University of West Bohemia in Pilsen
Department of Computer Science and Engineering
Univerzitni 8
30614 Pilsen
Czech Republic

Kompilovaný HyperFun

Research of Report

Karel Uhlíř, Václav Skala

Technical Report No. DCSE/TR-2002-07
April, 2002

Distribution: public

Technical Report No. DCSE/TR-2002-09
March 2002

Kompilovaný HyperFun

Karel Uhlíř, Václav Skala

Abstract

This paper describes the developing and testing of a new method for processing and modeling implicitly defined objects. The base idea started from the popular system HyperFun. The HyperFun is a system, which develops at the University of Hosei. It is the system for modeling complex models through the HyperFun language and a lot of library function and operations. Our new method is based on the similarity of the HyperFun and C languages. The new method uses the C language to describe of structured models, which are then compiled and evaluated. This provides an easier way in comparison to model parsing as it is in HyperFun. This procedure is the cause of the indispensable speedup this method brings.

This work was supported by the Ministry if Education of Czech Republic – project MSM23500005

Copies of this report are available on
http://www.kiv.zcu.cz/publications/
or by surface mail on request sent to the following address:

University of West Bohemia in Pilsen
Department of Computer Science and Engineering
Univerzitni 8
30614 Pilsen
Czech Republic

Osnova

1. Základní informace

Tato zpráva shrnuje výsledky dosažené při zkoumání možnosti urychlení zpracování implicitně definovaných těles. Základní motivací byl projekt HyperFun [Hyper] pracující s vysokoúrovňovým jazykem HyperFun. Projekt vznikl na University of Hosei a je zaměřen na jednoduché modelování s použitím již definovaných těles nebo těles popsaných právě jazykem HyperFun. Stručný popis jazyka, který je nutný pro pochopení další práce je uveden v kapitole 2. V kapitole 3. jsou již uvedeny základy práce na metodě umožňující vyhodnocení modelů popsaných jazykem C++ s návazností na modely popsané jazykem HyperFun.

2. Popis programu HyperFun

Je to prostředek pro vytváření modelů z implicitně definovaných těles (v budoucnu nejen implicitně definovaných), která jsou popsána jazykem HyperFun. Tento vysokoúrovňový jazyk umožňuje popisovat složité modely primitivy definovanými v interní knihovně programu nebo jazykem HyperFun, který je svou syntaxí velice podobný jazyku C. Základním stavebním kamenem jsou funkce, operace a operátory, které mohou být opět z knihovny nebo popsány jazykem. Výčet knihovních funkcí, operací a operátorů je uveden v příloze A.

HyperFun provádí překlad modelu (obrázek 1) interpretací do vnitřní datové struktury, kterou následně vyhodnocuje. Vyhodnocení je prováděno ve vrcholech mřížky, jež rovnoměrně rozděluje prostor na voxely. Z toho plyne, že před vyhodnocením modelu musí být znám interval, na kterém bude daný model vyhodnocován a dělení daného intervalu. Dělení intervalu určuje přesné souřadnice jednotlivých vrcholů, kde bude model vyhodnocen. Tyto parametry a mnoho dalších jsou zadávány při spuštění programu (příloha B). Program vyhodnocuje všechny voxely mřížky. To znamená, že neprovádí žádnou další heuristiku, kterou by zredukoval počet voxelů, které je nutné vypočítat k nalezení celého modelu. Program se interně skládá z několika části:

1. Parser pro zpracování modelu do interních datových struktur.
2. Výpočetní část vyhodnocující model v jednotlivých vrcholech mřížky.
3. Vytvoření trojúhelníkové sítě [Pasko88].
4. Zobrazeni trojúhelníkové sítě.

Základní struktura modelu popsaneho HyperFun jazykem se zkládá z několika částí:

Hlavička:

1. *my_model* – název modelu (může se předefinovat).
2. *x[i]* – vstupní parametry, nejčastěji prostorové souřadnice x,y a z, ve kterých bude model vyhodnocen. Parametrů může být až 7. Jejich popis lze nalézt v [Hyper].
3. *a[i]* – pole vstupních prametrů předávaných z jiného modulu.

Tělo:

1. *array* – definice polí použitých v modelu.
2. *hfSphere()* – použití knihovních primitiv.
3. *Parallelepiped()* – volání primitiva vytvořeného uživatelem.

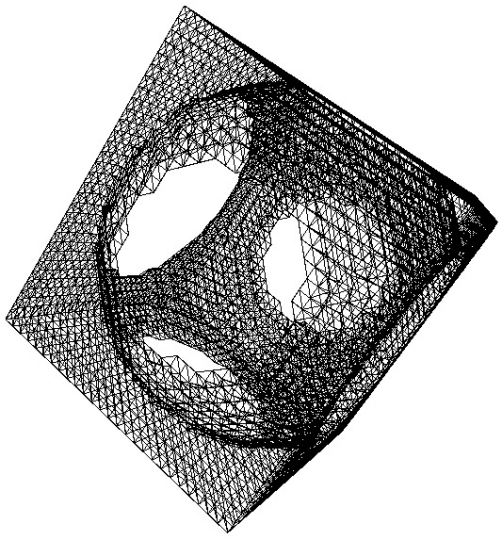
Konec:

1. *my_model* – navrácení hodnoty modelu.
2. *'\', '|'* – operátory.

```
- definice objektu "Parallelepiped"
Parallelepiped(x[3], a[6])
{
 Parallelepiped = (x[1] - a[1]) & (-(x[1] - a[1]) + a[4]) &
 (x[2] - a[2]) & (-(x[2] - a[2]) + a[5]) &
 (x[3] - a[3]) & (-(x[3] - a[3]) + a[6]);
}

- definice objektu "Sphere"
Sphere(x[3], a[1])
{
 Sphere = a[1]^2 - x[1]^2 - x[2]^2 - x[3]^2;
}

- definice s použitím dříve vytvořených objektů
my_model(x[3], a[1])
{
 array param[6], center[3], rad[1];
 center = [5,5,0];
 param = [-9.0, -8.0, -9.0, 18.0, 16.0, 18.0];
 rad = [10.];
 block = Parallelepiped(x, param);
 sp1 = hfSphere(x,center,4.0);
 my_model = block \ Sphere(x,rad) | sp1;
}
- výsledný model je rozdílem krychle a dvou sjednocených koulí
```



obrázek 1. Ukázka modelu v jazyce HyperFun.

Struktura modelu (obrázek 1) ukazuje základní objekty a definice, které se používají při vytváření modelu.

Pro úplnost ještě operátory, které jsou základním stavebním prvkem a dále budou použity jako základ pro naši metodu (tabulka 1).

Název	Operátor	Matematický popis
Sjednocení	|	$f_1 \mid f_2 = \frac{1}{1+\alpha}\left(f_1 + f_2 + \sqrt{\left\lvert f_1^2 + f_2^2 - 2\cdot\alpha\cdot f_1 f_2 \right\rvert}\right)$
Průnik	&	$f_1 \,\&\, f_2 = \frac{1}{1+\alpha}\left(f_1 + f_2 - \sqrt{\left\lvert f_1^2 + f_2^2 - 2\cdot\alpha\cdot f_1 f_2 \right\rvert}\right)$
Rozdíl	\	$f_1 \setminus f_2 = f_1 \,\&\, -f_2$

tabulka 1. Základní operátory a jejich matematický popis.

Uvedený matematický popis operátorů [Pasko95] zajišťuje C^0 spojitost ploch určených funkcemi f_1 a f_2. Spojitost je závislá na parametru α. Změnou tohoto parametru je možné měnit vyjádření operátorů. Například jestliže $\alpha = 1$, dostáváme praktické vyjádření operátorů (jeden ze singulárních případů), které je C^1 nespojité když $f_1(x) = f_2(x)$. Forma zápisu operátorů, kdy $\alpha = 1$ je použita dále.

Toto byl pouze stručný přehled programu HyperFun. Podrobnější informace lze nalézt na www stránkách projektu [Hyper] nebo v [Adzhi99].

