České vysoké učení technické v Praze
Fakulta elektrotechnická

Diplomová práce

Maya – Bezierovy křivky a plochy

Bc. Radek Tykal

Vedoucí práce: Ing. Jaroslav Sloup

Studijní program: Elektrotechnika a informatika, dobíhající

Obor: Výpočetní technika (magisterský)

leden 2007

Poděkování

Na tomto místě bych rád poděkoval Ing. Jaroslavu Sloupovi, vedoucímu mé diplomové práce, za vstřícnost, ochotu a čas věnovaný této práci.

Prohlášení

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.
Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 12. 1. 2007 ....

Abstract

The goal of this thesis is to design and implement a plug-in that will extend the modelling program Maya with a possibility of modelling through Bézier curves and surfaces. This plug-in should contain tools for converting these curves and surfaces to NURBS and back. The reason of this extension is a creation of tools for modelling with parametrical curves and surfaces that are simpler then standard Maya's tools for NURBS.

The following text describes problems related to a creation of such a plug-in. Basic principles of Maya and its application programming interface are also explained, because it is not possible to do the design without this knowledge. Convert algorithms are also described. These algorithms are the most important part of tools for a conversion between mentioned types of curves and surfaces. It is necessary to explain the relation between Bézier curves or surfaces and NURBS for the realization of the convert tools. There are discussed two possible solutions in the text, especially new data structures representation. The chosen implementation is subsequently described in detail and final tools are compared with Maya's tools for NURBS. Additional possibilities of a development of this work are listed in the final part.

Abstrakt

Cílem této diplomové práce je návrh a implementace programového modulu, který rozšíří modelovací program Maya o možnost modelování pomocí Bézierových křivek a ploch. Součástí modulu mají být nástroje umožňující konverzi těchto křivek a ploch na NURBS a zpět. Důvodem tohoto rozšíření je vytvoření nástrojů pro práci s parametrickými křivkami a plochami, které budou jednodušší než standardní nástroje Mayi pro NURBS.

Následující text popisuje problematiku vytváření takového modulu. Vysvětleny jsou také základní principy Mayi a jejího aplikačního programového rozhraní, jelikož se bez těchto znalostí nelze při návrhu obejít. Popsány jsou zde také převodní algoritmy, které jsou nejdůležitější částí nástrojů pro konverzi mezi zmiňovanými typy křivek a ploch. Pro realizaci převodních nástrojů je také potřeba vysvětlit vztah Bézierových křivek resp. ploch a NURBS. Dále jsou v textu diskutovány dvě možné varianty řešení, zejména co se týče reprezentace nových datových struktur. Zvolená varianta implementace je následně detailně popsána a výsledné nástroje jsou porovnány s nástroji Mayi pro NURBS. Závěrem jsou uvedeny návrhy na další možný rozvoj této práce.

Obsah

Seznam obrázků **xii**

Seznam tabulek **xiii**

1 Úvod **1**

2 Základní principy a pojmy **3**
2.1 Architektura Mayi 3
2.1.1 DG uzly 4
2.1.2 DAG uzly 7
2.1.3 Aktualizace DG grafu 11
2.2 MEL 14
2.3 C++ API 14

3 Popis problému a vymezení cílů **17**
3.1 Požadavky na výsledné řešení 19
3.1.1 Geometrické objekty 19
3.1.2 Ostatní DG uzly 20
3.1.3 Geometrická data 20
3.1.4 Příkazy a nástroje 21
3.2 Přehled cílů 21

4 Analýza a návrh řešení **23**
4.1 Volba rozhraní 23
4.1.1 C++ API 23
4.1.2 Jazyk MEL 24
4.2 Vztah Bézierových křivek a ploch k NURBS 25
4.3 Volba způsobu reprezentace dat 25
4.3.1 Využití dat NURBS 25
4.3.2 Vytvoření nových dat 26
4.4 Převodní algoritmy 26
4.4.1 Bézier – NURBS 26
4.4.2 NURBS – Bézier 27
4.5 Návrh modulu 30
4.5.1 Základní struktura modulu 30
4.5.2 Návrh DG uzlů 31
4.5.3 Návrh dat 34
4.5.4 Návrh příkazů a kontextu 36
4.5.5 Návrh modifikací GUI 39

5 Popis implementace **43**
5.1 Základní struktura programu 43
5.2 Přehled a popis použitých MPx tříd 43
5.3 Implementace jednotlivých částí 45
5.3.1 Shape uzly křivek a ploch 45
5.3.2 Geometrická data 49

5.3.3 Ostatní DG uzly 51
5.3.4 Příkazy 52
5.3.5 Kontext 53
5.3.6 Úprava GUI 54

6 Zhodnocení a návrh dalšího rozvoje práce 57
6.1 Funkčnost modulu a jeho částí 57
6.2 Další možná rozšíření práce 60

7 Závěr 63

8 Seznam literatury 65

A Seznam použitých zkratek 67

B Instalační a uživatelská příručka 69

C Obsah přiloženého CD 73

Seznam obrázků

2.1 Rozdělení systému Mayi do vrstev. 3
2.2 Příklad struktury *Dependecy Graph* reprezentujícího scénu. 4
2.3 Základní struktura obecného *Dependecy Graph* uzlu. 5
2.4 Atributy uzlu pro výpočet geometrie kružnice. 5
2.5 Příklad uzlu s atributem typu pole obsahujícího složené atributy. 6
2.6 Korektní napojení atributů. 7
2.7 Nedovolené napojení atributů. 7
2.8 Příklad struktury *Dependecy Graph* obsahující DG i DAG uzly. 8
2.9 Vztah *transform* a *shape* uzlu. 8
2.10 Reprezentace NURBS plochy pomocí *Dependecy Graph*. 9
2.11 Příklad hierarchie *Dependecy Graph* uzlů krychle. 10
2.12 Reprezentace kopií uzlu v *Dependecy Graph*. 10
2.13 Reprezentace instancí uzlu v *Dependecy Graph*. 11
2.14 Ukázka výpočtu v *push–pull* modelu – počáteční konfigurace. 11
2.15 Ukázka výpočtu v *push–pull* modelu – možné stavy uzlu. 12
2.16 Ukázka výpočtu v *push–pull* modelu – průběh propagace příznaku 12
2.17 Ukázka výpočtu v *push–pull* modelu – aktualizace uzlu D 12
2.18 Příklad skutečného propojení *Dependecy Graph* uzlů. 13
2.19 Propagace příznaku mezi *Dependecy Graph* uzly. 13
2.20 Struktura programového rozhraní Mayi. 15
2.21 Přístup k jádru Mayi pomocí rozhraní třídy *MObject*. 16

3.1 Princip generování geometrie. 17
3.2 Ukázka rozšíření dvou křivek na plochu – rozšiřované křivky. 18
3.3 Ukázka rozšíření dvou křivek na plochu – výsledná plocha. 18
3.4 Princip funkce konverzního uzlu. 19

4.1 Příklad rozkladu NURBS křivky – původní křivka. 27
4.2 Příklad rozkladu NURBS křivky – vložení prvního uzlu. 28
4.3 Příklad rozkladu NURBS křivky – vložení druhého uzlu. 28
4.4 Příklad rozkladu NURBS křivky – příprava na druhý segment. 29
4.5 Příkaz vložení čtverce – struktura *Dependecy Graph* vč. hierarchie. . . . 36
4.6 Příkaz vložení plochy – struktura *Dependecy Graph* vč. hierarchie. 37
4.7 Příkaz vložení krychle – struktura *Dependecy Graph*. 37
4.8 Příkaz rozšíření dvou křivek – struktura *Dependecy Graph*. 37
4.9 Příkaz převodu Bézier–NURBS – struktura *Dependecy Graph*. 38
4.10 Příkaz převodu NURBS–Bézier – struktura *Dependecy Graph*. 38
4.11 Příkaz převodu NURBS–Bézier – hierarchie uzlů. 39

5.1 Příklad vykreslené křivky včetně řídících bodů a obalu. 49
5.2 Příklad vykreslené vystínované plochy včetně řídících bodů a obalu. . . . 50
5.3 Převod NURBS křivky – reprezentace v *Dependency graph*. 52
5.4 Ukázka špatného převodu NURBS kružnice a principu opravy. 52
5.5 Vytvořené menu obsahující nové příkazy – první část. 55
5.6 Vytvořené menu obsahující nové příkazy – druhá část. 55
5.7 Vytvořené přihrádky pro Bézierovy křivky a plochy. 56

6.1 Vystínovaná NURBS plocha – normály ploch. 58
6.2 Vystínovaná Bézierova plocha – normály vrcholů. 59
6.3 Ilustrace problému s texturováním při převodu NURBS–Bézier. 60

B.1 Manažer modulů – okno. 70

C.1 Struktura přiloženého CD. 73

Seznam tabulek

2.1 Prefixy jmen tříd poskytovaných C++ API. 15

4.1 Přehled atributů navržených uzlů. 35
4.2 Přehled navržených příkazů. 40

5.1 Přehled obecných vlastností atributů. 46
5.2 Přehled režimů vykreslování. 48
5.3 Přehled MEL příkazů použitých k modifikaci GUI. 55

