........................................................................ 13
  4.2 Nalezení počátečního bodu ............................................................................ 13
    4.2.1 Numerické vyhledávání .............................................................................. 14
    4.2.2 Algoritmus Random search ...................................................................... 15
  4.3 Exhaustive search ............................................................................................. 15
  4.4 Paralelní výpočet ............................................................................................. 16
  4.5 Algoritmus Marching cubes ............................................................................ 18
    4.5.1 Detekce buněk protnutých povrchem funkce ............................................. 19
    4.5.2 Polygonizace buňky .................................................................................. 20
    4.5.3 Sestavení indexu a nalezení protnutých hran .......................................... 20
    4.5.4 Výpočet souřadnic vrcholů trojúhelníků ................................................... 21
    4.5.5 Výpočet normál ve vrcholech ................................................................... 21
1 Úvod

Většina programů umožňujících modelování 3D těles, využívá pro jejich reprezentaci jednoduchých primitiv jako je přímka, plocha, kvádř, 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 takovými 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.


1.1 Implicitní plochy

*Implicitní plocha* je množina bodů \(p\), pro které platí \(f(p) = 0\), kde \(p \in \mathbb{R}^3\), 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 \(p\), pro které platí \(f(p) = c\), 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 \(f(p) < 0\), což označuje množství bodů (objem prostoru) uzavřených hranicí (povrchem funkce) \(f(p) = 0\). 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 \( f(p) \geq 0 \). 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 \( c = (c_x, c_y, c_z) \) a poloměrem \( r \). Tato funkce může být vyjádřena parametricky jako množina bodů \( \{P\} \), pro které platí:

\[
(p_x - c_x)^2 + (p_y - c_y)^2 + (p_z - c_z)^2 - r^2 = 0.
\] (1.2)

\[
(p_x, p_y, p_z) = (c_x, c_y, c_z) + (r \cos \beta \cos \alpha, r \cos \beta \sin \alpha, r \sin \beta).
\] (1.1)

\[\text{1.2 Abstract}\]

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.

\[\text{1.3 Rozvržení dokumentu}\]

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 budoucní práce. Sedmou kapitolou je seznam použitých zdrojů a literatury.
2 Modelovací techniky

V této kapitole budou vysvěteny 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 odraží 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.

Vnitřní uzly CSG stromu obsahují logické operace a v listech jsou uchovávány informace o CSG primitivech. Prostorové transformace (rotace, posun, změna měřítka) mohou být chápany 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.

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

- sjednocení (union) \[ \min\{ f_1(p), f_2(p) \}, \]
- průnik (intersection) \[ \max\{ f_1(p), f_2(p) \}, \]
- rozdíl (differences) \[ \max\{ f_1(p), -f_2(p) \}. \]

Poznámka:

Uvedené logické operace jsou platné pro definici implicitních funkcí podle [Blo95], tj. $f(P) < 0$. Pokud je funkce definována v F-Rep (viz následující kapitola), jsou operace $\min/\max$ prohodíeny.

### 2.2 Funkcionální modelování

Funkcionální reprezentace, zkráceně F-rep (Function Representation), definuje celý geometrický objekt pomocí jedné spojité reálné funkce několika proměnných $f(p) \geq 0$ (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. 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 (skeleton based implicits [Blo95], 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 $C^0$ 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 $C^0$ 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.
Nej jednodušší operace mezi dvěmi implicitními funkcemi \( f_1 \) a \( f_2 \) vypadají takto:

- sjednocení (union) \( f_1 \mid f_2 = f_1 + f_2 + \sqrt{f_1^2 + f_2^2} \), (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 \&, |, ~, \setminus, \text{ 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 deformační 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, elipsoid), 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 přičinou 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 \( f(p) = 0 \), tj. nalezení takového bodu \( p \), který rovnici splňuje. Zpravidla nelze z rovnice explicitně vyjádřit jednu ze souřadnic, jejíž hodnoty by mohly být vypočítávány standardními postupy. Souřadnice bodu \( p \) musí být proto voleny s využitím prostředků numerické matematiky. Problém nalezení kořene rovnice potom spočívá v hledání minima funkce \( f \) vhodnou volbou souřadnic bodu \( p \).

V následujících odstavcích budou přestaveny možné metody, které vedou k výsledkům, ale jejichž použití je vždy jistým způsobem omezeno, buďťo rychlosti 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 známé 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 znárnou 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 [Trij01] využívají znalost 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 \( \Omega \) 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.
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 implicitného povrchu, tj. z rovnice \( f(X) = 0 \). Máme-li vygenerovan polygonální objekt, jehož všechny body leží přesně na povrchu implicitní funkce, musí pro součet funkčních hodnot ve všech těchto bodech platit:

\[
E_{\text{absolute}} = \sum_{i=0}^{N-1} | f(x_i) | = 0, \tag{3.4}
\]

kde \( N \) je celkový počet vygenerovaných bodů a \( x \) 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:

\[
E_{\text{average}} = \frac{\sum_{i=0}^{N-1} | f(x_i) |}{N} \tag{3.5}
\]

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:

\[
E_{\text{relative}} = \frac{E_{\text{average}}}{P} \tag{3.6}, \quad \text{kde } P \text{ je celkový obsah polygonálního objektu, který je počítán podle vzorce (3.7).}
\]

\[
P = \sum_{i=0}^{M} P_i = \frac{1}{2} \sum_{i=0}^{M} | \mathbf{u}_i \times \mathbf{v}_i | \tag{3.7}, \quad \text{kde } M \text{ je celkový počet trojúhelníků, } P_i \text{ je obsah jednotlivých trojúhelníků a } \mathbf{u}_i \text{ a } \mathbf{v}_i \text{ je dvojice vektorů vycházejících ze společného bodu v každém trojúhelníku.}
\]

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.](image)

**Algoritmus**

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

\[
r(t) = a + t \cdot b,
\]

kde \( a \) je zdroj paprsku, \( b \) je směrový vektor a \( t \) je parametr.