3. Kompilace modelu

Náš systém vychází z předpokladu, že HyperFun provádí překlad vstupního souboru s modelem do vnitřní datové struktury, kterou posléze vyhodnocuje. Abychom se vyhnuli této fázi, navrhli jsme způsob, při kterém bude model zpracováván standardním překladačem jazyka C (později C++). Takto jsme se vyhnuli vytváření parseru pro vstupní soubor což bylo výhodou, ale naproti tomu nám vznikla nutnost vlastnit překladač C (je možné použít volně šiřitelný kompilátor). Tato nevýhoda byla vykoupena značným urychlením.

Model v jazyce HyperFun	Model v jazyce C++

my_model(x[3], a[1])
{
array center[3];
 	
center = [5,5,0];
sp1=hfSphere(x,center,4.);
center[2]=-5;
sp2=hfSphere(x,center,4.);
 	
my_model = sp1 | sp2;
}
	FD FD::sphere(double x[])
	{
	array center[3];
	FD sp1,sp2;
	
	center[3] = 1.5,0,0;
	sp1 = fSphere(x,center,2.);
	center[3] = -1.5,0,0;
	sp2 = fSphere(x,center,2.);
	
	return (sp1 | sp2);
	}

tabulka 2. Shodný model v jazyce HyperFun a C++.

Základní myšlenkou bylo zapsání modelu tak, aby ho bylo možné přeložit standardním kompilátorem. Dalším krokem, bylo navržení datových struktur (příloha C) tak, aby byla co největší podobnost modelu zapsaného v jazyce HyperFun a modelu zapsaného v C++ (tabulka 2). Pro tento případ se nám velice hodilo přetěžování operátorů, které nám podobnost zajistilo (obrázek 2). Základy byly uvedeny již v [Uhlir01], kde byly uvedeny i první grafy ukazující porovnání paměťové a časové náročnosti pří vyhodnocování modelů (graf 1 a graf 2).

Sjednocení (union)

FmodelDouble& FmodelDouble::operator |(FmodelDouble &pA)
{
if(this->m_dRes < pA.m_dRes) return(pA);
else return(*this);
}

Průnik (intersection)

```
FmodelDouble& FmodelDouble::operator &(FmodelDouble &pA)
{
 if(this->m_dRes > pA.m_dRes) return(pA);
 else return(*this);
}
```

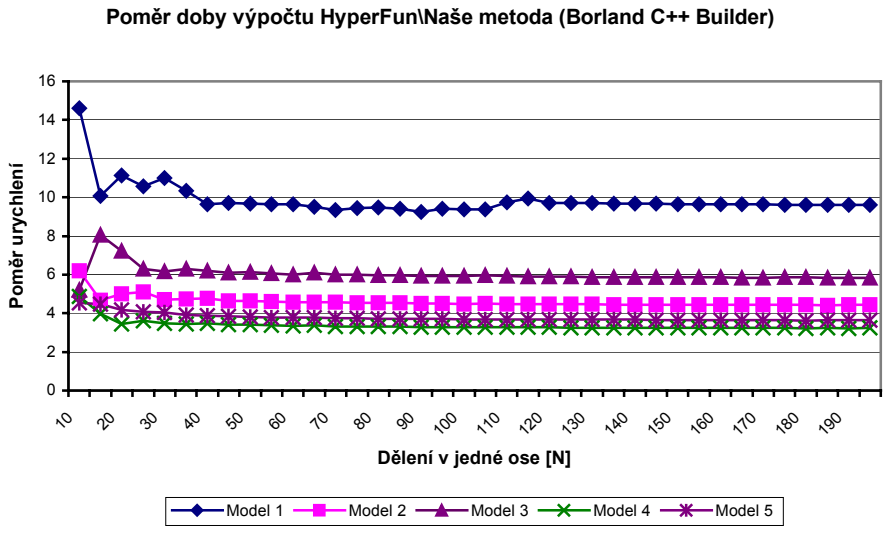
Rozdíl (subtraction)

```
FmodelDouble& FmodelDouble::operator %(FmodelDouble &pA)
{
 if(this->m_dRes > -pA.m_dRes)
 {
  pA.m_dRes = -pA.m_dRes;
  return(pA);
 }
 else return(*this);
}
```

tabulka 3. Ukázka přetížení operátorů pro sjednocení, průnik a rozdíl .

Název	Operátor HyperFun	Operátor v kompilované verzi
Sjednocení	|	|
Průnik	&	&
Rozdíl	\	%

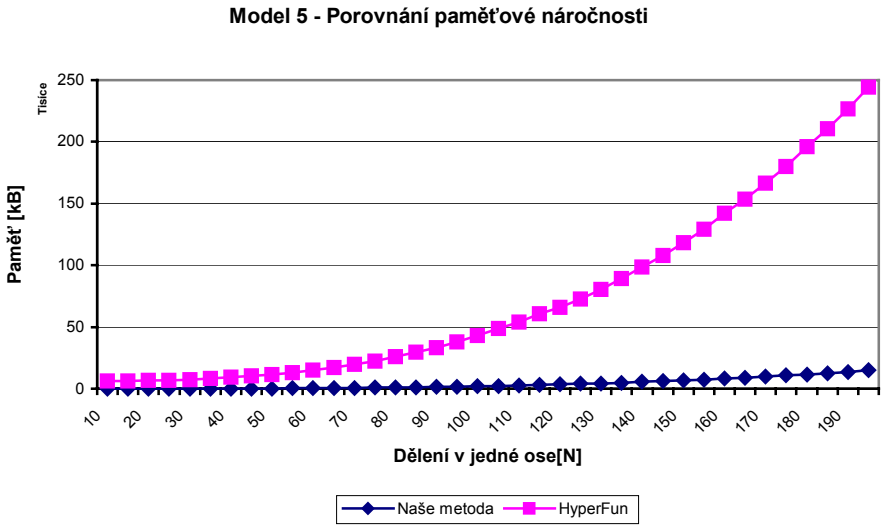
tabulka 4. Znaky pro operátory.



graf 1. Poměr urychlení na pěti různých modelech.

Přetěžování operátorů (tabulka 3), které zajistilo podobnost zápisu modelu sice nepatrně snižuje urychlení, ale umožňuje uživatelům, kteří znají jazyk HyperFun, zapisovat model v prakticky nezměněné podobě. Skoro všechny základní operátory se podařilo přetížit (tabulka 4). Jediný operátor, který se nepodařilo přetížit byl operátor rozdílu

používající znak ‘\’. Jelikož tento znak není operátor jazyka C, tak nemohl být přetížen a pro operaci rozdílu byl použit znak ‘%’.



graf 2. Porovnání paměťové náročnosti.

Z uvedených grafů je vidět dosažené urychlení a zároveň paměťová složitost, které vedli k dalšímu podrobnějšímu studiu použité metody. Je však nutné zde podotknout, že porovnání paměťové složitosti je trochu zkreslené. Podrobnosti budou osvětleny dále, kde bude také provedena podrobnější analýza.

4. Další vývoj metody a testování

Další směr vývoje se ubíral hlavně cestou testování. Základní věc, kterou bylo nutné vysvětlit byly rozdílné hodnoty urychlení na různých modelech. Tímto směrem se ubírala hlavní část testů, které byly prováděny na pěti různých modelech (příloha D).

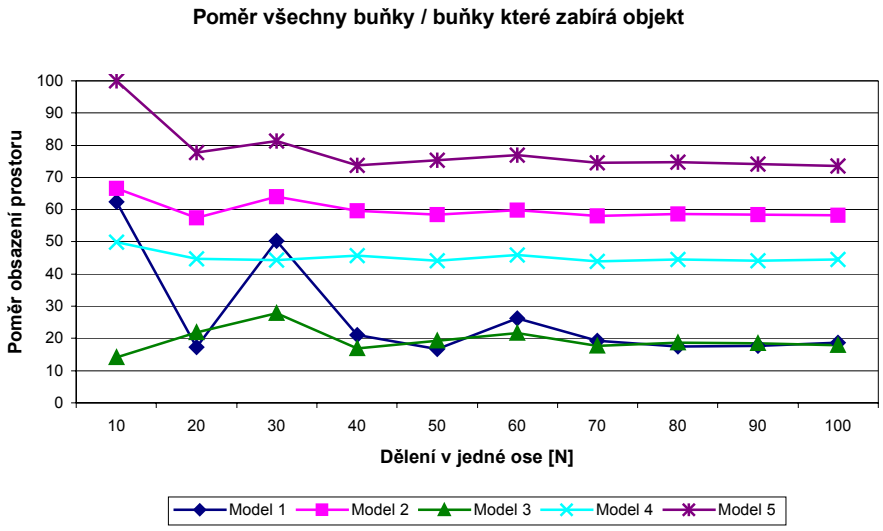
4.1 Test obsazení prostoru

V prvním ze série testů jsem se zabýval jak velkou část rovnoměrně rozděleného prostoru zabírají u jednotlivých modelů voxely, které jsou obsazeny modelem a je nutné je spočítat, vůči prázdným buňkám, které není nutné počítat (graf 3). Poměr obsazení prostoru nemůže mít vliv na rozdílná urychlení, ale tento test ukázal rezervy pro urychlování pokud bychom nějakým rozšířením metody [Cerm02] vyloučili voxely, které není nutné počítat. U některých modelů je poměr obsazených voxelů vůči celému prostoru (poměr v grafu) i několik desítek procent.