1 Úvod

Program Maya je nejen velice vyspělý 3D modelář, ale také výborný vizualizační a animační nástroj. Jeho architektura je otevřená, veškerou práci lze realizovat pomocí skriptů, naprogramovat prostřednictvím vestavěného skriptovacího jazyka nebo lze využít aplikační programové rozhraní (API). Z těchto důvodů je hojně využívána například filmovými grafiky, vývojáři počítačových her nebo profesionálními návrháři. Pro podrobnější přehled všech vlastností a možností využití se lze podívat na [2].
Nástrojů, které lze při práci v Maye využívat, je celá řada. Jsou zde k dispozici nástroje pro animace, deformace, efekty a mnoho dalších. Pro přehled kompletních možností odkazuji čtenáře na [1]. Mezi nejzajímavější z hlediska této práce patří nástroje pro modelovaní. Maya umožňuje vytvářet polygony, NURBS křivky a plochy a plochy, které jsou kombinací polygonů a NURBS ploch (*subdivision*). Právě zmíněné nástroje pro NURBS jsou velice robustní a lze pomocí nich vymodelovat téměř cokoli. Díky této univerzálnosti jsou ovšem pro některé aplikace zbytečně složité a rozsáhlé. To vedlo k myšlence navrhnou a implementovat modul pro práci s Bézierovými křivkami a plochami. Ty jsou v porovnání s NURBS jednodušší (jsou vlastně jejich podmnožinou) a proto mohou být v některých situacích vhodnější.
Implementace požadovaného modulu vyžaduje poměrně dobrou znalost základních konceptů Mayi. Jako u každého jiného systému jsou totiž i zde jistá omezení, která je nutno respektovat. Vzhledem k faktu, že systém Mayi je velice flexibilní, je možné řešit jednu úlohu mnoha způsoby. Kapitola 2 vysvětluje právě ty základní principy, které jsou z hlediska této práce důležité, zavádí také některé pojmy používané v dalších kapitolách. Kapitola 3 poté na základě těchto znalostí popisuje problém a vymezuje základní cíle práce.
Otevřená architektura Mayi dává obrovský prostor k modifikování a rozšiřování jejích vlastností, díky čemuž si lze vytvořit nástroje a pomůcky uzpůsobené konkrétním požadavkům. Dokonce lze Mayu přetvořit v něco na první pohled zcela odlišného. Naprogramovat lze v Maye řadu věcí. Následujících několik odstavců popisuje hlavní možnosti jak ji měnit a řídit prostřednictvím programových rozhraní, díky čemuž je možné si udělat představu o jejích možnostech.
Celé grafické rozhraní Mayi je naprogramováno a řízeno prostřednictvím vestavěného jazyka, který se nazývá MEL (*Maya Embedded Language*). Přidávání, odebírání a úpravy elementů grafického rozhraní jsou realizovány pomocí příkazů MELu a uživatel tedy může vše uzpůsobit svým potřebám a požadavkům. Standardní uživatelské rozhraní je možné dokonce zcela nahradit a změněné rozhraní lze svázat s určitými projekty či uživateli.
Maya je často používána jako nástroj uprostřed nějakého výrobního řetězce, přičemž je potřeba načítat a ukládat data v odlišném formátu, než Maya standardně podporuje. Pomocí aplikačního programového rozhraní lze napsat tzv. translátory, díky kterým není import či export dat zcela odlišných formátů žádný problém.
Při práci se mnoho úkonů opakuje, což je vhodné řešit programováním v MELu a úkony tak zautomatizovat. Díky vytvořeným MEL skriptům pak odpadá ruční provádění opakovaných činností a dojde k podstatnému urychlení a usnadnění práce.
Pro tuto diplomovou práci je nejpodstatnější možnost Mayu rozšiřovat zejména pomocí jejího C++ API, případně jazyka MEL. Pro tvorbu nových nástrojů a přidávání vlastností existuje jen velice málo omezení. Výsledné nástroje nejsou dokonce z uživatelského hlediska odlišitelné od standardních. Volba mezi těmito rozhraními, tedy MELem a C++

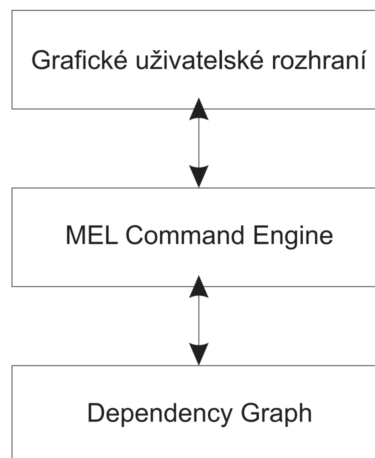
API, je v podstatě na uživateli. Jsou zde však jistá omezení, jelikož některé funkcionality nelze v MELu narozdíl od C++ API nalézt a naopak. Výběr jednoho rozhraní ovšem nevylučuje použití druhého, některé problémy lze dokonce řešit pouze kombinací obou. Podrobněji se rozdíly mezi rozhraními zabývá kapitola 4, ve které je také uvedena volba rozhraní pro konkrétní problémy. Jak již bylo naznačeno, modul lze v podstatě realizovat dvěma způsoby. Výběr varianty je z hlediska implementace vcelku zásadní a proto je potřeba ho dobře zvážit. Rozbor obou možností je v této kapitole uveden. Jelikož mezi požadavky patří i realizace konverzních nástrojů, obsahuje tato kapitola rozbor vztahu obou typů křivek a ploch. Dále jsou zde navrhnuty potřebné převodní algoritmy.
Zvolená varianta implementace je detailně popsána v kapitole 5, zejména se zaměřením na problematické části. Třídy a metody C++ API jsou sice téměř kompletně popsány, ale ne u všech metod je zřejmé jak je implementovat a co je v nich potřeba provést tak, aby bylo vše korektní.
Poslední dvě kapitoly v podstatě shrnují dosažené výsledky. Je zde uvedeno zhodnocení nových nástrojů, porovnání se standardními nástroji Mayi pro práci s NURBS a také návrhy na další možná rozšíření práce.

2 Základní principy a pojmy

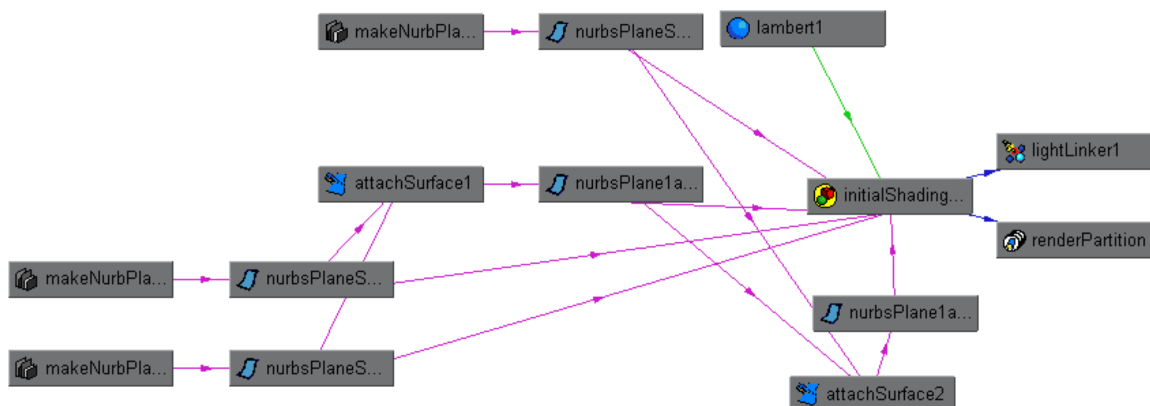
Pochopení základních konceptů Mayi je velice důležité, jelikož se jedná o flexibilní systém, který neurčuje nějaký pevný rámec pro řešení úloh. S tím totiž souvisí skutečnost, že Maya dovoluje řešit problémy tzv. špatným způsobem, což se může projevit zdlouhavým laděním a hledáním chyb. Následující text, vycházející z knihy [5], nejprve popisuje hlavní principy, které je potřeba si osvojit před vlastním návrhem modulu, dále vysvětluje práci s C++ API a také s jazykem MEL. Ukázkové příklady ke zmíněné knize lze nalézt na [4]. Pro první seznámení s problematikou pragramováni Mayi doporučuji stránky [9].

2.1 Architektura Mayi

Systém Mayi lze rozdělit do tří hlavních částí, jak ukazuje obr. 2.1. Jakmile uživatel nějakým způsobem pracuje s uživatelským rozhraním (výběr položek menu, posun objektů, změna parametrů atd.), Maya ve skutečnosti vytváří MEL příkazy. Ty jsou dále posílány do struktury nazvané *MEL Command Engine*, kde jsou interpretovány a provedeny. Většina těchto příkazů nějakým způsobem operuje s grafem scény označovaným jako *Dependency Graph*. *Dependency Graph* vlastně představuje celou scénu, tedy veškerá její data a informace potřebné pro popis 3D světa (objekty, materiály, animace atd.). To ovšem není vše, tento graf zároveň definuje způsob, jakým jsou zmíněná data zpracovávána. *Dependency graph* je vlastně srdcem Mayi.
Jádro Mayi je implementováno pomocí modelu označovaného jako tok dat (*data flow*), který je podrobněji popsán např. v [5]. Tento model vychází z představy, že data, vytvářená například nějakou 3D aplikací, vznikají sérií určitých operací. Výstup jedné operace je vstupem pro následující, přičemž každá operace data svým způsobem mění. Vzniká tak dojem toku dat jednotlivými operacemi. Na model lze tedy pohlížet jako na rouru (*pipeline*) skládající se z jednotlivých operátorů, přičemž data jsou vložena do prvního operátoru a získaná z posledního po určitém zpracování. Pokud toto dále zobecníme, dostaneme se ke struktuře na jejímž jednom konci vložíme libovolná data a na druhém konci získáme jakákoli jiná data (nemusí se ani jednat o grafická data).



Obrázek 2.1: Rozdělení systému Mayi do vrstev.



Obrázek 2.2: Příklad struktury *Dependecy Graph* reprezentujícího scénu.

Dependency Graph

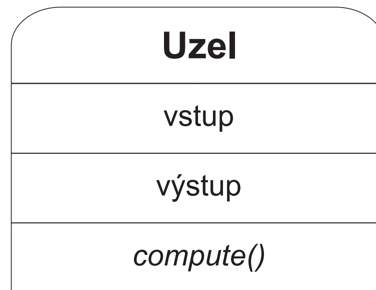
Tím se již dostáváme ke zmiňovanému grafu nazvanému *Dependency Graph*, zkráceně DG. Jádro Mayi je totiž fyzicky obaleno do DG. Tento graf poskytuje veškeré stavební bloky umožňující vytvářet zmíněnou strukturu, která dovoluje zpracovat jakákoli data do ní vložená. Bloky, o kterých se mluví jako o uzlech (*nodes*), v sobě zapouzdřují data i operace. Každý uzel tedy vlastní několik slotů obsahujících data používaná Mayou a také operátor, který z dat dokáže vypočítat nějaká jiná výstupní data.
Díky tomuto mechanismu lze pouhým zřetězením uzlů řešit téměř jakékoli grafické úlohy. Uzly mohou s daty dělat libovolné operace, mohou dokonce vytvářet zcela nová data. Každý uzel je navržen tak, aby prováděl pouze malou množinu odlišných operací. Složitější operace jsou tedy řešeny celou sítí jednoduchých uzlů. Složitost propojení mezi uzly není nijak omezena. Maya využívá tohoto principu k řešení všech potřebných úloh (modelování, dynamika, stínování, vykreslování atd.).