Pro hledaný průsečík s povrchem funkce platí rovnice:

\[
f(r) = f(a, b, t) = 0,
\]

zkráceně

\[
f(t) = 0.
\]

Pro každý paprsek je prováděn test, zda nedošlo k průsečíku s tělesem v zobrazované oblasti. Pokud k průsečíku nedošlo, je píxel 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 funkcii \( f(t) \) 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

Jedným 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 primitivy.

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 primitivy a každý z nich obsahuje trojici vektorů představující prostorové transformace posunu, rotace a změny měřítky. 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 primitivy. 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 funkci. 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 vnitř objektu a druhý vně. Říďme-li se definicí implicitních objektů podle [Bloo95], tak pro vnitřní bod platí $f(p_{in}) < 0$ a pro vnější bod $f(p_{out}) > 0$. Uvažujeme-li $F$-$Rep$, jsou uvedené závislosti obrácené, tj. $f(p_{in}) > 0$ a $f(p_{out}) < 0$.

Obsahuje-li scéna více oddělených objektů, metoda nalezení nejblížší z nich vzhledem k náhodně zvolenému bodu $S$ uvnitř definované oblasti $\Omega$ (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 přibliží o $\delta$ ve všech třech osách.

Algoritmus: (4.1)

1. Náhodná volba souřadnic bodu $S$ uvnitř definované oblasti $\Omega$, $A$, $B$ jsou pomocné body, velikost proměnné $\delta$ je úměrná velikosti rastru ($cubsize$, viz obrázek 4.2).

2. Opakuj, dokud neplatí, že $f(A)^* f(B) < 0$.
   a. $A(x, y, z) = S(x-\delta, y, z)$, $B(x, y, z) = S(x+\delta, y, z)$, test podmínky $f(A)^* f(B) < 0$, pokud platí $|f(A)| < |f(B)|$ tak $S = A$, jinak $S = B$.
   b. $A(x, y, z) = S(x, y-\delta, z)$, $B(x, y, z) = S(x, y+\delta, z)$, test podmínky $f(A)^* f(B) < 0$, pokud platí $|f(A)| < |f(B)|$ tak $S = A$, jinak $S = B$.
   c. $A(x, y, z) = S(x, y, z-\delta)$, $B(x, y, z) = S(x, y, z+\delta)$, test podmínky $f(A)^* f(B) < 0$, pokud platí $|f(A)| < |f(B)|$ tak $S = A$, jinak $S = B$.

3. Platí-li $f(A) < 0$, tak $p_{in} = A$, $p_{out} = B$; jinak $p_{in} = B$, $p_{out} = A$.

4. Přesné souřadnice počátečního bodu na povrchu funkce jsou vypočítány numerickou metodou půlení intervalu mezi body $p_{in}$, $p_{out}$.

14
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 [Blo094]).

\textbf{Algoritmus:} (4.2)

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

Poznámka:

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

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 $\Omega$, indexy $(i, j, k)$; $i, j, k = 1, 2, \ldots N-1$.
   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 $f(x_i, y_j, z_k) \neq f(x_{i+1}, y_j, z_k) < 0$, 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 $f(x_i, y_j, z_k) \neq f(x_i, y_{j+1}, z_k) < 0$, 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 $f(x_i, y_j, z_k) \neq f(x_{i}, y_{j}, z_{k+1}) < 0$, 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 navrhnut paralelizaci zvoleného řešení. Metody založené na počátečním bodě by byly paralelizovatelem snadno pro implicitní funkce, které definují více oddělených objektů. Každý takový objekt by mohl být polygonizován samostatným výpočtovým 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 $\Omega$ rozdělena na tok (část, 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 kontrolu znamének funkčních hodnot v rozích buněk (krychlí) a případné polygonizace buněk spotřebují relativně málo času. Tak je zajištěno, že procesory jsou vytištěny přibližně rovnoměrně i pokud je většina objektů ve scéne umístěna v oblasti jednoho procesoru. Tento předpoklad potvrzuje dosažené výsledky v kapitole 5.2.3.

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

Slab je plátek prostoru, který má ve dvou osách stejný rozměr jako oblast \( \Omega \) a ve třetí ose je široký jako rozměr buněk (cubsize).ZNázorněno na obrázku 4.2.

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

\[
\text{Algoritmus:} \quad (4.4)
\]

Mějme oblast \( \Omega \) rozdělenou na \( N \) slabů a k dispozici \( n_p \) volných procesorů.

1. Vypočítejme \( n_s = \text{trunc} \frac{N}{n_p}, n_c = \text{mod} \frac{N}{n_p} \).

2. Prvních \( n_c \) vláken bude polygonizovat \((n_s+I)\) slabů, ostatní vlákna budou polygonizovat \( n_s \) slabů.

Poznámka:

Operace \( \text{trunc} \) představuje převod reálného čísla na celé, odcísnutím 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 napojovalení víc trojúhelníkových sítí.