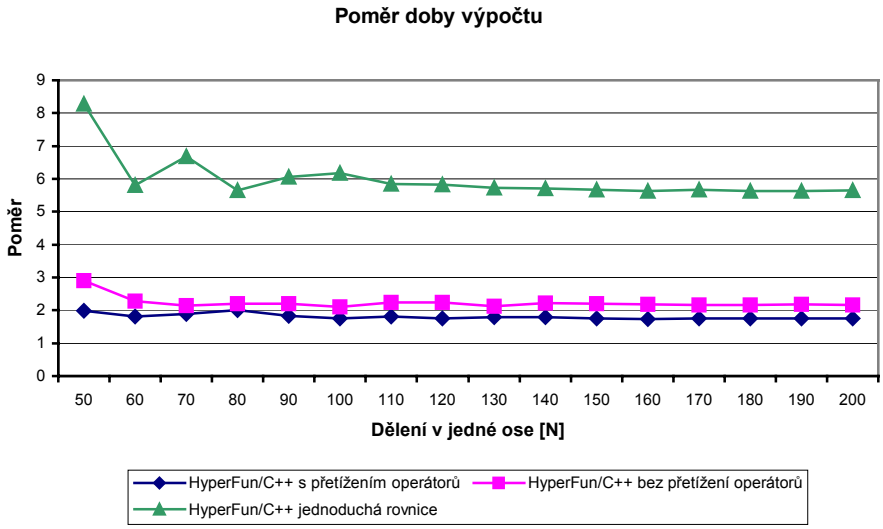
4.2 Stanovení základního poměru urychlení

Neboť urychlení pro různé modely bylo rozdílné, tak jsme se snažili stanovit odrazovou hladinu pro urychlení. Předpokladem bylo, že základní urychlení bude stanoveno

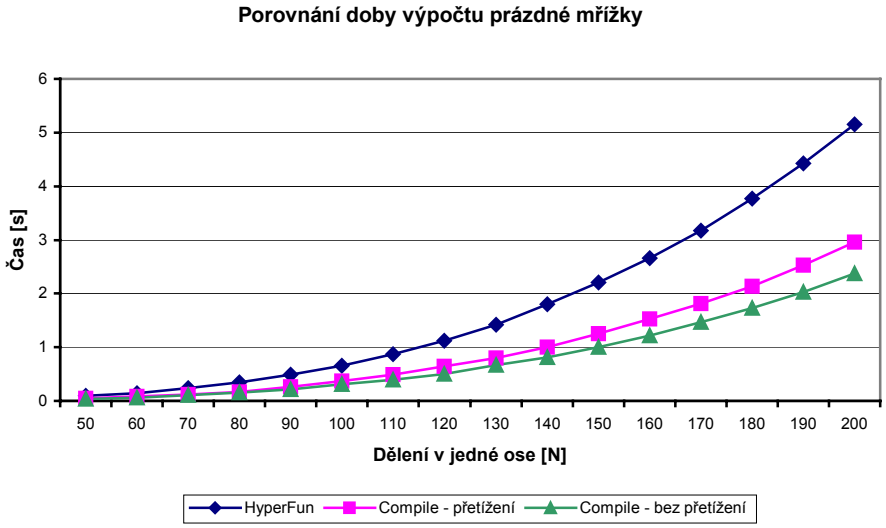
poměrem doby výpočtu prázdného modelu. V podstatě průchod celé mřížky a vypočtení prázdného modelu bez toho, aby do výpočtu zasáhly vnitřní proměnné modelu (graf 4). Zároveň s tímto testem byla otestována i režie při přetěžování operátorů, které bylo použito z hlediska přiblížení zápisu modelu zapsaného v C++ k zápisu v jazyce HyperFun (graf 5).



graf 3. Poměr obsazení prostoru u jednotlivých modelů.

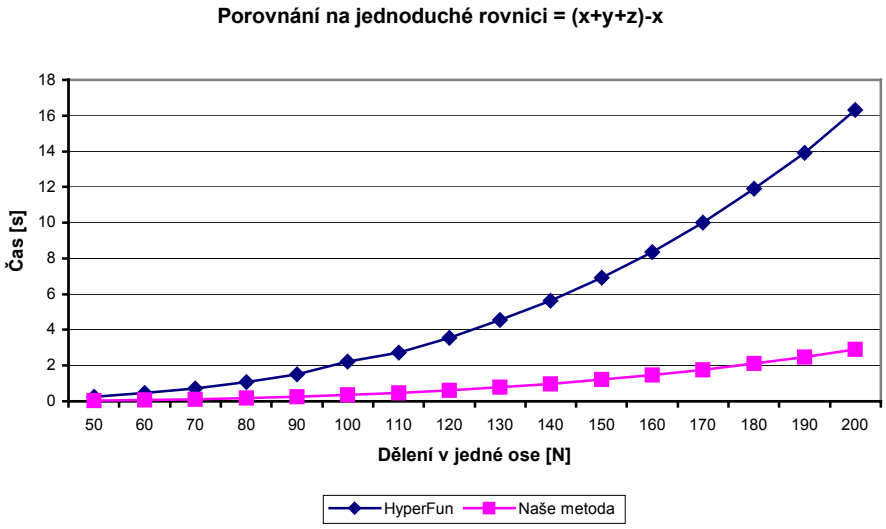


graf 4. Stanovení základního urychlení.



graf 5. Porovnání doby výpočtu na prázdné mřížce.

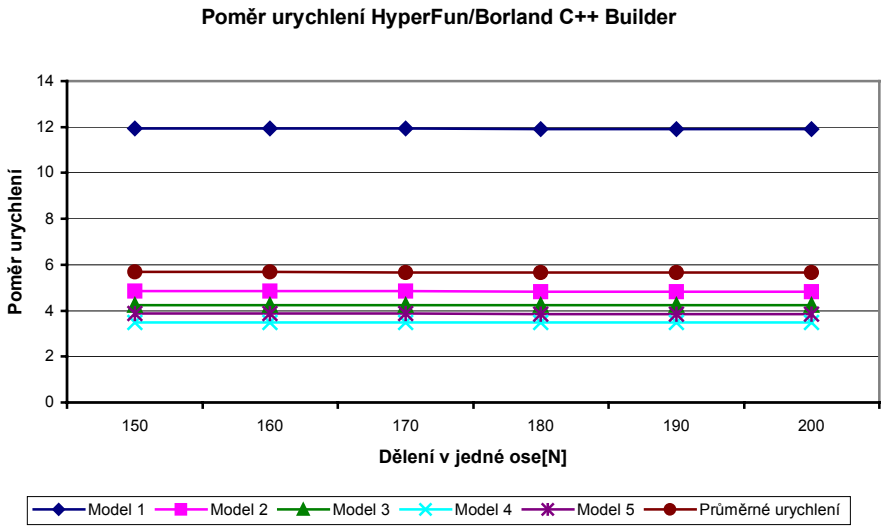
Jednalo se o zápis modelu pouze s jedním přetíženým operátorem (=). Ukázalo se, že přetěžování má vliv na dobu výpočtu. Rychlejší přístup je určitě bez přetěžování, ale otázkou je zda zvolit pohodlnost a podobnost zápisu nebo rychlost. Pro názornost byl proveden ještě výpočet jednoduché funkce. Její porovnání s výpočty na prázdné mřížce lze nalézt na graf 6.