Scéna

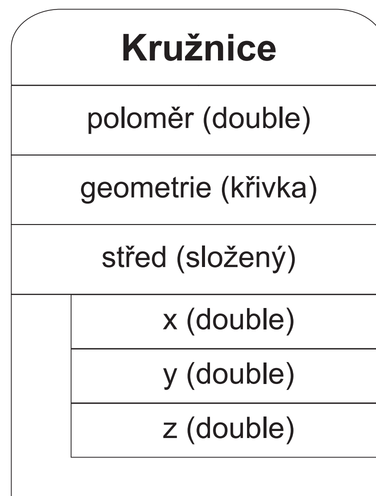
Podle popisu DG grafu je zřejmé, že vlastně popisuje kompletně celou scénu, tudíž je jasné, že DG graf je vlastně scéna. Jelikož veškerá data (např. rozmístění všech kamer a světel ve scéně) nejsou uložena centrálně, nýbrž v propojené síti uzlů, bylo by pro uživatele velice náročné tato data získat. Maya ovšem obsahuje pomocné funkce, které uživatele odstiňují od zmíněného problému. Příklad kompletního DG grafu reprezentujícího celou scénu je na obr. 2.2. Na obrázku je vidět graf složený z uzlů různých druhů, čáry značí propojení a šipky jejich směr.

2.1.1 DG uzly

Nyní je již zřejmé, že rozšíření Mayi je možné prostřednictvím přidávání nových typů uzlů. K tomu, aby bylo možné nějaký uzel navrhnout a implementovat je ovšem ještě potřeba pochopit jeho strukturu. Na obr. 2.3 je schématicky naznačena struktura obecného uzlu. Ten obsahuje několik atributů (uzel na obrázku má 2 a ty jsou nazvány *vstup*, *výstup*) a dále má metodu *compute()*, což je výpočetní funkce zodpovědná za výpočet jednoho a více výstupních atributů z jednoho a více vstupních atributů.



Obrázek 2.3: Základní struktura obecného *Dependecy Graph* uzlu.



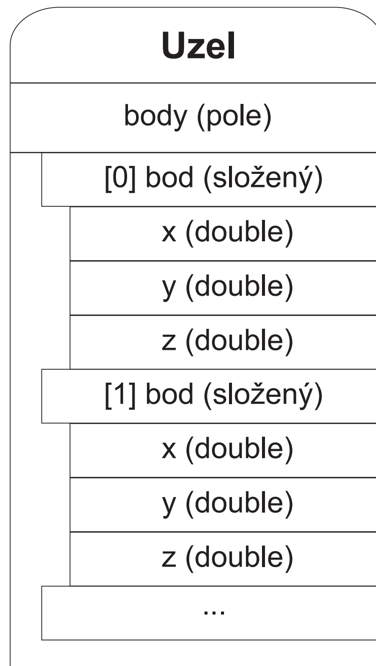
Obrázek 2.4: Atributy uzlu pro výpočet geometrie kružnice.

Atributy

Veškerá data jsou v uzlech uložena v podobě již zmíněných atributů, z tohoto důvodu má každý atribut jméno a datový typ definující druh dat v atributu. Podporovány nejsou jen základní datové typy (*long*, *double*, *boolean* atd.), ale i komplexní typy (např. obsahující celou geometrii NURBS křivky či plochy). Dále je možné vytvářet složené atributy, ty se pak skládají z dalších atributů. Jako příklad si lze představit uzel na obr. 2.4, který slouží k výpočtu geometrie kružnice. Uzel obsahuje jeden jednoduchý atribut *poloměr*, datového typu *double*, jeden složený atribut *střed*, ten obsahuje tři jednoduché atributy *x*, *y*, *z* typu *double*, a také jeden atribut *geometrie* obsahující geometrii NURBS křivky. Na první pohled je jasné, že atributy *poloměr* a *střed* ovlivňují pozici a velikost výsledné kružnice, která je uložena v atributu *geometrie*. Kromě složených atributů lze dále vytvářet pole atributů. Vše lze i kombinovat, jak ukazuje obr. 2.5.

Výpočetní funkce

O výpočet geometrie ze zadaného středu a poloměru se, u uzlu na obr. 2.4, stará dříve zmíněna výpočetní funkce. Ta je vlastně jádrem uzlu, jelikož je zodpovědná za veškerou jeho činnost. Obecný popis výpočetní funkce vypadá takto:



Obrázek 2.5: Příklad uzlu s atributem typu pole obsahujícího složené atributy.

výstup = compute(vstup1, ..., vstupN)

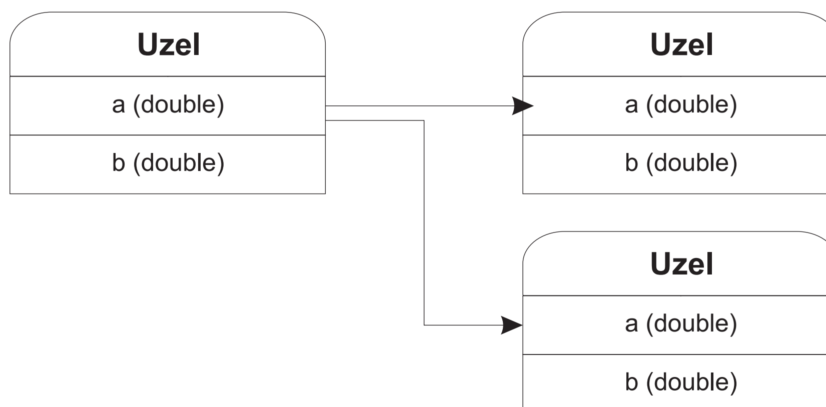
Důležitá je skutečnost, že jednotlivé vstupní a výstupní atributy jsou v uzlu lokální. Výpočetní funkce tedy bere v úvahu pouze data daného uzlu, nikdy data jiných uzlů. Z hlediska výpočtů v DG grafu je vnitřní chování uzlu skryté. Podstatné je, že pokud mu předáme požadovaná vstupní data, získáme data výstupní, přičemž způsob výpočtu je vedlejší. Z vnějšího pohledu je tedy uzel určen pouze jeho vstupními a výstupními atributy, ty definují vlastně jeho rozhraní. Maya ve skutečnosti nerozlišuje zda je atribut vstupní či výstupní, z jejího pohledu jen vlastní data. Logické rozlišení je ovšem potřeba zavést, jelikož některé atributy musím obsahovat data vstupující do výpočetní funkce a jiné zase data získaná výpočtem.

Závislé atributy

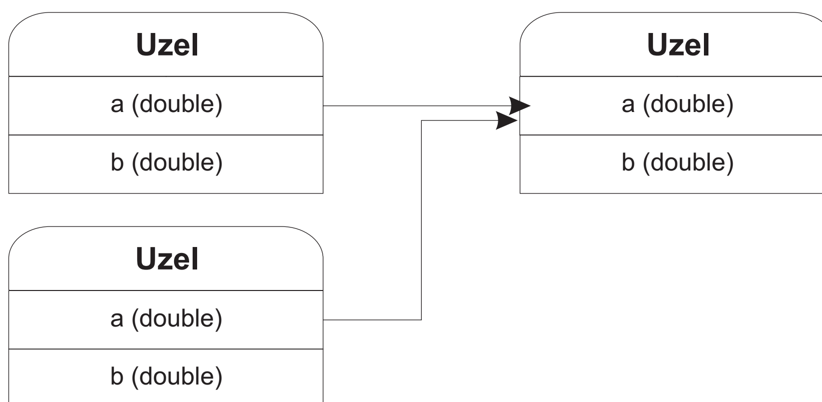
Pokud se vrátíme k příkladu uzlu na obr. 2.4, vidíme, že atribut *geometrie* je vypočítán na základě atributů *poloměr* a *střed*, tedy:

geometrie = compute(poloměr, střed)

Je zde tedy patrný vztah mezi dvojicemi atributů *geometrie* – *poloměr* a *geometrie* – *střed*. Vezmeme-li první z nich, můžeme říci, že atribut *geometrie* je závislý na atributu *poloměr*. Tuto závislost musí každý uzel definovat, jelikož díky ní Maya pozná, jaké atributy získá pomocí výpočtu a jaké atributy je potřeba k tomuto výpočtu použít. Definicí závislostí atributů je tedy stanoveno, jak uzel generuje data. Pokud je požadována hodnota atributu *geometrie*, je zavolán výpočet z hodnot atributů *poloměr* a *střed*. Pokud se změní hodnota alespoň jednoho z atributů *poloměr* nebo *střed*, atribut *geometrie* je rovněž přepočítán.



Obrázek 2.6: Korektní napojení atributů.



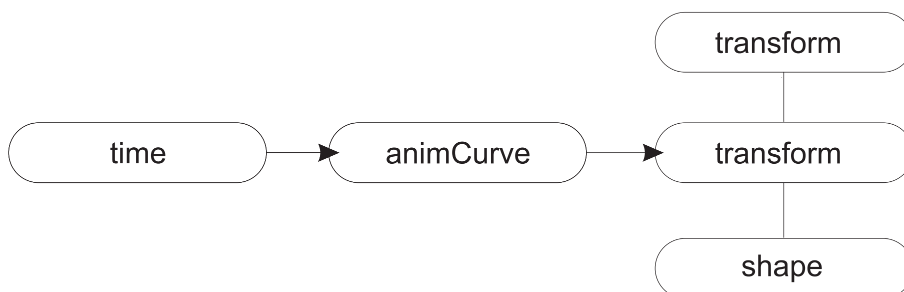
Obrázek 2.7: Nedovolené napojení atributů.

Napojování uzlů

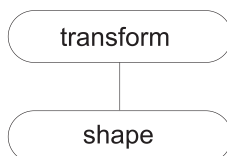
Nyní, když víme jakým způsobem uzly pracují, zbývá popsat pravidla jejich propojování. Dá se předpokládat, že uzly jsou mezi sebou propojovány přes atributy. Výstupní atributy jednoho uzlu dodávají data vstupním atributům jiných uzlů, jejich výstupní atributy pak vstupním atributům dalších uzlů atd. Propojené atributy musí být stejného datového typu s výjimkou několika implicitních konverzí, kterými Maya disponuje. Jeden výstupní atribut lze napojit na libovolné množství vstupních, ne však opačně. V tomto případě by totiž nemohla být hodnota vstupního atributu vyhodnocena. Příklad korektního napojení atributů je na obr. 2.6, nesprávného pak na obr. 2.7. Důležité je podotknout, že žádný uzel nemá ponětí o svém okolí, neví tedy zda a s jakým uzlem je propojen. Každý uzel si je vědom pouze svých atributů, o tok dat se stará Maya. Právě tento princip lokality umožňuje používat uzly jako stavební bloky.

2.1.2 DAG uzly

V počítačové grafice je velmi často používáno hierarchické uspořádání objektů. Např. představíme-li si model auta s koly, kola by měla být potomky auta, aby se zajistil jejich pohyb spolu s celým autem. DG graf ovšem tuto hierarchii neumožňuje. Je potřeba



Obrázek 2.8: Příklad struktury *Dependecy Graph* obsahující DG i DAG uzly.