Naproti tomu, práce vláken nad stejnými datovými strukturami je příčinou vzniku kritických secí, 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 na trojúhelníkové síť. Použitelnost metody je nadále rozšířena 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].


**Algoritmus:**

1. Umístě střed první buňky (krychele, *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 prázdnný.
   a. Polygonizuj aktuální buňku a vyřaď ji ze seznamu aktivních buněk.
   b. Vypočítej funkční hodnoty v rozích sousedních⁶ buněk a vlož do seznamu aktivních buněk ty, u kterých se liší hodnoty znaménka.
   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: 

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 \( f(e) > 0 \),
- „0“, pokud platí \( f(e) < 0 \), kde \( e \) 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 (±ε).

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 jednotkové normálového vektoru ve vrcholech je prováděn podle vzorce:

\[
\mathbf{n} = \frac{\nabla f(v)}{\left\| \nabla f(v) \right\|},
\]

kde \(v\) je příslušný vrchol a \(\nabla f\) 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:

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

Asymetrická diference:

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

dle hodnoty \(\delta\) je úměrná rozměru buňky (*cubsize*) a platí relace \(\delta \ll \text{cubsize}\).
4.6 Algoritmus Marching triangles

Tato metoda patří mezi nové metody generující polygonální síť z implicitních funkcí, a proto ji budeme věnovat větší pozornost. Algoritmus Marching triangles přistupuje k polygonizaci implicitní funkce zejména 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 \( p \) na povrchu objektu definovaného implicitní funkcí, k výpočtu jeho normálového vektoru \( n \) a dvou tečných vektorů \( t_1 \) a \( t_2 \). Trojice ortogonálních vektorů \( (t_1, t_2, n) \) představuje lokální souřadnicový systém bodu \( p \).

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

Algoritmus:

\begin{align*}
1. & \quad \text{Inicializuj pomocný bod } u_0 = q. \\
2. & \quad \text{Opakuj, dokud není } f(u_{k+1}) < \varepsilon \text{ nebo } f(u_k) \cdot f(u_{k+1}) < 0. \\
& \quad \quad - u_{k+1} = u_k - \frac{f(u_k)}{\nabla f(u_k)^2} \nabla f(u_k). \\
& \quad \quad - u_k = u_{k+1}. \\
3. & \quad \text{Pokud platí, že } f(u_k) \cdot f(u_{k+1}) < 0, \text{ tak použij metodu půlení intervalu dokud neplatí } f(u_{k+1}) < \varepsilon. \\
4. & \quad \text{Bod na povrchu } p = u_{k+1}.
\end{align*}
5. Jednotkový normálový vektor bodu $p$,

$$\mathbf{n} = \frac{\nabla f(p)}{\|\nabla f(p)\|}.$$ 

6. Tečné vektory $t_1$ a $t_2$ lze určit:

- pokud $n_x > 0.5$ nebo $n_y > 0.5$, potom $t_1 = (n_y, -n_x, 0)$, jinak $t_1 = (-n_x, 0, n_y)$,
- $t_2 = n \times t_1$, kde $n = (n_x, n_y, n_z)$.

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].

4.6.3 Originální algoritmus Marching triangles

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

Algoritmus:

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

\[
q_{i+2} = p_1 + \delta_i \cos(i\pi/3)t_{11} + \delta_i \sin(i\pi/3)t_{12}, i = 0,\ldots,5.
\]

Pro každý z bodů \( q_2 \ldots q_7 \) je volána procedura \( Surface\ point \), která vypočítá odpovídající body \( p_2 \ldots p_7 \), které leží na povrchu objektu. Trojúhelníky na povrchu hexagonu jsou prvními šesti trojúhelníky triangulace \((p_1, p_2, p_3), (p_1, p_3, p_4), (p_1, p_4, p_5), (p_1, p_5, p_6), (p_1, p_6, p_7), (p_1, p_7, p_2)\). Seřazené pole bodů \( p_2 \ldots p_7 \) se nazývá \( Actual\ front\ polygon \( \Pi_0 \). První šestiúhelník je znázorněn na obrázku 4.4.

Obrázek 4.4: První šestiúhelník se zvýrazněným front úhlem u bodu \( p_5 \).

2. Pro každý bod aktuálního front polygonu \( \Pi_0 \) určíme úhel oblasti, která se bude triangulovat. Tyto úhly nazveme \( Front\ angles \) (znázorněn na obrázku 4.4 u bodu \( p_5 \)). Dále určíme sousedy bodu \( p_0 \).

- Nechť bod \( v_1 \) je levým sousedem (na obrázku 4.4 je levým sousedem bodu \( p_5 \) bod \( p_4 \)), který má v lokálním souřadnicovém systému bodu \( p_{0i} (t_{i1}, t_{i2}, n_i) \) souřadnice \((\xi_i, \eta_i, \zeta_i)\) a

- bod \( v_2 \) pravým sousedem se souřadnicemi \((\xi_2, \eta_2, \zeta_2)\).

\(^7\) Tvar šestiúhelníka je volen proto, aby vzniklá polygonální síť byla tvořena trojúhelníky, jejichž průměrný vnitřní úhel je \( 60^\circ \), tj. \( \pi/3 \). Takové trojúhelníky jsou přibližně rovnostranné a výsledná síť je kvalitnější.
Potom $\omega_1 = \text{polární úhel souřadnic } (\xi_1, \eta_1)$ a $\omega_2 = \text{polární úhel souřadnic } (\xi_2, \eta_2)$.