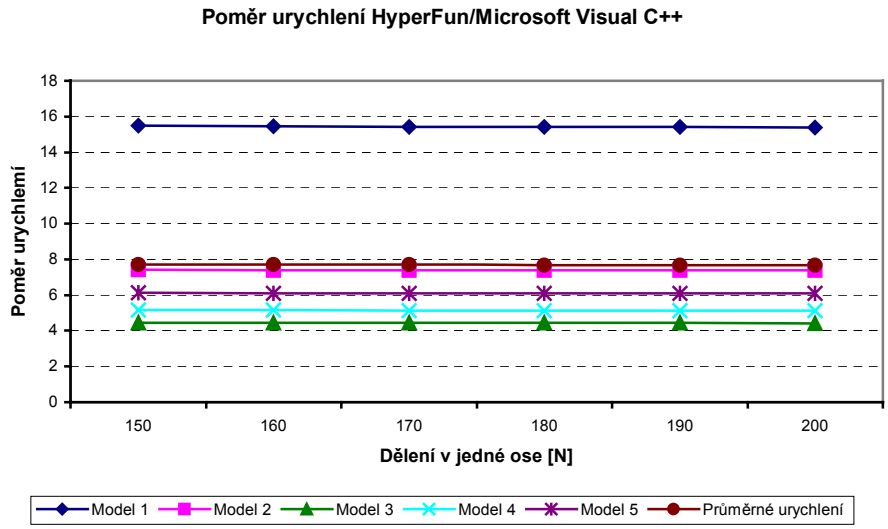
graf 6. Porovnání doby výpočtu na jednoduché rovnici.

4.3 Různé kompilátory

Vzhledem k tomu, že na rychlost výpočtu modelu má určitě vliv i optimalizace kompilátoru, tak byl zdrojový kód vytvářený v Borland C++ Builder přenesen do Microsoft Visual C++, aby bylo možné testovat tento aspekt (graf 7). Včetně tohoto testu byla vyzkoušena různá nastavení kompilátoru. Ukázalo se, že přepínání nastavení kompilátoru nemá žádný vliv na urychlení výpočtu. Naopak změnou kompilátoru se ukázalo několik zajímavých skutečností. První z nich byla rozdílná doba výpočtu pro první model vůči všem ostatním. Při výpočtu na prvním modelu byl rychlejší kompilátor Borland C++ Builder, ale ve všech ostatních případech tomu bylo naopak a rychlejší byl kompilátor Microsoft Visual C++. Bylo nutné stanovit důvod tohoto výsledku.



graf 7. Poměr urychlení HyperFun/Borland C++ Builder.

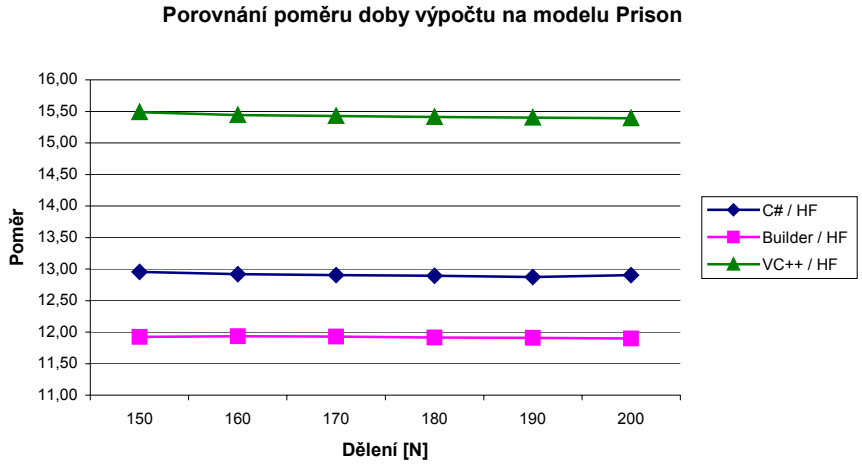


graf 8. Poměr urychlení HyperFun/Microsoft Visual C++.

Po zkoumání struktury modelu bylo zjištěno, že problém je v použití funkce *pow(mocnitel, mocnina)*, která je standardní funkcí Microsoft Visual C++ a provádí umocnění čísla na definovaný mocnitel. V modelu byla tato funkce použita pro druhou mocninu čísla. Jelikož, Borland C++ Builder si provádí optimalizaci tohoto výrazu na prosté vynásobení dvou čísel, tak se tento problém neprojevil při testech prováděných pouze v Borland C++ Builderu. Microsoft Visual C++ provádí jiný způsob optimalizace a to byl důvod zpomalení. Vyplývá z toho, že je nutné dávat pozor jaké jsou použité standardní funkce kompilátoru a v případě jejich použití se ptát na jejich konstrukci. Pro tento případ by bylo určitě lepší přetížit nějaký operátor a vytvořit si mocninu.

Dodatečně byl program přepsán i do jazyka C#. Tento jazyk postavený z části na jazyce Java a z části na jazyce C++, je plně objektový. Hned na začátku bylo nutné upustit od stejné podobnosti modelu popsaného v C++. Důvodů bylo několik jako například nemožnost přetížení některých operátorů nebo nutnost dynamické alokace proměnných v modelu. Je nutné také podotknout, že pro použití programu na jakémkoliv počítači je nutné nainstalovat .NET Framework, který zabírá zhruba 120MB. Použitelnost se tímto omezuje, ale podstatnější bylo otestování rychlostí zpracování modelu. Urychlení výpočtu modelu bylo proti kompilované verzi v C++ pomalejší. To bylo určitě z části způsobeno tím, že program byl plně objektový. Program se tedy svým urychlením posunul mezi Borland C++ Builder a Microsoft Visual C++, který vyšel z testů nejlépe (graf 9).

Dále bylo z těchto uvedených tesů stanoveno průměrné urychlení pro oba použité kompilátory. Tyto hodnoty lze nalézt v následující tabulce. Hodnoty jsou již bez vlivu funkce *pow()*. Mocnina byla pouze řádu dva a byla tedy nahrazena prostým násobením dvou shodných čísel.



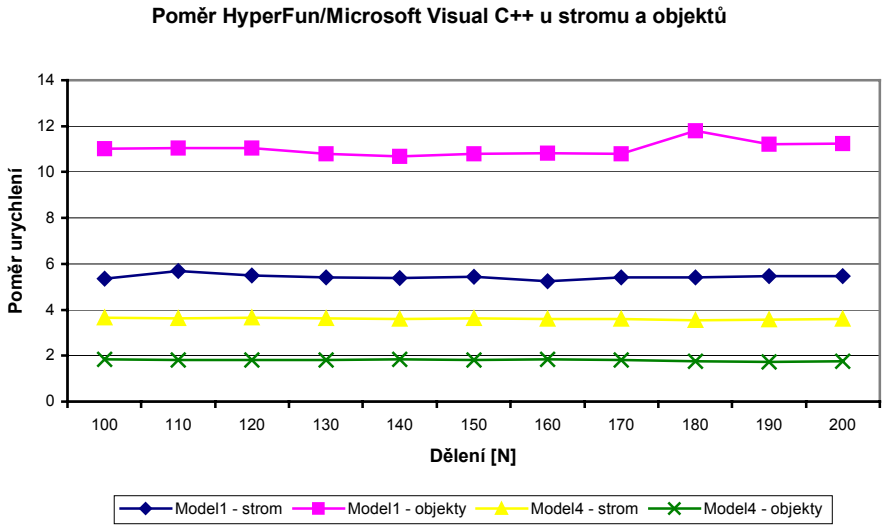
graf 9. Porovnání urychlení na jednom modelu.

Kompilátor	Minimální urychlení	Maximální urychlení	Průměrné urychlení
Borland C++ Builder	3,48 (Model 4)	11,92 (Model 1)	5,67
Microsoft Visual C++	4,44 (Model 3)	15,43 (Model 1)	7,70
.NET C#	3,86 (Model 4)	12,91 (Model 1)	6,32

tabulka 5. Poměry urychlení.

4.4 Objekty a strom

Tento test sloužil ke zjištění, kterým směrem se vydat při dalším urychlování. Měl stanovit, zda je náročnější vyhodnotit strom, nebo objekty. Výsledky ukazují na náročnější výpočet objektů, než vyhodnocení stromu. Poměr se pohybuje podle složitosti modelu, ale ani u velmi složitých objektů není vyhodnocení stromu časově složitější než vyhodnocení objektů (graf 10).



graf 10. Celková doba výpočtu a doba výpočtu objektů.