Obrázek 2.9: Vztah *transform* a *shape* uzlu.

zavést zvláštní uzly, označované jako *Directed Acyclic Graph* (DAG) uzly. *Directed Acyclic Graph* je technický výraz pro hierarchii, ve které se uzel nemůže vyskytovat na dvou místech pokud procházíme hierarchii shora dolů. Důležité je podotknout, že DAG uzel je vlastně DG uzel. DAG uzly jsou součástí DG grafu stejně jako každý jiný uzel, je jen specializovaný s ohledem na zavedení vztahu rodič–potomek.
Obr. 2.8 ukazuje, jak může vypadat DG graf s oběma typy uzlů. *Transform* a *shape* jsou DAG uzly, kdežto *time* a *animCurve* jsou DG uzly. Tento graf reprezentuje nějaký objekt, jeho geometrie je v uzlu *shape*, a ten je umístěn do scény pomocí dvou transformací (první *transform* má za potomka druhý *transform* a ten má za potomka *shape*). Druhý uzel *transform* je nějakým způsobem řízen uzlem *animCurve*, jehož výstupní hodnota se mění v čase (uzel *time*). Jedná se tedy o nějakou animaci. Na tomto příkladu je vidět, že graf lze číst ze dvou směru. Shora dolů od kořenových uzlů *transform* až k uzlům *shape*. Tyto uzly tedy tvoří stromovou hierarchickou strukturu. Druhý způsob čtení je zleva doprava, kde jsou horizontálně propojeny uzly přes jejich atributy.
Pro úplnost je potřeba uvést, že hierarchie nemá striktně stromovou strukturu, jeden DAG uzel zde totiž může mít i více rodičů. To je používáno při vytváření instancí, k tomu se ale dostaneme později.

Uzly tansform a shape

Pokud v Maye vytvoříme libovolný 3D objekt (NURBS plocha či křivka, polygon atd.), ve skutečnosti vznikne kombinace *transform* a *shape* uzlu, jak lze vidět na obr. 2.9. *Transform* uzel definuje pozici v prostoru, *shape* potom formu objektu. V Maye vypadá tato struktura přibližně podle obr. 2.10, jiné případy se liší pouze v pojmenování uzlů. *Shape* uzel pojmenovaný *nurbsPlaneShape1* obsahuje geometrii NURBS plochy a je potomkem *transform* uzlu *nurbsPlane1*. *Transform* v této situaci (není zde více nadřazených uzlů *transform*) vlastně převádí plochu definovanou v objektovém prostoru do světového. Bez tohoto uzlu by tedy nemohl *shape* uzel existovat (Maya by nevěděla

Obrázek 2.10: Reprezentace NURBS plochy pomocí *Dependecy Graph*.

kam má plochu umístit).
Jak bylo naznačeno dříve, z *transform* uzlů lze vytvářet celou hierarchii. Obr. 2.11 ukazuje příklad, ve kterém je zkonstruována krychle z jednotlivých stěn, přičemž pomocí kořenového *transform* uzlu lze transformovat krychli jako celek, kdežto *transform* uzly stěn mají vliv pouze na příslušné stěny. Výsledná transformace aplikovaná například na stěnu *topnurbsCubeShape1* je následující:

výsledná matice = topnurbsCube1 matice x nurbsCube matice

Transformace se tedy skládají násobením matic zprava, při postupu od přímého rodiče *shape* uzlu ke kořeni.

DAG cesty

Pro práci s jednotlivými DAG uzly je potřeba je nějakým způsobem adresovat. Maya zavádí pro kazdý uzel jeho DAG cestu (DAG *path*), což je vlastně popis, jak se od kořene dostat k požadovanému uzlu. Cesta ke stěně *topnurbsCubeSpape1* na obr. 2.11 je následující (symbol | odděluje jednotlivé názvy):

| nurbsCube1 | topnurbsCube1 | topnurbsCubeShape1

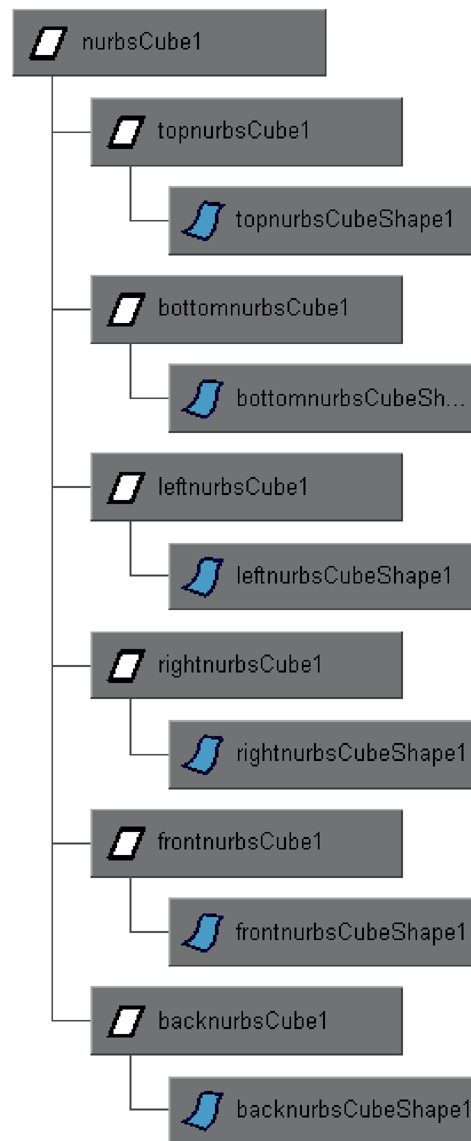
První nepojmenovaný uzle je kořenem celé scény.

Reprezentace instancí

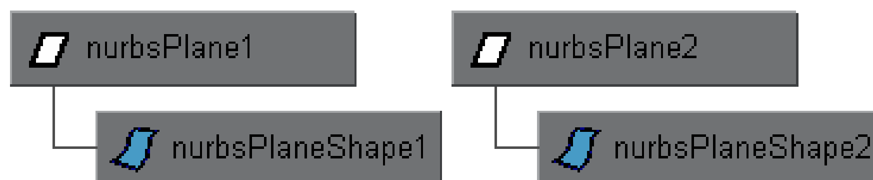
Princip instancí je jistě všeobecně známý, zde se zaměříme pouze na způsob reprezentace v Maye. Obr. 2.12 a obr. 2.13 ukazují rozdíl ve vytvoření kopie a instance. Kopie obsahuje dva rozdílné *shape* uzly a jejich *transform* uzly, kdežto instance sdílí stejný *shape* uzel a liší se pouze v *transform* uzlech. DAG cesty k oběma instancím vypadají takto:

| nurbsPlane1 | nurbsPlaneShape1
| nurbsPlane2 | nurbsPlaneShape1

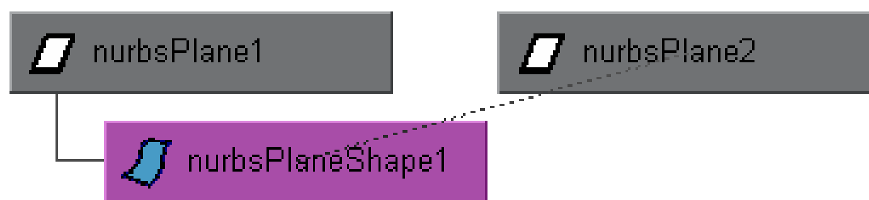
Vytváření instancí lze aplikovat rovněž na *transform* uzly.



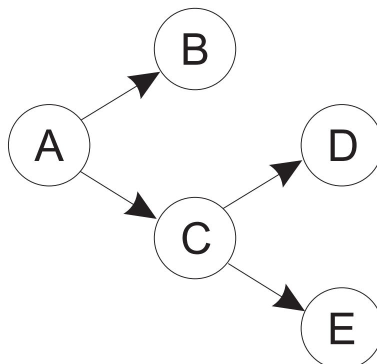
Obrázek 2.11: Příklad hierarchie *Dependecy Graph* uzlů krychle.



Obrázek 2.12: Reprezentace kopií uzlu v *Dependecy Graph*.



Obrázek 2.13: Reprezentace instancí uzlu v *Dependecy Graph*.



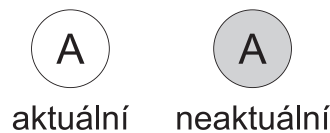
Obrázek 2.14: Ukázka výpočtu v *push–pull* modelu – počáteční konfigurace.

2.1.3 Aktualizace DG grafu

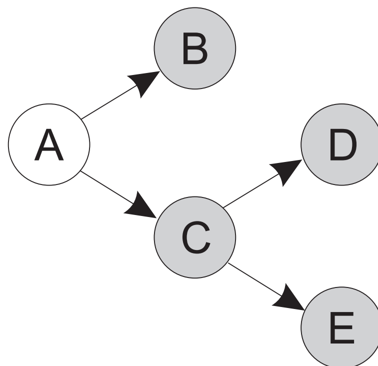
Pochopení principu aktualizace DG grafu při změně nějaké hodnoty je klíčové pro implementaci modulu. Bez těchto znalostí není možné odhadnout, kdy je nějaká hodnota platná a kdy naopak není. Aktualizace DG grafu totiž nemusí probíhat tak, jak může uživatel na první pohled předpokládat. Jak již bylo dříve řečeno, jedině Maya má kontrolu nad tokem dat a jednotlivé uzly nemají o aktuálnosti dat žádné ponětí. Návrhář nějakého uzlu tedy nemá žádnou kontrolu nad vyhodnocováním dat uložených v uzlu.