Výsledný úhel je určen jako:

\[
\text{Front angle} = \omega_1 - \omega_2, \text{ pokud } \omega_1 > \omega_2, \text{ jinak } \text{Front angle} = \omega_1 - \omega_2 + 2\pi.
\]

3. Test který zabraňuje překrývání nových trojúhelníků s již triangulovanou částí.
   - Testujeme vzdálenost dvojic bodů aktuálního front polygonu $P_0$.
   - Testujeme vzdálenost bodů aktuálního front polygonu $P_0$ s body ostatních front polygonů $P_k$, 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 $p_{0m}$ je bod aktuálního front polygonu $P_0$ s minimálním front úhlem $\omega$. Triangulace okolí bodu $p_{0m}$ se skládá z následujících kroků.
   - Určení levého $v_1$ a pravého $v_2$ souseda bodu $p_{0m}$.
   - Určení počtu trojúhelníků $n_i$, které budou generovány, podle vzorce:
     \[
     n_i = \text{trunc} \left( \frac{3\omega}{\pi} \right) + 1, \quad \Delta \omega = \frac{\omega}{n_i}.
     \]  
     (4.6)
   - Oprava hodnoty $\Delta \omega$ pro extrémní případy, které jsou znázorněny na obrázku 4.5, podle následujících pravidel.
     - Pokud je $\Delta \omega < 0,8$ a $n_i > 1$, potom $n_i = n_i - 1, \Delta \omega = \frac{\omega}{n_i}$. Obrázek 4.5a.
     - Pokud je $n_i = 1$ a $\Delta \omega > 0,8$ a $\left\| v_1 - v_2 \right\| > 1,2 \delta_i$, potom $n_i = 2, \Delta \omega = \frac{\omega}{2}$. Obrázek 4.5b.
     - Pokud je $\omega < 3$ a \( \left\| v_1 - p_{0m} \right\| \leq 0,5 \delta_i \) nebo \( \left\| v_2 - p_{0m} \right\| \leq 0,5 \delta_i \), potom $n_i = 1$. Obrázek 4.5c.
Obrázek 4.5: Oprava hodnot $\Delta \omega$ pro extrémní případy.

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

$$b = p_{0m} + \delta_i (q_0 - p_{0m}) / \left\| q_0 - p_{0m} \right\|. \quad (4.7)$$

Obrázek 4.6: Triangulace okolí bodu $p_5$.

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

Z nových bodů vzniknou následující trojúhelníky:

$$(v_1, p_{N+1}, p_{0m}), (p_{N+1}, p_{N+2}, p_{0m}), \ldots, (p_{N+n_t-1}, v_2, p_{0m})$$
5. Aktualizace aktuálního front polygonu.
- Vyřazení bodu $p_{0m}$ z aktuálního front polygonu.
- Pokud je $n_t > 1$, vlož do aktuálního front polygonu body $p_{N+1}, \ldots, p_{N+nt-1}$.
- Nastav příznak $angle\_changed$ u bodů $v_1, v_2, p_{N+1}, \ldots, p_{N+nt-1}$ z důvodu aktualizace jejich front angle.

6. Opakuj kroky 2-5, dokud aktuální front polygon $\Pi_0$ neobsahuje poslední 3 body, které společně tvoří trojúhelník. Pokud existuje nějaký další (neprázdny) front polygon, stane se novým aktuálním front polygonem $\Pi_0$ 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. Používaný jsou dva následující testy:

1. Test vzdálenosti dvojic bodů aktuálního front polygonu $\Pi_0$.
2. Test vzdálenosti bodů aktuálního front polygonu $\Pi_0$ s body ostatních front polygonů $\Pi_k$, 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 \( p_{0i}, p_{0j} \in \mathcal{P}_0 \). Pokud existuje bod \( p_{0j} \), který není pro bod \( p_{0i} \) přímým sousedem ani sousedem souseda a platí:

\[
\| p_{0i} - p_{0j} \| < \delta, \quad \text{kde} \quad \delta \quad \text{představuje průměrnou délku hran vznikajících trojúhelníků.}
\]

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

- Nový aktuální front polygon \( (p_{01}, \ldots, p_{0i}, p_{0j}, \ldots, p_{0N}) \) s novým počtem bodů
  \[ N_{\text{new}} = N - (j-i-1). \]

- Zbývající front polygon \( (p_{0i}, \ldots, p_{0j}) \) s počtem bodů \( N_{\text{new}} = j-i+1 \).

Body \( p_{0i}, p_{0j} \) nesmí být zúčastněny dalších testů vzdálenosti, musí být nastaven jejich příznak \text{border_point}.

2. test

Testujeme vzdálenost bodů aktuálního front polygonu \( \mathcal{P}_0 \) s body ostatních front polygonů \( \mathcal{P}_k, \ k > 0 \). Pokud existují body \( p_{0i} \in \mathcal{P}_0 \) a \( p_{mj} \in \mathcal{P}_m \), pro které platí:

\[
\| p_{0i} - p_{mj} \| < \delta, \]

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

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

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

Body \( p_{0i}, p_{mj} \) nesmí být zúčastněny dalších testů vzdálenosti, musí být nastaven jejich příznak \text{border_point}.

Poznámka:

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

\[ O(n(n+1)/2) \]

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 složitost \( O(n \cdot m) \), 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 \( \delta \) (průměrná délka hran trojúhelníků) a platí, že \( n_{pf} \ll n_p \), 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 přiši 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 složitostí $O(n)$, 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. Přičinou jsou opět zmíňované 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 polygyony, 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 $t_1$ a $t_2$, což by mohlo mít za následek obrázení směru polygonizace zpět do triangulované části.