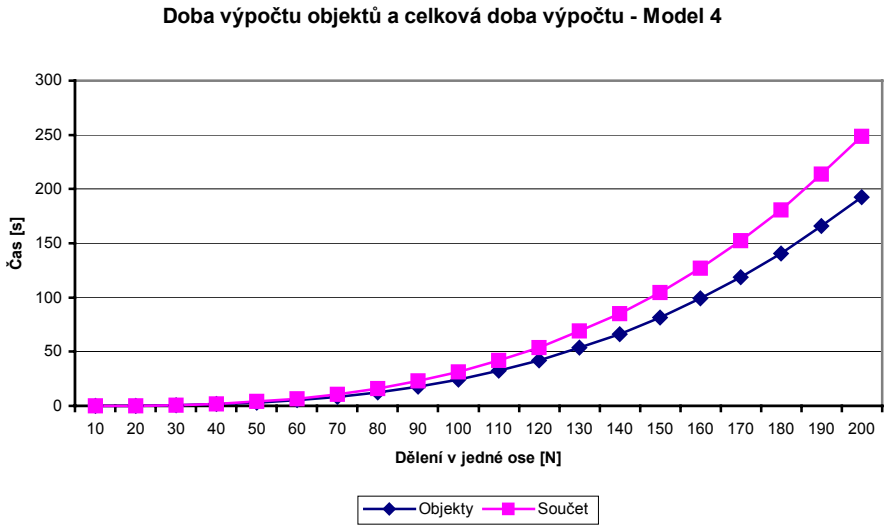
4.5 Volání objektů a funkcí

Počty volání operací a vyhodnocování objektů má vliv na urychlení. Aby byla lepší představa o tom, kolik je těchto operací prováděno a jaký je poměr urychlení na těchto operacích, tak byl vytvořen

graf 11. Kompletní tabulka počtu volání je uvedena v příloze E.

4.6 Paralelní implementace

V rámci možných způsobů bylo uvažováno i o paralelní implementaci. Byla provedena implementace pro MPI (Message passing interface) [MPI94] a thread verze pro symetrický multiprocesor.



graf 11. Poměr urychlení na vyhodnocování stromu a objektů.

4.6.1 MPI verze

Základní princip MPI spočívá v distribuci úlohy na počítače sdružené v heterogenní síti neboli multiprocesor s distribuovanou pamětí. Procesy mezi sebou komunikují prostřednictvím zpráv. Pro naší implementaci byl využit princip Farmer – Worker. Tento přístup spočívá ve vytvoření samostatných procesů, které v našem případě nepotřebují komunikovat. Jeden proces inicializuje potřebná data pro všechny pracující procesy a ty po skončení práce vrátí spočtená data.

Úloha byla distribuována jako řezy. Tato koncepce vychází z toho, že je nutné spočítat hodnotu modelu (funkce) ve všech buňkách 3D mřížky. Byl tedy určen pracovní interval pro každý pracující proces (počet 2D řezů). Každý proces dále pracuje nad svou kopií modelu a není nutné provádět interakci a řešit kritické sekce.

Výpočet byl prováděn na maximálně 14 počítačích v tomto složení:

4x Vectra desktop, Pentium Pro 200MHz
4x Vectra tower, Pentium Pro 200MHz
2x Integraph, Pentium II 450MHz
4x Dell, různé parametry

Všechny počítače jsou umístěny v laboratořích UL405 a UL407. Aby byly výsledky porovnatelné, budeme ukazovat pouze výsledky získané z počítačů Vectra desktop

a Vectra tower, které mají shodnou konfiguraci. Následující tabulky ukazují získané výsledky (tabulka 6). Čísla a písmena jsou jejich označení v síti.

Desktop	4	5	9	A	suma	průměr	par.koeficient.
Dělení 60	10,825				10,825	10,81575	1
	5,408	5,408			10,816		1,000832
	3,606	3,615	3,605		10,826		0,999908
	2,704	2,704	2,694	2,694	10,796		1,002686
Dělení 120	86,455				86,455	86,442	1
	43,202	43,193			86,395		1,000694
	28,801	28,801	28,872		86,474		0,99978
	21,601	21,601	21,651	21,591	86,444		1,000127

Tower	3	7	8	C	suma	průměr	par.koeficient.
Dělení 60	10,828				10,828	10,8205	1
	5,406	5,422			10,828		1
	3,594	3,609	3,61		10,813		1,001387
	2,703	2,704	2,703	2,703	10,813		1,001387
Dělení 120	86,438				86,438	86,422	1
	43,188	43,234			86,422		1,000185
	28,797	28,828	28,812		86,437		1,000012
	21,594	21,594	21,609	21,594	86,391		1,000544

tabulka 6. Doby výpočtu na jednotlivých procesorech.

Z uvedené tabulky je vidět, že urychlení je úměrné počtu procesorů. Musíme brát také na zřetel zatížení lokální sítě, které může mít vliv na nepatrné odchylky v časech při výpočtu na vícero počítačích. Tento test byl proveden na nezatížených počítačích. Dělení je 60 a 120 v jedné ose.

Samozřejmě, že výpočet lze na jednotlivé procesy rozdělit různým způsobem, ale v tomto případě šlo pouze o vyzkoušení základní myšlenky. Zajímavější by byl způsob použitý při použití nějakého výpočetního urychlení.

4.6.2 Thread verze

Tato verze byla také implementována, ale výsledky nemají velkou váhu. Na začátku výpočtu se vytvoří požadovaný počet vláken, které ukazují na jeden model. Model může obsahovat různé výpočty včetně lokálních proměnných a může tak docházet k jejich aktualizaci v různých vláknech. Kvůli této nechtěné interakci je nutné, aby celý model byl uzavřen v kritické sekci a nedocházelo k vzájemnému ovlivňování. Vlákna se potom střídají ve vyhodnocení modelu v jednotlivých souřadnicích a výpočet se tak stává sekvenční. Navíc je nutné počítat s časem pro přepnutí vláken. Tento čas je značný již při malé hustotě mřížky. Kritická sekce nám tak degradovala paralelní výpočet na sekvenční.

Jednou z možností, která nebyla zatím implementována je vytvoření samostatných instancí třídy pro jednotlivá vlákna. Výpočet by poté mohl fungovat stejně jako v MPI verzi a nedocházelo by ke zbytečným zdržením vlivem kritické sekce.

4.7 Paměťová náročnost

Jak bylo již na začátku řečeno, není základní test na paměťovou složitost příliš objektivní. Hlavním důvodem, proč tomu tak je, je rozdílná velikost nutně alokované paměti pro výpočet. Námi vytvořený program provádí méně činností než program HyperFun a potřebuje méně paměti pro datové struktury. Pokusíme se tedy uvést vše na pravou míru.

Tabulka činností ukazuje, pro co je nutné alokovat paměť v obou aplikacích a na co se tedy musíme zaměřit při přepočtu.

Pro co	HyperFun	Naše verze	Odhad
Hodnoty funkce na mřížce	ANO	ANO	k x m x n doubles
Vrcholy trojúhelníků	ANO	NE	3 x k x m x n doubles
Trojúhelníky	ANO	NE	3 x N integer

tabulka 7. Alokace paměti pro jednotlivé činnosti.

Tabulka ukazuje množství paměti pro jednotlivé operace. Ještě je nutné uvést, jak je to s N. Velikost N vychází z použité metody na vytváření trojúhelníků. Velikost N může být maximálně 8 x k x m x n. Ve skutečnosti budou hodnoty menší, neboť pouze velice komplikovaná data mohou způsobit, že model bude v každém voxelu.

Zaměříme-li se na graf 1 uvedený v kapitole 2, je jasné, že uvedené výsledky nejsou korektní. Pro uložení hodnot funkce ve mřížce musí alokovat oba programy stejné množství pamětí (používají-li stejný datový typ).

V našem případě jsme občas používali pouze datový typ Short, neboť zobrazování bylo prováděno přes modul MVE[1] zobrazující CT[2] data. Tato data jsou zaznamenána v datovém typu Short nebo Unsigned Short.