Push-pull model

Již dříve bylo vysvětleno, že DG graf pracuje na principu *data flow*. Realita je ovšem trochu jiná, ve skutečnosti se jedná o model tlačit-táhnout (*push-pull*). Toto označení vychází ze skutečnosti, že určité informace lze tzv. protlačit sítí a také je lze z ní tzv. vytáhnout. První varianta znamená propagování změny přes uzly napojené na výstup daného uzlu, druhá pak propagování požadavků přes uzly poskytující vstup danému uzlu. Tento princip jako daleko efektivnější než čistý *data flow* model.
Vezměme v úvahu situaci zobrazenou na obr. 2.14. Vidíme zde pět propojených uzlů, z nichž každý nějakým způsobem zpracovává přicházející data a výsledek posílá dál. Nyní zavedeme příznak uzlu, který indikuje neaktuálnost dat v něm. Na obr. 2.15 vidíme, jak jsou uzly značeny. Bílý uzel je aktuální a šedý neaktuální (data nejsou platná a potřebují aktualizovat). Neaktuální uzel musí nejprve vyhodnotit svá výstupní data, aktuální je může rovnou odeslat.
Prvním krokem *push-pull* modelu je propagace příznaku, při změně nějakých dat. Jakmile uzel *A* vygeneruje novou hodnotu, data nejsou ihned poslána, ale nejprve je prove-

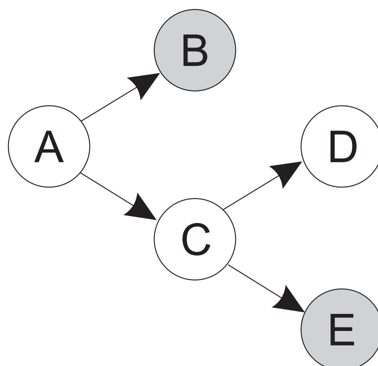


Obrázek 2.15: Ukázka výpočtu v *push–pull* modelu – možné stavy uzlu.

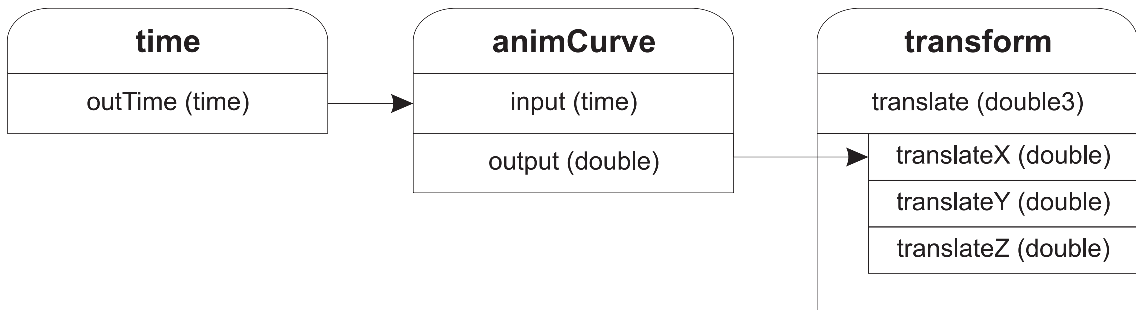


Obrázek 2.16: Ukázka výpočtu v *push–pull* modelu – průběh propagace příznaku

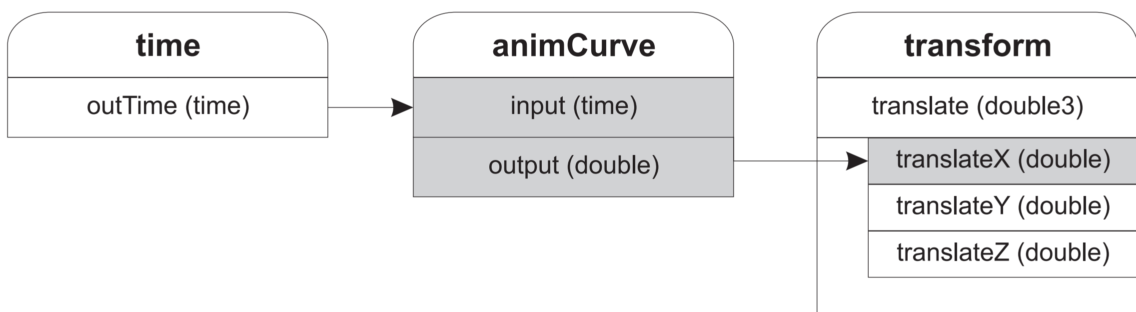
dena propagace příznaku. Uzly B a C tedy jako první nastaví své příznaky a propagují ho dále. Výsledkem je situace na obr. 2.16, kde jsou všechny uzly kromě A neaktuální a zůstanou tak, dokud nebude požadována jejich hodnota.
Nyní, pokud budeme například vyžadovat hodnotu uzlu D, se musí provést aktualizace potřebných uzlů. Uzel D není aktuální, je tedy potřeba provést nový výpočet. K tomu jsou ovšem potřeba aktuální data uzlu C. Ten ovšem také není aktuální a proto se musí přepočítat. Uzel A aktuální je, tudíž pouze předá data uzlu C a ten v zápětí po výpočtu vše předá D. Výsledek je na obr. 2.17. Za povšimnutí stojí, že uzly B a E jsou stále neaktuální, jelikož jejich data nebyla vyžadována.



Obrázek 2.17: Ukázka výpočtu v *push–pull* modelu – aktualizace uzlu D



Obrázek 2.18: Příklad skutečného propojení *Dependecy Graph* uzlů.



Obrázek 2.19: Propagace příznaku mezi *Dependecy Graph* uzly.

Aktualizace v praxi

Na základě popisu modelu lze nyní vysvětlit, jak probíhá aktualizace DG grafu. Již bylo zmíněno, že uzly jsou navzájem propojeny pomocí atributů, Maya tedy každému atributu přiřazuje příznak neaktuálnosti dat (*dirtyBit*).
Vezměme si příklad na obr. 2.18. Vidíme zde tři uzly propojené pomocí několika atributů realizující nějakou animaci v čase. Na začátku předpokládejme, že všechny atributy jsou aktuální. Nyní, pokud dojde ke změně času, změní se hodnota atributu *outTime* uzlu *time*. Tato změna má za následek propagaci zmiňovaného příznaku. Dojde tedy k nastavení příznaku atributu *input* uzlu *animCurve* a také atributu *output*. Tato propagace je provedena na základě explicitně označené závislosti mezi atributy, protože *output* je vypočítán na základě hodnoty *input*. Příznak je poté propagován k atributu *translateX* uzlu *transform*. Tento atribut již neovlivňuje žádné jiné a propagace končí. Výsledek je na obr. 2.19.
Pokud bude nyní z nějakého důvodu vyžadována hodnota atributu *translateX*, dojde k postupnému vyhodnocení podle výše pospaného *push-pull* modelu. V některých případech Maya vyžaduje od DG grafu, aby se sám vyhodnotil (např. vykreslování), přičemž se snaží být co nejefektivnější. Vyhodnocovány jsou jen ty uzly, které to opravdu potřebují. Při zmíněném překreslování scény je postupně procházena hierarchie DAG uzlů počínaje kořenem a od viditelných uzlů je vyžadována aktuální hodnota. Pokud tedy narazí na náš *transform* uzel, dojde ke zmíněnému vyhodnocení.

2.2 MEL

Pro účely této práce není jazyk MEL až tak podstatný. Jak je uvedeno dále, využívá se jen jeho velice malá část pro úpravu grafického uživatelského rozhraní (GUI). Zde je uveden jen přehled základních vlastností sloužících pro pochopení jeho existence. Podrobné informace o příkazech a syntaxi lze nalézt v [5] a v [1].
Maya Embedded Language (MEL) je velice snadno použitelný vestavěný jazyk. Pomocí jeho příkazů lze využívat téměř celou funkcionalitu Mayi. Jednoduchým příkladem je vytvoření či vybrání objektu nebo zobrazování dialogových boxů. Pomocí MELu je také možné ovládat a kompletně měnit uživatelské rozhraní, ve skutečnosti je totiž pomocí něj vytvořeno.
MEL je interpretovaný jazyk, což je jeho nesmírná výhoda. Není tedy potřeba žádný překlad, vše je za běhu vyhodnocováno a prováděno. Při práci v Maye je jakákoli akce uživatele s GUI převáděna na příkazy MELu, které jsou vyhodnoceny a realizovány. Lze tedy pomocí něj automatizovat stále se opakující úlohy, stačí jen napsat příslušný skript. Mezi nevýhody tohoto jazyka patří rychlost, což souvisí s jeho výhodou – je to jazyk interpretovaný. Skripty v něm napsané mohou být až několikrát pomalejší než adekvátní program napsaný prostřednictvím C++ API. Pro většinu úloh je ovšem plně postačující, vše záleží na konkrétních požadavcích. Jak již bylo naznačeno, MEL nemá takové možnosti v rozšiřování Mayi jako C++ API. Pouze pomocí C++ API lze totiž vytvářet nové DG uzly a mnoho dalších.

2.3 C++ API

Jak již bylo naznačeno v předchozím odstavci, pouze pomocí C++ API lze Mayu plnohodnotně rozšiřovat. Nově integrované nástroje či zcela nové vlastnosti nelze v podstatě z uživatelského hlediska odlišit od standardních. Možnosti rozšiřování jsou opravdu rozsáhlé, zde je jen nejdůležitější část relevantní této práci:

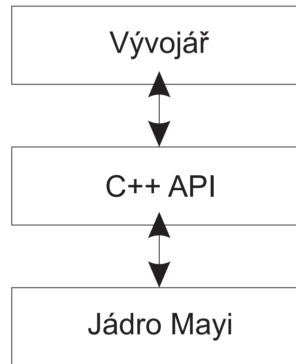
- **Příkazy** (*Commands*) – vlastní příkazy mohou být vytvořeny tak, že mají k dispozici veškerou funkcionalitu Mayi. Jelikož jsou psané v C++, lze využívat libovolné externí knihovny. Příkazy lze také volat z MELu. Chovají se identicky jako vestavěné příkazy.

- **DG uzly** (*DG nodes*) – přes C++ API lze vytvořit jak jednoduché tak velice rozsáhlé uzly, se kterými lze po integraci pracovat stejně jako se standardními uzly. Kromě základních DG lze vytvářet specializované uzly a to s již předdefinovanou základní funkcionalitou závislou na daném typu.

- **Nástroje/Kontexty** (*Tools/Contexts*) – některé operace prováděné v Maye musí být interaktivní a k tomu slouží právě tyto kontexty. Ty umožňují zpracovávat například kliknutí myši či tažení, díky čemuž je možné realizovat různé modelovací nebo animační úkony.

- **Tvary** (*Shapes*) – tyto uzly obsahují geometrická data, jedním z příkladů je NURBS plocha. Maya umožňuje vytváření zcela nových typů těchto uzlů, které se chovají stejně jako uzly standardní.

- **Data** – vytvořit lze i zcela nový typ standardně použitelných dat.



Obrázek 2.20: Struktura programového rozhraní Mayi.

Prefix	**Logická skupina**	**Příklad**
M	Třída Mayi	MObject MPoint MVector
MPx	Proxy objekt	MPxNode
MIt	Třída iterátoru	MItDag
MFn	Funkční sada	MFnMesh MFnDagNode

Tabulka 2.1: Prefixy jmen tříd poskytovaných C++ API.