Popisovaná situace může nastat v bodech $C^i$ nespojitostí, 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 $\delta$ (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.
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**


### 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 určení 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 hodnotou (8 byte).

Metoda *Marching triangles* využívá stejné datové struktury pro trojúhelník, ale odlišné pro vrchol, neboť využívá 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 bodů – AngleChanged, BorderPoint a OutPoint (3*4 Byte). Pro určení 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 Uhlře (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.

<table>
<thead>
<tr>
<th>Funkce</th>
<th>Metoda</th>
<th>N</th>
<th>160</th>
<th>240</th>
<th>400</th>
<th>630</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Triangles:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sphere</td>
<td>Marching Cubes</td>
<td>3728</td>
<td>8432</td>
<td>23408</td>
<td>58472</td>
<td>147368</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vertices:</td>
<td>1866</td>
<td>4218</td>
<td>11706</td>
<td>29238</td>
<td>73688</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Time [ms]:</td>
<td>94</td>
<td>156</td>
<td>375</td>
<td>937</td>
<td>2266</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Triangles:</td>
<td>3291</td>
<td>7445</td>
<td>20567</td>
<td>50496</td>
<td>12713</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vertices:</td>
<td>1647</td>
<td>3724</td>
<td>10285</td>
<td>25249</td>
<td>63567</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Time [ms]:</td>
<td>47</td>
<td>141</td>
<td>500</td>
<td>1562</td>
<td>5110</td>
<td></td>
</tr>
<tr>
<td>Torus</td>
<td>Marching Cubes</td>
<td>520</td>
<td>1280</td>
<td>3568</td>
<td>8352</td>
<td>22912</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vertices:</td>
<td>256</td>
<td>640</td>
<td>1784</td>
<td>4176</td>
<td>11456</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Time [ms]:</td>
<td>63</td>
<td>62</td>
<td>93</td>
<td>172</td>
<td>562</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Triangles:</td>
<td>515</td>
<td>1159</td>
<td>3229</td>
<td>7891</td>
<td>20025</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vertices:</td>
<td>258</td>
<td>579</td>
<td>1615</td>
<td>3945</td>
<td>10012</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Time [ms]:</td>
<td>15</td>
<td>31</td>
<td>78</td>
<td>172</td>
<td>500</td>
<td></td>
</tr>
<tr>
<td>Blob</td>
<td>Marching Cubes</td>
<td>12704</td>
<td>28600</td>
<td>79348</td>
<td>197180</td>
<td>496636</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vertices:</td>
<td>6354</td>
<td>14298</td>
<td>39676</td>
<td>98592</td>
<td>248320</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Time [ms]:</td>
<td>469</td>
<td>969</td>
<td>2531</td>
<td>6016</td>
<td>14000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Triangles:</td>
<td>11283</td>
<td>24869</td>
<td>68807</td>
<td>170753</td>
<td>430251</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vertices:</td>
<td>5643</td>
<td>12434</td>
<td>34405</td>
<td>85376</td>
<td>215127</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Time [ms]:</td>
<td>390</td>
<td>859</td>
<td>2718</td>
<td>8000</td>
<td>25969</td>
<td></td>
</tr>
<tr>
<td>Jack</td>
<td>Marching Cubes</td>
<td>16984</td>
<td>38680</td>
<td>106888</td>
<td>264728</td>
<td>667144</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vertices:</td>
<td>8494</td>
<td>19342</td>
<td>53446</td>
<td>132368</td>
<td>333574</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Time [ms]:</td>
<td>703</td>
<td>2172</td>
<td>4047</td>
<td>7610</td>
<td>20828</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Triangles:</td>
<td>15647</td>
<td>34707</td>
<td>96051</td>
<td>237405</td>
<td>597965</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vertices:</td>
<td>7825</td>
<td>17355</td>
<td>48025</td>
<td>118704</td>
<td>298982</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Time [ms]:</td>
<td>469</td>
<td>1063</td>
<td>3344</td>
<td>10485</td>
<td>29578</td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th></th>
<th>min</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>-8</td>
<td>8</td>
</tr>
<tr>
<td>y</td>
<td>-8</td>
<td>8</td>
</tr>
<tr>
<td>z</td>
<td>-8</td>
<td>8</td>
</tr>
</tbody>
</table>

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.](image)

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 fronty 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 vznikajícího 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.
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.
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.

<table>
<thead>
<tr>
<th>N</th>
<th>160</th>
<th>240</th>
<th>400</th>
<th>630</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Naměřený čas [ms]</td>
<td></td>
</tr>
<tr>
<td>Sphere</td>
<td></td>
<td></td>
<td></td>
<td>Originál</td>
<td>63</td>
</tr>
<tr>
<td></td>
<td>1. Uprava</td>
<td>31</td>
<td>46</td>
<td>156</td>
<td>422</td>
</tr>
<tr>
<td></td>
<td>2. Uprava</td>
<td>16</td>
<td>32</td>
<td>109</td>
<td>265</td>
</tr>
<tr>
<td>Torus</td>
<td></td>
<td></td>
<td></td>
<td>Originál</td>
<td>78</td>
</tr>
<tr>
<td></td>
<td>1. Uprava</td>
<td>31</td>
<td>62</td>
<td>141</td>
<td>297</td>
</tr>
<tr>
<td></td>
<td>2. Uprava</td>
<td>16</td>
<td>31</td>
<td>63</td>
<td>188</td>
</tr>
<tr>
<td>Blob</td>
<td></td>
<td></td>
<td></td>
<td>Originál</td>
<td>641</td>
</tr>
<tr>
<td></td>
<td>1. Uprava</td>
<td>125</td>
<td>297</td>
<td>922</td>
<td>2844</td>
</tr>
<tr>
<td></td>
<td>2. Uprava</td>
<td>94</td>
<td>218</td>
<td>609</td>
<td>1610</td>
</tr>
</tbody>
</table>