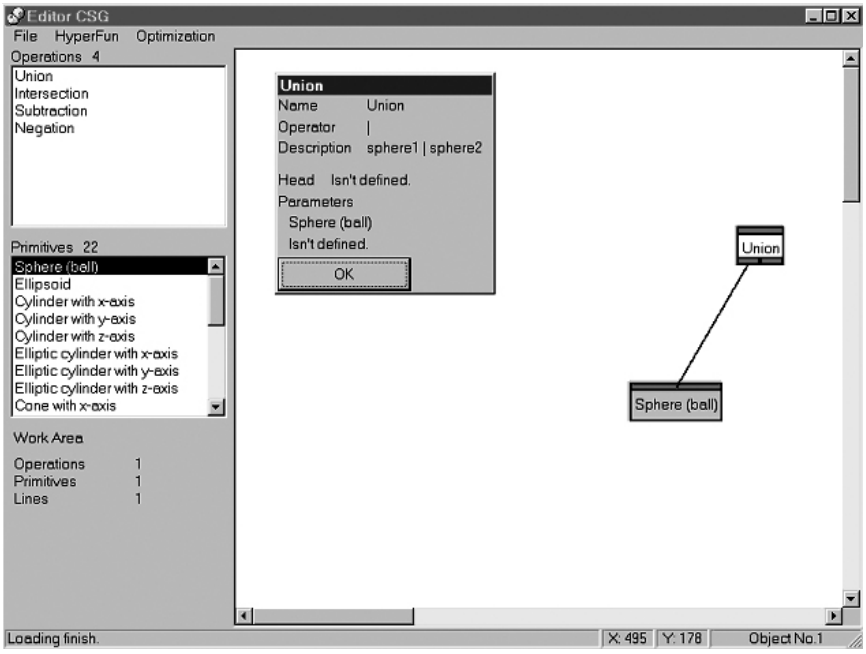
Rozdíl v alokované paměti tvoří položky uvedené v tabulka 3. Je tedy zřejmé, že paměťové rozdíly zde uvedené nejsou tak značné. Paměť potřebnou na struktury parseru není možné experimentálně stanovit, ale vzhledem k ostatním nárokům programu na paměť bude tato hodnota zanedbatelná.

4.8 Programová realizace

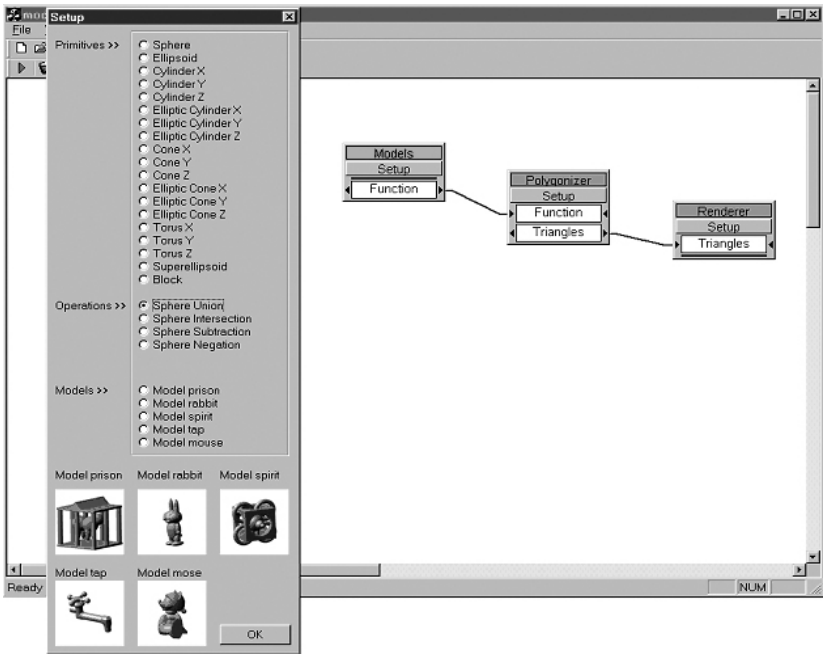
V rámci práce vzniklo několik programových realizací, z nichž některé byly určeny pro systém MVE a některé určené pro testování byly vytvořeny pouze jako samostatný program. Vznikly celkem dva moduly pro MVE. První modul je editor scén využívající metodu komplice modelu, ale modely není možné přidávat jako samostatné soubory. Model je vytvářen editací stromové struktury (obrázek 2). Druhý modul je čistě výpočetní a byl vytvořen jako testovací modul pro modul „Polygonizer“. Interně je v něm uloženo několik základních modelů, včetně pěti na kterých bylo prováděno veškeré testování [Hyper]. Je možné s v něm vybrat model, který má být spočítán, ale nelze v něm modely přidávat či ubírat (obrázek 3).

[1] Modular Visualisation Environment
[2] Computer Tomography

Jako samostatný program byl vytvořen editor scén pro HyperFun, který z vytvořené stromové struktury vygeneruje model v jazyce HyperFun, který může být zpracován. Samostatně byly vytvořeny i programy pro MPI a Thread verzi programu, včetně programů pro různé testy.



obrázek 2. Modul editor do MVE.



obrázek 3. Výpočetní modul do MVE.

5. Zhodnocení

Prokázali jsme, že naší metodou lze vytvořit shodný systém pro modelování pomocí implicitně definovaných těles jako zmíněný systém HyperFun. Není nutné vytvářet speciální parser na nově vzniklý jazyk HyperFun, pokud prokazatelně tento jazyk nepotřebuje konstrukce, které nejsou pomocí námi použitého jazyka C++ vytvořitelné. Navíc jazyk C++ nám poskytuje určité rozšíření funkcionality, které se značné rozšíří, pokud bude program platformě závislý. Pokud nám půjde o platformní nezávislost, budeme se muset spokojit s jazykem, jak ho definuje norma ANSI. Použitím standardního kompilátoru jsme získali kontrolní prvek schopný okamžitě detekovat chyby a možnost optimalizovat úlohu na definovanou platformu. Urychlení optimalizací je sice zanedbatelné vůči urychlení získaném použitím kompilátoru místo parseru, ale oblast testovacích úloh nebyla dostatečně široká na to, abychom s jistotou mohli říci, že v určitém konkrétním případě optimalizace nepomůže. Výhodou je také možnost měnit kompilátory s tím, že určitě existují i jiné kompilátory, které mohou být ještě rychlejší při kompilaci na danou platformu. Jistou nevýhodou je nutnost nějaký kompilátor vlastnit. Existuje ale množství kvalitních volně šířitelných kompilátorů. Určitě by měla být tato oblast zmapována, hlavně pokud by měl být tento systém volně šířitelný. Je nutné tímto směrem podotknout, že cílem nebylo se specializovat na konkrétní překladač, ale prozkoumat možnost jejich použití. Testováním bylo také ověřeno, že existují určité možnosti, jakým směrem by se mělo ubírat další urychlování. Hlavní směr je určen omezením prostoru, který není nutné počítat.

Shrneme-li dosažené výsledky, tak se uvedeným přístupem povedlo vytvořit systém, který pracuje stejně jako HyperFun, ale je minimálně dvakrát rychlejší (na prázdné mřížce) a podle vnitřní struktury modelu se urychlení pohybuje v intervalu od 4,4 do 15,42 (Microsoft Visual C++). Systém je funkční a dále rozšiřitelný.

Na závěr zmiňme ještě nesrovnalosti zjištěné v průběhu testování. Aktuální verze programu HyperFun by měla zvládat strukturované modely (obrázek 1), ale při testování se nepodařilo tuto skutečnost prokázat. Uvedený model je sice příklad udávaný v HyperFun projektu k otestování, avšak tento model nedokázal HyperFun zobrazit ani po opravení několika chyb, které se v něm vyskytují. Náš program vytvořený k otestování metody nemá s těmito strukturami problémy, neboť se jedná o jednoduché volání funkcí.

Reference

[Hyper] http://wwwcis.k.hosei.ac.jp/~F-rep/HF_proj.html

[Adzhi99] Adzhiev,V., Cartwright,R., Fausett,E., Ossipov,A., Pasko,A., Savchenko,V.: HyperFun project: Language and Software tools for F-rep Shape Modeling, Computer Graphics & Geometry, vol. 1, No. 10, 1999.