Abstraktní vrstva

Ač se může na první pohled zdát, že za pomoci C++ API lze přímo přistupovat k jádru Mayi, opak je pravdou. Jak ukazuje schématické znázornění vrstev rozhraní na obr. 2.20, mezi vývojářem a jádrem Mayi je právě vrstva C++ API, která veškerý přístup zprostředkovává. Přímý přístup tudíž není vůbec možný. Přístup k jádru Mayi a operace (vytváření, získání a manipulace) s daty se provádějí pomocí metod jednotlivých tříd definovaných právě v C++ API. API tak částečně odstiňuje vývojáře od způsobu implementace jádra a také chrání před chybami (např. nechtěné smazání kritických dat).

Třídy

C++ API se skládá z řady tříd, které jsou v závislosti na jejich typu logicky rozděleny do několika hierarchických skupin . Maya nevyužívá žádné jmenné prostory C++, z toho důvodu začínají všechny její třídy prefixem *M*. Jednotlivé skupiny se odlišují prefixy, tabulka 2.1 uvádí jejich přehled.
I když se hierarchie tříd může zdát podobná běžnému objektově orientovanému modelu, jsou zde podstatné rozdíly, které je potřeba dále vysvětlit. Základní rozdíl spočívá v oddělení dat od sady funkcí (klasický přístup zahrnuje data u funkce do jedné třídy). Na jedné straně je tedy hierarchie tříd obsahující data a na druhé straně podobná hierarchie tříd obsahující ovšem funkce pro práci s daty v první skupině. Třídy spadající do druhé skupiny jsou označovány jako funkční sady (*function sets*). Pokud je tedy vyžadována nějaká operace s daty, musí se třída obsahující data přiřadit příslušné funkční sadě a poté je možné operaci provést. Následující ukázka vše ilustruje:

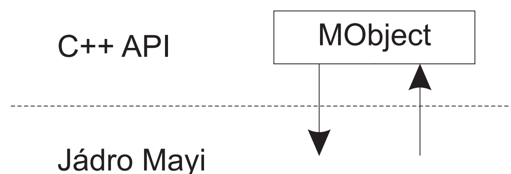
```
class ObjectFn {
    Object *data;
    void setObject(Object *obj) {data = obj;}
    virtual void doSomething() {data->doSomething();}
};

Object obj;
ObjectFn objFn;
objFn.setObject(obj);
objFn.doSomething();
```

Z příkladu je tedy zřejmé, že i když je k operaci nad daty využívána příslušná funkční sada, skutečnou práci odvede třída s daty.
Tento mechanizmus je zaveden z důvodu odstínění vývojáře od dat. Maya nikdy nedovolí přímý přístup k datům, pouze k třídě nazvané *MObject*. Tato třída ví vše o hierarchii datových tříd, operace nad nimi jsou ovšem možné jen prostřednictvím funkčních sad.

Třída MObject

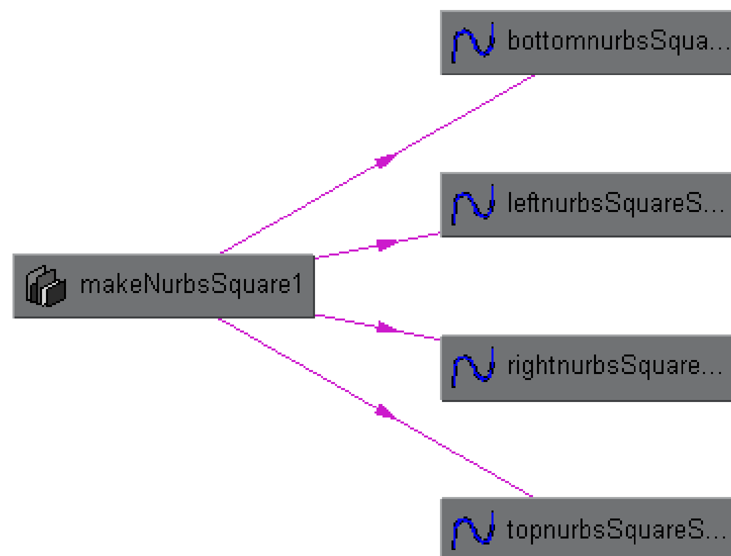
Tato třída je předkem všech datových tříd Mayi a je jako jediná viditelná z pohledu vývojáře. *MObject* tedy slouží jako přístupový bod ke všem datů. Mohlo by se tedy zdát, že tato data přímo obsahuje. Ve skutečnosti je tato třída pouze jakýmsi ovladačem k jiné třídě uvnitř jádra Mayi a obsahuje tedy jen ukazatel (*void **) na data v jádře, přičemž jen jádro může ukazatel použít. Situace je naznačená na obr. 2.21. Z výše uvedeného vlastně vyplývá, že ani pomocí třídy *MObject* nelze data smazat (smaže se pouze ukazatel), vývojář tedy nemá nikdy přímý přístup k datům.



Obrázek 2.21: Přístup k jádru Mayi pomocí rozhraní třídy *MObject*.

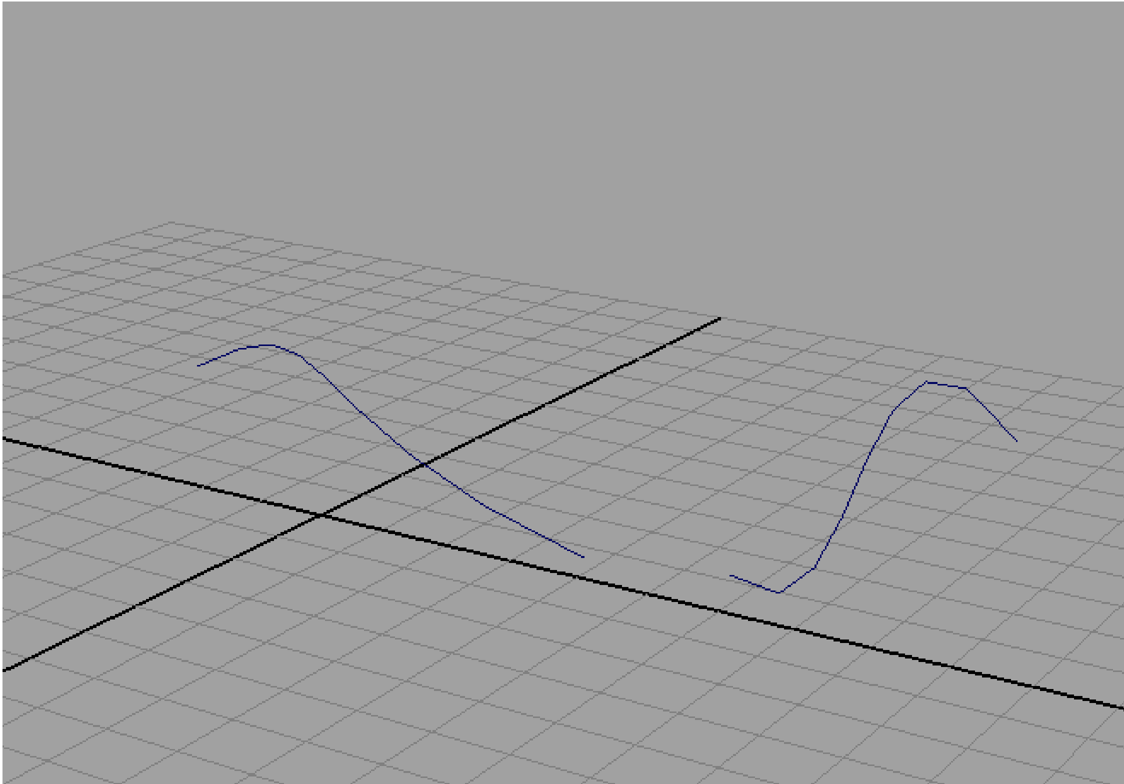
3 Popis problému a vymezení cílů

Implementace a začlenění Bézierových křivek a ploch do Mayi je již na první pohled poměrně náročný úkol. Jedná se totiž o vytvoření dvou zcela nových typů geometrie, s čímž souvisí tvorba celého balíku nástrojů, které umožní nové geometrie plnohodnotně využívat. Pro zvládnutí tohoto úkolu je nutné nastudovat principy Mayi, značnou část C++ API a také základy jazyka MEL. Důležité jsou také ukázkové příklady obsažené ve vývojovém prostředí Mayi.
Z poznatků z předchozí kapitoly vyplývá, že implementace nového typu křivek a ploch spočívá ve vytvoření dvou nových specializovaných DG uzlů, konkrétně *shape* uzlů. Problém tkví v návrhu jejich rozhraní (atributů) a také v implementaci potřebných metod. Implementace některých metod je relativně velký problém. Dokumentace API obsažená v [1] sice popisuje význam a parametry všech metod, nepopisuje však jak metodu korektně implementovat a jak přistupovat k potřebným strukturám. Kromě těchto komplikací je navíc implementace potřebných tříd značně rozsáhlá.

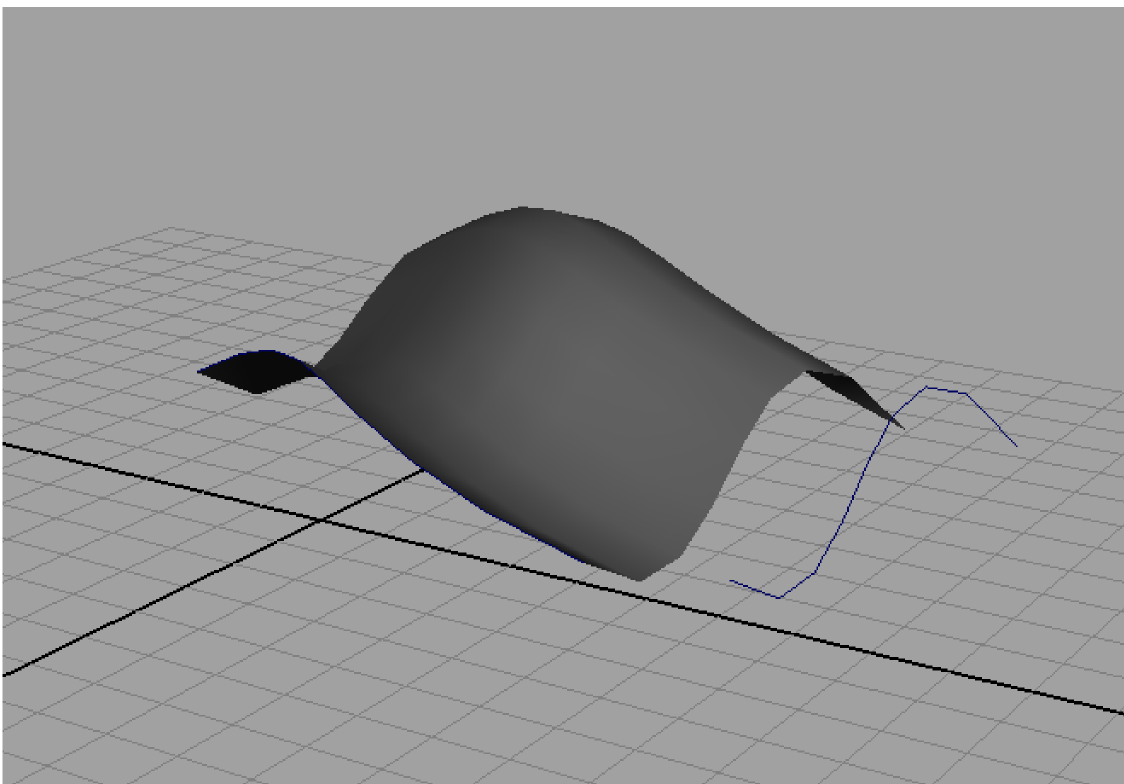


Obrázek 3.1: Princip generování geometrie.