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 snižení detailů (zvětšení mřížky, tzn. parametr *cubsize*), 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 zvášť uváděny grafy s poměry rychlosti výpočtu $q = \frac{1.\text{metoda}}{2.\text{metoda}}$. 

41
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 podrobená 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

\[ s = \frac{t_1}{t_n} \]  

(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:

\[
e = \frac{s}{p} = \frac{t_{1}}{p \cdot t_{n}}
\]  

(5.2).

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

<table>
<thead>
<tr>
<th>Funkce</th>
<th>Počet procesorů</th>
<th>Naměřený čas [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>N</td>
<td>20</td>
</tr>
<tr>
<td>Spheres</td>
<td>1</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>78</td>
</tr>
<tr>
<td>Blob</td>
<td>1</td>
<td>94</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>78</td>
</tr>
<tr>
<td>Jack</td>
<td>1</td>
<td>94</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>94</td>
</tr>
<tr>
<td>Tap</td>
<td>1</td>
<td>547</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>375</td>
</tr>
<tr>
<td>Spirit</td>
<td>1</td>
<td>609</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>500</td>
</tr>
</tbody>
</table>

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

Nastavení oblasti v průběhu měření je znázorněno v tabulce 5.6.

<table>
<thead>
<tr>
<th></th>
<th>min</th>
<th>max</th>
<th></th>
<th>min</th>
<th>max</th>
<th></th>
<th>min</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>-2</td>
<td>2</td>
<td>x</td>
<td>-4</td>
<td>4</td>
<td>x</td>
<td>-12</td>
<td>12</td>
</tr>
<tr>
<td>y</td>
<td>-2</td>
<td>2</td>
<td>y</td>
<td>-4</td>
<td>4</td>
<td>y</td>
<td>-12</td>
<td>12</td>
</tr>
<tr>
<td>z</td>
<td>-2</td>
<td>2</td>
<td>z</td>
<td>-4</td>
<td>4</td>
<td>z</td>
<td>-12</td>
<td>12</td>
</tr>
</tbody>
</table>

a) b) c)

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.

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:

\[
s \leq \frac{1}{f + \frac{1-f}{p}} \leq \frac{1}{f}
\]

(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:

\[
q = \frac{p \cdot (s_{\text{teor}} - 1)}{s_{\text{teor}} \cdot (p-1)}
\]

(5.4),

kde \(s_{\text{teor}}\) 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.](image-url)
Výpočet potenciálně paralelní části kódu je, podobně jako efektivita a urychlení, ovlivněn reží 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:

\[
p_x^2 + p_y^2 + p_z^2 - r^2 = 0. \quad (5.5)
\]

Normalizovaný tvar:

\[
\sqrt{p_x^2 + p_y^2 + p_z^2} - r = 0. \quad (5.6)
\]

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 \( f(X) < \varepsilon \), kde \( X \) je hledaný bod na povrchu. Konstanta \( \varepsilon \) 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.

<table>
<thead>
<tr>
<th></th>
<th>min</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>-4</td>
<td>4</td>
</tr>
<tr>
<td>y</td>
<td>-4</td>
<td>4</td>
</tr>
<tr>
<td>z</td>
<td>-4</td>
<td>4</td>
</tr>
</tbody>
</table>