[Uhlir01] Uhlíř,K. (supervised by Skala,V.): Interaktivní systém pro generování implicitních funkcí a jejich modelování, Diplomová práce, Západočeská univerzita 2001.

[Cerm02] Cermak,M., Skala,V.: Space Subdivision for Fast Polygonization of Implicit Surfaces, submitted for publication 2002.

[Pasko88] Pasko,A.A., Pilyugin,V.V., Pokrovskiy,V.N.: Goemetric modeling in the analysis of trivariate functions, Computer & Graphics vol. 12, No. ¾, pp. 457-465,1988.

[Pasko95] Pasko,A., Adzhiev,V., Sourin,A., Savchenko,V.: Function representation in geometric modeling: concepts, implementation and applications, The Visual Computer, vol. 11, pp. 429-446,1995.

[Bloom97] Bloomenthal, J., et al.: Introduction to Implicit Surfaces, Morgan-Kaufman, 1997.

[Rousal] Rousal, M., Skala, V.: Modular Visuali-sation Environment MVE, Int.Conf. ECI 2000, Herlany, Slovakia, pp.245-250, ISBN 80-88922-25-9.

[Rigau99] Rigaudiere,D., Gesquiere,G., Faudot,D.: New Implicit Primitives Used in Reconstruction by Skeletons, WSCG 99, Czech Republic, Plzen, 1999.

[MPI94] MPI: A Message-Passing Interface Standard, University of Tennessee, Knoxville, Tennessee,1994.

[Ganter93] Ganter,M.,A., Storti,D.,W.: Impicit solid modeling: A renewed method for geometric design, Innovations in Engineering Design Education Resource Guide, 1993.

Příloha A

Primitiva

Sphere	hfSphere(x,center,R);
Ellipsoid	hfEllipsoid(x,center,a,b,c);
Cylinder	hfCylinderX(x,center,R);
Elliptic cylinder	hfEllCylX(x,center,a,b);
Cone	hfConeX(x,center,R);
Elliptic cone	hfEllConeX(x,center,a,b);
Torus	hfTorusX(x,center,R,r0);
Superellipsoid	hfSuperell(x,center,a,b,c,s1,s2);
Block	hfBlock(x,vertex,dx,dy,dz);
Blobby object	hfBlobby(x,x0,y0,z0,a,b,T);
Metaball	hfMetaball(x,x0,y0,z0,b,d,T);
Soft object	hfSoft(x,x0,y0,z0,d,T);
Solid noise	hfNoiseG(x,amp,freq,phase);
Convolution object with skeleton:	
Points	
Line segments	hfConvLine(x,begin,end,S,T);
Arcs	hfConvArc(x,center,radius,theta,axis,angle,S,T);
Triangles	hfConvTriangle(x,vect,S,T);
Curve	hfConvCurve(x,vect,S,T);
Mesh	hfConvMesh(x,vect,tri,S,T);
Object based on parametric functions:	
Cubic B-spline object	hfCubicBSplineF(x,l,m,n,bbox,ctr_pts);
Bezier spline object	hfBezierSplineF(x,l,m,n,bbox,ctr_pts);

Operace

Blending union	hfBlendUni(f1,f2,a0,a1,a2);
Blending intersection	hfBlendInt(f1,f2,a0,a1,a2);
Scaling	hfScale3D(xt,sx,sy,sz);
Shifting	hfShift3D(xt,dx,dy,dz);
Rotation	hfRotate3DZ(xt,theta);
Twisting	hfTwistZ(xt,z1,z2,theta1,theta2);
Stretching	hfSTretch3D(xt,x0,sx,sy,sz);
Tapering	hfTaperZ(xt,z1,z2,s1,s2);
Cubic space mapping	hfSpaceMapCubic(xt,original_points,delta_points,b);

Operátory aritmetické

Součet	+
Rozdíl	-
Součin	*
Podíl	/
Mocnina	^

Operátory geometrické

Union (sjednocení)	|
Intersection (průnik)	&
Subtraction (rozdíl)	\
Negation (negace)	~
Cartesian product	@

Matematické funkce
Unary functions:
'sqrt', 'exp', 'log', 'logd', 'sin', 'cos', 'tan', 'asin', 'acos', 'atan', 'abs', 'sinh', 'cosh', 'tanh', 'sign'.
Binary functions:
'max', 'min', 'atan2', 'mod'.

Logické výrazy
'<', '>', '<=', '>=', '=', '/='

Příloha B

Parametry programu HyperFun

```
hfp (FileName) <S>
        [-a <D,D,D....> (Parameters)]
        [-b <D> <D,D> <D,D,D> <D,D,D,D,D,D> (BoundingBox)]
        [-cf <I> <I,I,I> (FaceColor)]
        [-cl <I> <I,I,I> (LineColor)]
        [-d <I> (DisplayMode)]
        [-g <I> <I,I,I> (GridDensity)]
        [-h <> (Help)]
        [-i <D> (IsoValue)]
        [-o <S> (ObjectName)]
        [-s <D> (Search)]
        [-t <> (Time Report)]
        [-w <I> <I,I> (WindowSize)]
        [-wrl <S> (VRMLOut)]
        [-x <D/C,D/C,D/C....> (Mapping)]

(I: integer)
(D: double)
(C: character)
(S: string)
```

-a	Polem parametrů pro polygonizovaný objekt.
-b	Nastavení bounding boxu.
-cf	Barva ploch.
-cl	Barva hran.
-d	Mód zobrazení (1 – 7)
-g	Hustota mřížky pro polygonizer.
-h	Help.
-i	Nastavení iso-hodnoty.
-o	Název objektu.
-s	Nalezení vrcholu specifikovaného čísla.
-t	Časové informace o běhu programu.
-w	Velikost okna pro zobrazení výstupu.
-wrl	VRML výstup.
-x	Mapování souřadnic objektu.

Příklad použití:

hfp.exe **model.txt** –t –g 500 –b 15 –d 1

Příloha C

Datové struktury

```
-- třída slučující primitiva
class Frep
{
public:
  double fSphere (double x[],double ct[],double r);
  double fEllipsoid (double x[],double ct[],double a, double b,double c);
                .          .          .          .          .
};

-- třída slučující operace a modely
class FmodelDouble : public Frep
{
public:

  // operátory
  FmodelDouble& operator |(FmodelDouble &pA);      // sjednoceni - union
  FmodelDouble& operator &(FmodelDouble &pA);    // prunik - intersection
  FmodelDouble& operator %(FmodelDouble &pA);    // rozdil - subtraction
  FmodelDouble& operator +(FmodelDouble &pA);    // soucet - add
  FmodelDouble& operator ~();                    // negace - negation

  FmodelDouble operator =(double dX);      // rovnost  FmodelDouble = double
                .          .          .          .          .

  // operace
  double fBlendInt(FmodelDouble &f1,FmodelDouble &f2,double a0,double a1,double a2);
  double fBlendUni(FmodelDouble &f1,FmodelDouble &f2,double a0,double a1,double a2);
  double fRotate3DX(double p[],double theta);
                .          .          .          .          .

  // modely
  FmodelDouble Sphere(double x[]);    // sphere
                .          .          .          .          .

};

-- přetížený operátor sjednocení (union)
FmodelDouble& FmodelDouble::operator |(FmodelDouble &pA)
{
  if(ctFile->m_bSet) ctFile->m_lCUnion++;
  if(this->m_dRes < pA.m_dRes) return(pA);
  else return(*this);
}

-- primitivum koule
double Frep::fSphere (double x[],double ct[],double r)
{
  double mx = x[0] - ct[0], my = x[1] - ct[1], mz = x[2] - ct[2];
                double x2 = mx*mx, y2 = my*my, z2 = mz*mz, r2 = r*r;
                return (r2 - x2 - y2 - z2);
}
-- model
FmodelDouble FmodelDouble::Sphere(double x[])
{
 double sp_c[3];
 FmodelDouble  sp1,sp2,sp3,sp_final;

 sp_c[0] = 0; sp_c[1] = 0; sp_c[2] = 0;
 sp1 = fSphere(x, sp_c, 4);
 sp2 = fSphere(x, sp_c, 5);
 sp3 = fSphere(x, sp_c, 6);
 sp_final = (((sp1 & sp2) | (sp3 % sp1)) & sp2);
 return(sp_final);
}
```