Mimo těchto uzlů, jež jsou stěžejním problémem této práce, je potřeba implementovat některé další DG uzly. Jedním druhem jsou uzly pro generování geometrie, které dodávají předem definovanou geometrii *shape* uzlům. Jedná se například o uzel popisující geometrii čtverce pomocí čtyř křivek, který dodává geometrii jednotlivých křivek čtyřem *shape* uzlům. Na obr. 3.1 je zobrazen příslušný DG graf. Tento princip je v Maye používán např. pro vytváření základních geometrických útvarů. Druhým příkladem je uzel, který generuje geometrii plochy na základě dvou vstupních křivek. Obr. 3.2 a 3.3 ilustrují, jak se takové rozšíření provádí. Na prvním obrázku jsou dvě křivky, které se následně rozšíří na plochu. Druhý obrázek pak ukazuje výslednou plochu.
Dalším druhem DG uzlů jsou konverzní uzly, které umožní převod Bézierových křivek resp. ploch na NURBS a opačně. Tyto uzly berou geometrii *shape* uzlu napojeného na vstup a definují geometrii *shape* uzlu připojeného na výstup, jak ukazuje obr. 3.4. Základním problémem u těchto uzlů jsou samozřejmě algoritmy převodu. Pro jejich



Obrázek 3.2: Ukázka rozšíření dvou křivek na plochu – rozšiřované křivky.



Obrázek 3.3: Ukázka rozšíření dvou křivek na plochu – výsledná plocha.

Obrázek 3.4: Princip funkce konverzního uzlu.

návrh je potřeba nastudovat vztah převáděných typů křivek a ploch, což je spolu s návrhem algoritmů uvedeno v následující kapitole.
S možností napojování uzlů souvisí další problém. Atributy, přes které jsou uzly napojovány, obsahují data určitého typu. V našem případě je nutné vytvořit nová data pro reprezentaci Bézierových křivek resp. ploch, aby bylo možné je v DG grafu předávat a zpracovávat. Pokud v Maye vytvoříme nová data, musíme rovněž vytvořit iterátory, které umožní přistupovat k jednotlivým komponentám dané struktury (v tomto případě k řídícím bodům).
Aby bylo možné s nově vytvořenými uzly pracovat, je potřeba naimplementovat také sadu příkazů, které příslušné uzly vloží do DG grafu a napojí je. Realizace příkazů je poměrně snadná a dobře zdokumentovaná.
Jako poslední je potřeba vytvořit nástroje, které umožní definovat geometrický objekt tzv. ručně. Spuštěním tohoto nástroje je tedy možné definovat geometrii objektu přímo pomocí myši. Tento nástroj se nazývá kontext a obsahuje příkaz, který vytvoří *shape* uzel se zadanou geometrií.
Nově vytvořené příkazy je možné volat pomocí jazyka MEL, ovšem pro usnadnění práce je vhodné rozšířit grafické uživatelské rozhraní o ikony příp. položky menu dovolující spouštět příkazy pohodlnějším způsobem.

3.1 Požadavky na výsledné řešení

3.1.1 Geometrické objekty

Určitým vodítkem při návrhu jsou vlastnosti NURBS křivek a ploch příp. jejich nástrojů, jelikož podobnost obou typů je poměrně veliká (Bézierovy křivky resp. plochy jsou v podstatě podmnožinou NURBS). Co nejpodobnější chování obou typů křivek resp. ploch je tedy žádoucí. Obecné vlastnosti NURBS křivek a ploch vzhledem k Maye jsou uvedeny v knize [6]. Další vlastnosti, jako je rozhraní uzlu nebo funkce jednotlivých nástrojů, lze vypozorovat přímo při práci s programem. Podstatné je zejména ovládání objektů prostřednictvím standardních nástrojů Mayi, vykreslování v jednotlivých režimech zobrazování, již zmiňovaná struktura rozhraní nebo možnost modifikací objektů. Důležitou vlastností je také možnost aplikování standardních deformátorů. Nyní probereme některé vlastnosti objektů trochu podrobněji.

Ovládání standardními nástroji

Mezi standardní nástroje Mayi patří například posun, rotace či změna měřítka. Aplikace těchto nástrojů na objekt jako celek není problémem vlastního *shape* uzlu, nýbrž nadřazeného *transform* uzlu, a není tedy potřeba ho řešit. To ovšem neplatí u komponent dané geometrie. Jedním z požadavků tedy je aplikovatelnost těchto nástrojů na jednotlivé řídící body.

Vykreslování

Co se týče vykreslování, je vhodné umožnit explicitní povolení či zakázání zobrazení jednotlivých částí objektů (vlastní geometrie, řídících bodů či obalu – *hull*). Kromě toho je také žádoucí nastavitelná kvalita vykreslování v jednotlivých režimech zobrazení (jemnost rozkladu křivek, hustota drátového modelu plochy atd.). Samozřejmostí je zvýraznění vybraných objektů či jejich komponent.

Struktura rozhraní

Vytváření předdefinovaných geometrických objektů je standardně realizováno pomocí dvou uzlů, jak je vidět například na obr. 3.1. To je umožněno rozhraním *shape* uzlu, které dovoluje přijmout geometrii od jiného DG uzlu. Rozhraní musí být také schopno předávat geometrii dalším uzlů, například kvůli stínování či prostě dalšímu zpracování. Tento princip je běžně využíván a proto je nutné implementovat rozhraní tak, aby dovolovalo potřebné struktury v DG grafu konstruovat.

Modifikace objektu

Po vytvoření objektu je často potřeba ho určitým způsobem modifikovat. Zejména je tím myšleno odebírání řídících bodů geometrie.

Použití deformátorů

Pro možnost použití deformátorů je potřeba implementovat speciální rozhraní, což přináší řadu problémů s návrhem a implementací *shape* uzlu. Problematika návrhu s ohledem na použití deformátorů je relativně složitá a velmi málo zdokumentovaná.

3.1.2 Ostatní DG uzly

Tyto uzly (jedná se o zmíněné generátory geometrie a konverzní uzly) by měly mít vlastnosti, které od nich běžný uživatel očekává. Konkrétně u uzlů definujících geometrii by mělo být možné měnit pomocí parametrů vlastnosti výsledné geometrie, tedy např. rozměry plochy či stupeň křivek. Tato změna by měla být možná kdykoli po vytvoření daného uzlu. Je tedy zřejmé, že je potřeba vhodně navrhnou jeho rozhraní. Možnost výstupu geometrie tohoto uzlu je samozřejmostí, jinak by ostatně nemohl dodávat geometrii dalším uzlům.
Co se týče rozhraní konverzních uzlů, to musí být schopné pouze přijímat geometrii pro převod a vysílat výslednou geometrii následujícím uzlům.

3.1.3 Geometrická data

Požadavky na data reprezentující Bézierovy křivky a plochy jsou vcelku jasné. Data musí obsahovat veškeré informace potřebné pro popis geometrie a její následné zpracování. U křivky se jedná zejména o řídící body a její stupeň, v případě plochy pak o řídící body a stupně v jednotlivých parametrických směrech U a V.

3.1.4 Příkazy a nástroje

Příkazy vytvářející a propojující požadované uzly nemusí mít z hlediska uživatele žádné zvláštní vlastnosti. Nezbytné není ani definování argumentů, jelikož příslušné hodnoty lze nastavit explicitně pomocí příkazů MELu nebo ručně z GUI. Samozřejmým požadavkem je intuitivní a snadné použití. Reprezentace příkazu ikonou nebo položkou v menu je pro usnadnění práce tedy žádoucí.
Nové nástroje by měly být jednoduché a jejich chování by se mělo co nejvíce přibližovat odpovídajícím nástrojům pro NURBS. Uživatel je tak bude schopen okamžitě používat. Možnost spuštění nástroje ikonou je opět žádoucí.

3.2 Přehled cílů

Na závěr této kapitoly je přehledně uveden souhrn všech hlavních cílů této práce.

Shape uzly

- Bézierova křivka
- Bézierova plocha

Geometrická data

- reprezentace Bézierovy křivky
- reprezentace Bézierovy plochy

DG uzly definující geometrii

- čtverec složený ze čtyř Bézierových křivek
- Bézierova plocha ve tvaru čtverce
- krychle složená ze šesti Bézierových ploch
- Bézierova plocha vzniklá z rozšíření dvou křivek

Převodní DG uzly

- Bézierova křivka na NURBS křivku
- NURBS křivka na Bézierovu křivku
- Bézierova plocha na NURBS plochu
- NURBS plocha na Bézierovu plochu

Kontexty

- manuální vytvoření Bézierovy křivky definováním řídících bodů

Příkazy

- vytvoření Bézierovy křivky pomocí bodů předaných jako argumenty (je součástí výše zmíněného kontextu, ale lze volat i samostatně z příkazové řádky)
- vytvoření čtverce složeného ze čtyř Bézierových křivek
- vytvoření Bézierovy plochy ve tvaru čtverce
- vytvoření krychle složené ze šesti Bézierových ploch
- rozšíření dvou křivek na Bézierovu plochu
- převod všech vybraných Bézierových křivek na NURBS křivky
- převod všech vybraných NURBS křivek na Bézierovy křivky
- převod všech vybraných Bézierových ploch na NURBS plochy
- převod všech vybraných NURBS ploch na Bézierovy plochy
- převod všech Bézierových ploch ve scéně na NURBS plochy

Menu

- vytvoření nového menu obsahujícího všechny příkazy

Přihrádky

- vytvoření nových přihrádek (*shelfs*) obsahujících všechny příkazy, včetně nakreslení potřebných ikonek

4 Analýza a návrh řešení

Z předchozí kapitoly je zřejmé, jaké nové prvky je potřeba navrhnout a implementovat, tedy co všechno musí obsahovat programový modul. Již v úvodu této práce bylo uvedeno, že Mayu lze rozšířit prostřednictvím dvou rozhraní – MELu a C++ API. Před vlastním návrhem je potřeba nejprve rozhodnout, které rozhraní (příp. jaká kombinace) je pro úkol nejvhodnější.
Zmíněna byla také možnost reprezentovat data ve dvou variantách. Pro výběr jedné z nich je nutné jednotlivé možnosti analyzovat. Další součástí jsou také převodní algoritmy, k jejichž návrhu je potřeba pochopit vztah Bézierových křivek resp. ploch a NURBS.