Tabulka 5.7: Parametr *Area size* během výpočtů.
<table>
<thead>
<tr>
<th></th>
<th>N</th>
<th>160</th>
<th>240</th>
<th>400</th>
<th>630</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Normalizovaný</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>triangles</td>
<td>13205</td>
<td>29681</td>
<td>81682</td>
<td>203133</td>
<td>509459</td>
<td></td>
</tr>
<tr>
<td>vertices</td>
<td>6604</td>
<td>14842</td>
<td>40842</td>
<td>101568</td>
<td>254731</td>
<td></td>
</tr>
<tr>
<td>time [ms]</td>
<td>266</td>
<td>1500</td>
<td>2796</td>
<td>9188</td>
<td>33953</td>
<td></td>
</tr>
<tr>
<td>total deviation</td>
<td>0,04730</td>
<td>0,10616</td>
<td>0,29780</td>
<td>0,70550</td>
<td>1,51291</td>
<td></td>
</tr>
<tr>
<td>average deviation</td>
<td>7,16291E-06</td>
<td>7,15293E-06</td>
<td>7,29139E-06</td>
<td>6,94610E-06</td>
<td>5,93926E-06</td>
<td></td>
</tr>
<tr>
<td>relative deviation</td>
<td>5,70238E-07</td>
<td>5,69325E-07</td>
<td>5,80256E-07</td>
<td>5,52768E-07</td>
<td>4,72638E-07</td>
<td></td>
</tr>
<tr>
<td>surface area</td>
<td>12,56126</td>
<td>12,56388</td>
<td>12,56563</td>
<td>12,56602</td>
<td>12,56618</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Nenormalizovaný</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>triangles</td>
<td>13141</td>
<td>29585</td>
<td>81217</td>
<td>202130</td>
<td>509851</td>
<td></td>
</tr>
<tr>
<td>vertices</td>
<td>6572</td>
<td>14794</td>
<td>40610</td>
<td>101066</td>
<td>254927</td>
<td></td>
</tr>
<tr>
<td>time [ms]</td>
<td>250</td>
<td>719</td>
<td>2828</td>
<td>10094</td>
<td>254927</td>
<td></td>
</tr>
<tr>
<td>total deviation</td>
<td>0,04174</td>
<td>0,10736</td>
<td>0,29403</td>
<td>0,69446</td>
<td>1,66140</td>
<td></td>
</tr>
<tr>
<td>average deviation</td>
<td>6,35088E-06</td>
<td>7,25688E-06</td>
<td>7,24040E-06</td>
<td>6,87139E-06</td>
<td>6,51715E-06</td>
<td></td>
</tr>
<tr>
<td>relative deviation</td>
<td>5,05591E-07</td>
<td>5,77598E-07</td>
<td>5,76214E-07</td>
<td>5,46821E-07</td>
<td>5,18624E-07</td>
<td></td>
</tr>
<tr>
<td>surface area</td>
<td>12,56129</td>
<td>12,56385</td>
<td>12,56548</td>
<td>12,56598</td>
<td>12,56623</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>N</th>
<th>160</th>
<th>240</th>
<th>400</th>
<th>630</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Normalizovaný</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>triangles</td>
<td>14984</td>
<td>33752</td>
<td>93992</td>
<td>233720</td>
<td>588728</td>
<td></td>
</tr>
<tr>
<td>vertices</td>
<td>7494</td>
<td>16878</td>
<td>46998</td>
<td>116862</td>
<td>294366</td>
<td></td>
</tr>
<tr>
<td>time [ms]</td>
<td>766</td>
<td>1796</td>
<td>3735</td>
<td>5109</td>
<td>13813</td>
<td></td>
</tr>
<tr>
<td>total deviation</td>
<td>0,03360</td>
<td>0,07506</td>
<td>0,20303</td>
<td>0,50700</td>
<td>1,32124</td>
<td></td>
</tr>
<tr>
<td>average deviation</td>
<td>4,48368E-06</td>
<td>4,44696E-06</td>
<td>4,32002E-06</td>
<td>4,33842E-06</td>
<td>4,48843E-06</td>
<td></td>
</tr>
<tr>
<td>relative deviation</td>
<td>3,57013E-07</td>
<td>3,53972E-07</td>
<td>3,43809E-07</td>
<td>3,45253E-07</td>
<td>3,57182E-07</td>
<td></td>
</tr>
<tr>
<td>surface area</td>
<td>12,55889</td>
<td>12,56301</td>
<td>12,56518</td>
<td>12,56590</td>
<td>12,56623</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Nenormalizovaný</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>triangles</td>
<td>14984</td>
<td>33752</td>
<td>93992</td>
<td>233720</td>
<td>588728</td>
<td></td>
</tr>
<tr>
<td>vertices</td>
<td>7494</td>
<td>16878</td>
<td>46998</td>
<td>116862</td>
<td>294366</td>
<td></td>
</tr>
<tr>
<td>time [ms]</td>
<td>250</td>
<td>796</td>
<td>1734</td>
<td>3719</td>
<td>9672</td>
<td></td>
</tr>
<tr>
<td>total deviation</td>
<td>0,03046</td>
<td>0,07426</td>
<td>0,21062</td>
<td>0,50627</td>
<td>1,28977</td>
<td></td>
</tr>
<tr>
<td>average deviation</td>
<td>4,06433E-06</td>
<td>4,39965E-06</td>
<td>4,48145E-06</td>
<td>4,33223E-06</td>
<td>4,38153E-06</td>
<td></td>
</tr>
<tr>
<td>relative deviation</td>
<td>3,23622E-07</td>
<td>3,50207E-07</td>
<td>3,47615E-07</td>
<td>3,48678E-07</td>
<td>3,48678E-07</td>
<td></td>
</tr>
<tr>
<td>surface area</td>
<td>12,55891</td>
<td>12,56301</td>
<td>12,56517</td>
<td>12,56590</td>
<td>12,56619</td>
<td></td>
</tr>
</tbody>
</table>

Tabulka 5.9: Naměřené hodnoty odchylek aproximací normalizovaného a nenormalizovaného tvaru funkce *Sphere* generované metodou *Marching cubes*.
V grafích 5.15 a 5.16 je znázorněna závislost kvality aproximace polygonálního objektu normalizované a nenormalizované funkce \( \textit{Sphere} \) \((R = 1)\).

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 \( \textit{Sphere} \) polygonizované metodou \textit{Marching triangles}. 