Vyhodnocení – základní verze

```
void CountFunc::Count(int iModel)
{
 FmodelDouble fmOut;
 double dStepX,dStepY,dStepZ;
 long   lStart,lEnd;
 double dCoor[3];

 switch(iModel)
 {
  case 1: P_MODEL = &FmodelDouble::prison;        break;
  case 2: P_MODEL = &FmodelDouble::rabbit;        break;
            .            .            .            .
 }

 dStepX = dStepY = dStepZ = (double)m_sMinBound;
 m_lNumVal = 0;
 for(long i=0;i<m_lSlidesZ;i++){
  for(long j=0;j<m_lSlidesY;j++){
   for(long k=0;k<m_lSlidesX;k++){
    dCoor[0] = dStepX; dCoor[1] = dStepY; dCoor[2] = dStepZ;

          -- vyhodnocení modelu
    fmOut = (fmdWork.*P_MODEL)(dCoor);
          -- uložení hodnot
    *(m_dData + m_lNumVal) = fmOut.m_dRes;

    m_lNumVal++;
    dStepX += m_dStep;
   }
   dStepX = (double)m_sMinBound;
   dStepY += m_dStep;
  }
  dStepY = (double)m_sMinBound;
  dStepZ += m_dStep;
 }
}
```

Vyhodnocení – MPI verze

```
-- základní část pro inicializaci a vytvoření procesu MPI
int _tmain(int argc, TCHAR* argv[], TCHAR* envp[])
{
 int         nRetCode = 0;
 int         iMyRank,iNumProc;
 double dVector[4];
 char        chProcName[50];
 int         iSize;
 MPI_Status status;

 MPI_Init(&argc,&argv);

 MPI_Comm_size(MPI_COMM_WORLD,&iNumProc);
 MPI_Comm_rank(MPI_COMM_WORLD,&iMyRank);
 if(iMyRank == 0)
 {
  dVector[0] = (double)iNumProc;  // počet procesorů
  dVector[1] = 190;               // dělení v jedné ose
  dVector[2] = 15;               // rozsah hodnot na ose
  dVector[3] = 1;                // vybrané číslo modelu

  // rozeslání informací všem procesorům
  for(int i=1;i<iNumProc;i++)
  {
   MPI_Send(dVector,4,MPI_DOUBLE,i,99,MPI_COMM_WORLD);
  }
 }
 else
```

```
  {
   MPI_Recv(dVector,4,MPI_DOUBLE,0,99,MPI_COMM_WORLD,&status);

   // výpočet modelu
   // init function
   MPI_Get_processor_name(chProcName,&iSize);
   printf("\n%s",chProcName);
   cfMake->InitFunc(dVector,iMyRank);
   // ************************************
   cfMake->Count((short)dVector[3]);
   // ************************************
   cfMake->FreeMem();
  }

  MPI_Barrier(MPI_COMM_WORLD);
  MPI_Finalize();
  return nRetCode;
}
```

-- funkce Count je shodná, pouze prostor se rozdělí na shodně velké časti podle počtu dostupných procesorů

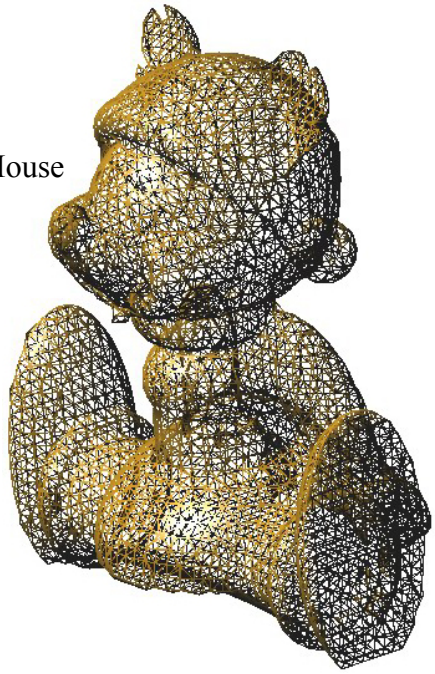
Spuštění: MPIRun.exe -np 2 „umístění spustitelného souboru“

- -np … na kolika počítačích se má provádět výpočet
- spustitelný soubor musí být umístěn na síťovém disku, který je přístupný ze všech počítačů, které se účastní výpočtu
- program spolupracuje s utilitou MPD Configuration Tool, která detekuje počítače, na kterých běží MPI client, a kde lze určit na jaké počítače se bude úloha distribuovat

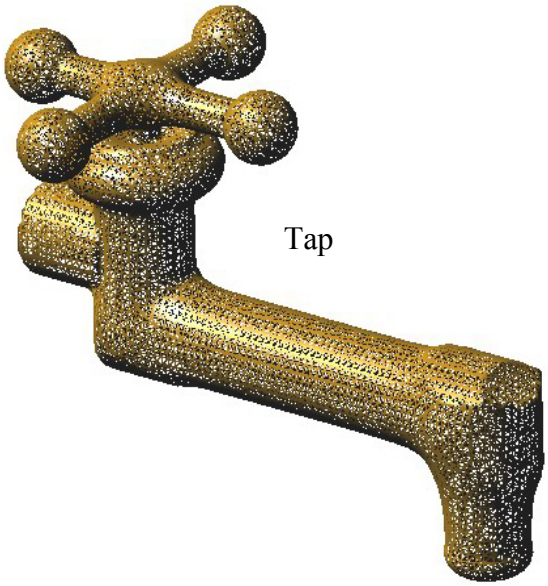
Příloha D

Modely

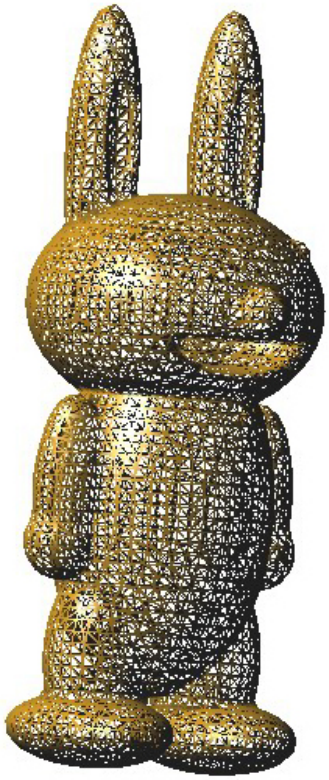
Mouse



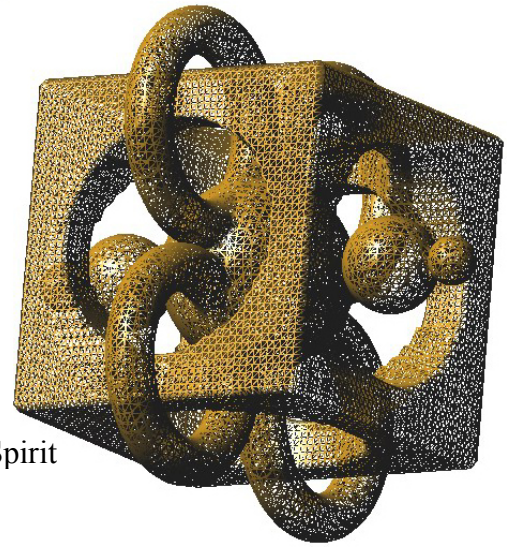
Tap



Prison



Rabbit



Spirit

Příloha E

Volání objektů

	Prison	Rabbit	Spirit	Tap	Mouse
Operace	M1	M2	M3	M4	M5
Union	17	15	11	11	38
Intersection	20	2	1	16	9
Subtraction		3	4		30
Add			2		
Negation					1
Eqq	24	20	19	36	71
Power					
BlendIntersec					
BlendUnion				4	3
Rotate3DX					5
Rotate3DY				1	2
Rotate3DZ					
Shift					3
Scale					5

Objekty					
Sphere	3	5	5	4	5
Ellipsoid	4	15			19
CylinderX			1	4	
CylinderY			1	3	1
CylinderZ			1	1	2
EllipticCylX					
EllipticCylY					
EllipticCylZ					
ConeX					
ConeY					
ConeZ					
EllConeX					
EllConeY					
EllConeZ					
TorusX			4		4
TorusY				2	6
TorusZ					3
Superellip					
Block	2		2		

Operace	61	40	37	68	167
Objekty	9	20	14	14	40