---

1,00000E-06 1,00000E-05 1,00000E-04 1,00000E-03 1,00000E-02 1,00000E-01 1,00000E+00 

160 240 400 630 1000 

Průměrná odchylka 

\[ \text{Normalizovaná} \]

\[ \text{Nenormalizovaná} \]
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 ve 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 \( f(X) < \varepsilon \) 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.
<table>
<thead>
<tr>
<th>R = 0,5</th>
<th>N</th>
<th>160</th>
<th>240</th>
<th>400</th>
<th>630</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triangles:</td>
<td>3323</td>
<td>7415</td>
<td>20423</td>
<td>50611</td>
<td>127257</td>
<td></td>
</tr>
<tr>
<td>Vertices:</td>
<td>1663</td>
<td>3709</td>
<td>10213</td>
<td>25307</td>
<td>63630</td>
<td></td>
</tr>
<tr>
<td>Time [ms]:</td>
<td>30</td>
<td>70</td>
<td>220</td>
<td>641</td>
<td>2464</td>
<td></td>
</tr>
<tr>
<td>Total deviation:</td>
<td>0,01181</td>
<td>0,02610</td>
<td>0,07477</td>
<td>0,17803</td>
<td>0,42580</td>
<td></td>
</tr>
<tr>
<td>Average deviation:</td>
<td>7,09962E-06</td>
<td>7,03777E-06</td>
<td>7,32089E-06</td>
<td>7,03468E-06</td>
<td>6,69178E-06</td>
<td></td>
</tr>
<tr>
<td>Relative deviation:</td>
<td>2,26393E-06</td>
<td>2,24202E-06</td>
<td>2,33098E-06</td>
<td>2,23947E-06</td>
<td>2,13014E-06</td>
<td></td>
</tr>
<tr>
<td>Surface area:</td>
<td>3,135971</td>
<td>3,139031</td>
<td>3,140695</td>
<td>3,141223</td>
<td>3,141474</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R = 1</th>
<th>N</th>
<th>160</th>
<th>240</th>
<th>400</th>
<th>630</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triangles:</td>
<td>13141</td>
<td>29585</td>
<td>81217</td>
<td>202130</td>
<td>509851</td>
<td></td>
</tr>
<tr>
<td>Vertices:</td>
<td>6572</td>
<td>14794</td>
<td>40610</td>
<td>101066</td>
<td>254927</td>
<td></td>
</tr>
<tr>
<td>Time [ms]:</td>
<td>250</td>
<td>719</td>
<td>2828</td>
<td>10094</td>
<td>33422</td>
<td></td>
</tr>
<tr>
<td>Total deviation:</td>
<td>0,04174</td>
<td>0,10736</td>
<td>0,29403</td>
<td>0,69446</td>
<td>1,66140</td>
<td></td>
</tr>
<tr>
<td>Average deviation:</td>
<td>6,35088E-06</td>
<td>7,25686E-06</td>
<td>7,24040E-06</td>
<td>6,87139E-06</td>
<td>6,51715E-06</td>
<td></td>
</tr>
<tr>
<td>Relative deviation:</td>
<td>5,05591E-07</td>
<td>5,77598E-07</td>
<td>5,76214E-07</td>
<td>5,46821E-07</td>
<td>5,18624E-07</td>
<td></td>
</tr>
<tr>
<td>Surface area:</td>
<td>12,56129</td>
<td>12,56385</td>
<td>12,56548</td>
<td>12,56607</td>
<td>12,56623</td>
<td></td>
</tr>
</tbody>
</table>

Tabulka 5.10: Naměřené hodnoty odchylek aproximace 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

$$q = \frac{\text{Naměřený obsah}}{\text{Ideální obsah}}.$$ 

Ideální obsah, tj. obsah koule, je počítán podle vzorce $P = 4 \cdot \pi \cdot r^2$. Pro námi zvolený poloměr je tedy ideální obsah $P_{id} = 4 \cdot \pi \cdot 0.5^2 = \pi = 3,1415926535897932384626433832795$. 

55
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 její 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.

Literatura


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


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

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


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.
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).
  - 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 \((\pi/n)\).
  - 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.

![Implicit surface editor dialog](image)

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

```cpp
// 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 symmetric 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 {
    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 {
    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 {
    POINT1 position;  // coordinates
    POINT1 normal;   // surface normal
} VERTEX;

Array of vertices.

typedef struct vertices {
    int count, max;  // # vertices, max # allowed
    VERTICES *ptr;  // dynamically allocated
} VERTICES;

Triangle data structure.

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

Array of triangles.

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

Coordinates definition in 3D.

typedef struct point1 {
    double x, y, z;  // its coordinates
} POINT1;
Data structure of polygonizing cell.

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

typedef struct corner {
    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 {
    CUBE cube; // a single cube
    struct cubes *next; // remaining elements
} CUBES;

HASH tables.

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

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

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

typedef struct intlists {
    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
  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]).

\[
\begin{align*}
#&\text{define HASHBIT } \quad \{6\} \\
#&\text{define HASHSIZE } \quad (\text{size}_t)(1<<\{3\*\text{HASHBIT}\}) \quad /* \text{hash table size } (262144) */ \\
#&\text{define MASK } \quad ((1<<\text{HASHBIT})-1) \quad /* 00111111 */ \\
#&\text{define HASH}(i,j,k) \quad ((( ((i)\&\text{MASK})<<\text{HASHBIT})|((j)\&\text{MASK}))<<\text{HASHBIT})|((k)\&\text{MASK}))
\end{align*}
\]

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.](image)

Note:

In the Marching triangles method, the indexes \(i, j, k\) are computed from coordinates of point \(p\), as:

\[
\begin{align*}
i &= (\text{int})(p.x / dt) - \text{middle}[0]; \\
j &= (\text{int})(p.y / dt) - \text{middle}[1]; \\
k &= (\text{int})(p.z / dt) - \text{middle}[2];
\end{align*}
\]

where \(dt = \text{cubsize} = \delta\) and middle is array which contents indexes of start point.