4.1 Volba rozhraní

4.1.1 C++ API

Již ze základních principů uvedených v kapitole 2 lze vcelku jednoznačně usoudit, že valná většina prvků modulu musí být implementována pomocí C++ API, protože MEL není schopen vytvářet zcela nové uzly či kontexty. Prostřednictvím C++ API bude tedy realizováno:

Shape uzly

- Bézierova křivka
- Bézierova plocha

Geometrická data

- reprezentace Bézierovy křivky
- reprezentace Bézierovy plochy

DG uzly definující geometrii

- čtverec složený ze čtyř Bézierových křivek
- Bézierova plocha ve tvaru čtverce
- krychle složená ze šesti Bézierových ploch
- Bézierova plocha vzniklá z rozšíření dvou křivek

Převodní DG uzly

- Bézierova křivka na NURBS křivku
- NURBS křivka na Bézierovu křivku
- Bézierova plocha na NURBS plochu
- NURBS plocha na Bézierovu plochu

Kontexty

- manuální vytvoření Bézierovy křivky definováním řídících bodů

Příkazy, které modifikují DG graf prostřednictvím přidávání nových uzlů a jejich napojováním, lze v zásadě realizovat i pomocí MELu. Ovšem jen prostřednictvím C++ API je možné získat větší kontrolu nad těmito modifikacemi. Dalším kritériem hovořícím pro C++ API je vyšší rychlost výsledného příkazu. Dále bude tedy pomocí tohoto rozhraní realizováno následující:

Příkazy

- vytvoření čtverce složeného ze čtyř Bézierových křivek
- vytvoření Bézierovy plochy ve tvaru čtverce
- vytvoření krychle složené ze šesti Bézierových ploch
- rozšíření dvou křivek na Bézierovu plochu
- převod všech vybraných Bézierových křivek na NURBS křivky
- převod všech vybraných NURBS křivek na Bézierovy křivky
- převod všech vybraných Bézierových ploch na NURBS plochy
- převod všech vybraných NURBS ploch na Bézierovy plochy

4.1.2 Jazyk MEL

Jazyk MEL oproti C++ API disponuje rozsáhlejšími možnostmi modifikace grafického uživatelského rozhraní, čehož je v této práci využito pro přidání následujících prvků:

Menu

- vytvoření nového menu obsahujícího všechny příkazy

Přihrádky

- vytvoření nových přihrádek (*shelfs*) obsahujících všechny příkazy

Následně uvedený příkaz není nutné implementovat pomocí C++ API. S výhodou lze využít příkaz pro převod všech vybraných Bézierových křivek na NURBS křivky a to tak, že před jeho provedením je vybrána celá scéna. Pro tento úkol je MEL vhodný, protože umožňuje jeho provedení pomocí pouhých dvou příkazů:

Příkazy

- převod všech Bézierových ploch ve scéně na NURBS plochy

4.2 Vztah Bézierových křivek a ploch k NURBS

Nebudeme se zde zabývat podrobnými definicemi a vlastnostmi obou typů křivek resp. ploch, to může čtenář nalézt např. v knize [7]. Pro tuto práci jsou spíše důležité závěry, které lze z těchto informací vyvodit.
NURBS (*Non Uniform Rational Basis Spline*) křivky jsou parametrické křivky vzniklé zobecněním neracionálních křivek označovaných jako *B-splines* a racionálních i neracionálních Bézierových křivek. Z toho vyplývá, že Bézierovy křivky jsou jakousi podmnožinou NURBS křivek. Je tedy zřejmé, že pomocí NURBS lze popsat i adekvátní Bézierovu křivku. Například NURBS křivka definovaná promocí množiny řídících bodů P a uzlového vektoru U takto:

P = (0.5, 0.0, 1.0), (0.0, 0.0, 0.0), (-6.2, 3.0, 8.2), (-10.0, 0.0, 4.3)
U = (0, 0, 0, 0, 1, 1, 1, 1)

je vlastně Bézierova křivka definovaná pouze množinou řídících bodů P:

P = (0.5, 0.0, 1.0), (0.0, 0.0, 0.0), (-6.2, 3.0, 8.2), (-10.0, 0.0, 4.3)

Odlišnost je tedy pouze v uzlovém vektoru, který Bézierova křivka nemá. Pomocí NURBS křivky lze tedy definovat libovolnou Bézierovu křivku, obráceně to samozřejmě neplatí (existuje ovšem algoritmus převodu libovolné NURBS křivky na několik Bézierových křivek, viz převodní algoritmy dále).
Výše uvedené závěry platí analogicky i pro vztah NURBS a Bézierových ploch, jelikož plocha je pouhým tenzorovým produktem.

4.3 Volba způsobu reprezentace dat

Jednou z klíčových otázek návrhu tohoto modulu je způsob realizace dat pro reprezentaci Bézierových křivek resp. ploch. Jak již bylo zmíněno, jednotlivé uzly mají svá rozhraní v podobě definovaných atributů, přičemž atribut vlastně jen obsahuje data daného typu. Tato data si pak jednotlivé uzly mezi sebou vyměňují. V úvahu přicházejí dvě varianty realizace nových dat, přičemž obě mají svá pro i proti. První možnost spočívá ve využití stávajících dat pro reprezentaci NURBS, druhá potom ve vytvoření zcela nových dat. Rozbor obou variant je uveden dále, vysvětlen je zde také důvod volby druhé varianty.

4.3.1 Využití dat NURBS

Možnost využití stávající datové reprezentace NURBS pro Bézierovy křivky či plochy vyplývá z výše uvedených vztahů. Tato varianta s sebou nese mnoho pozitiv, ale i některá negativa.
Pokud by se této reprezentace využilo, bylo by možné s daty pracovat pomocí stávajících funkčních sad Mayi pro práci s NURBS, což by přineslo jistě mnoho zjednodušení při implementaci. Dalším kladem je, že data jsou již připravena k použití. To znamená, že by nebylo potřeba implementovat žádné třídy jako v případě zcela nových dat. Odpadla by tudíž implementace nového typu geometrie, iterátoru a samozřejmě vlastních dat.
Nevýhodou ovšem je, že tato data nelze měnit, zejména co se týče vnitřní struktury. Pro reprezentaci Bézierových křivek a ploch je struktura pro NURBS také zbytečně složitá.

Dalším faktorem hovořícím proti použití struktur pro NURBS je způsob, jakým jsou jednotlivé geometrické objekty, již v Maye existující, realizovány. Každý typ geometrického objektu má totiž i svůj vlastní typ dat, což je jistě čisté a přehledné řešení.

4.3.2 Vytvoření nových dat

Tato varianta s sebou přináší v první řadě pracnější návrh. Jelikož se bude jednat o zcela nová data, nebude možné přímé použití žádné funkční sady pro NURBS, tudíž bude potřeba veškeré metody naimplementovat.
Velkou výhodou ovšem je, že si lze libovolně nadefinovat strukturu dat a v podstatě libovolně ji podle potřeb rozšiřovat či měnit. Tento způsob realizace také zachová již zmíněnou jednotu typů geometrií a jejich dat.
Více důležitějších argumentů tedy hovoří pro návrh zcela nového typu dat uzpůsobeného pro Bézierový křivky resp. plochy. To ovšem vede k pracnější realizaci modulu. Kromě vlastní geometrie a dat je nutné implementovat i již zmíněné iterátory.

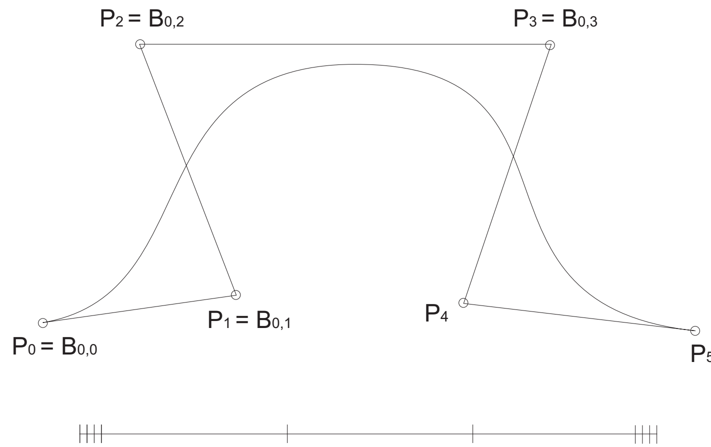
4.4 Převodní algoritmy

Ještě před návrhem vlastního modulu se budeme blíže věnovat převodním algoritmům, které budou použity v konverzních uzlech. Jedná se o již zmíněnou čtveřici algoritmů, převod Bézier – NURBS pro křivky i plochy a NURBS – Bézier také pro křivky i plochy. První varianta převodu je triviální a lze ji odvodit na základě vlastností křivek. Druhá varianta již tak triviální není, vycházím z algoritmů uvedených v [7] s několika úpravami s ohledem na vlastnosti NURBS v Maye.
Před uvedením algoritmů je důležité zmínit jednu odlišnou vlastnost NURBS křivek v Maye. Podle definic uvedených ve zmíněné knize [7], obsahuje uzlový vektor (počet řídících bodů + stupeň + 1) uzlů. Maya ovšem vyžaduje k definici NURBS o 2 uzly méně, tedy jen (počet řídících bodů + stupeň - 1) uzlů. Proč tomu tak je, není nikde v dokumentaci uvedeno. Důležité je, že při přechodu od uzlového vektoru podle definice k uzlovému vektoru v Maye stačí první a poslední uzel odstranit. Při opačném přechodu je pak potřeba první a poslední uzel duplikovat. Algoritmy zde uvedené vycházejí z klasické definice.
Nyní již k vlastním algoritmům, začneme nejprve triviální variantou převodu.

4.4.1 Bézier – NURBS

Tento převod je opravdu jednoduchý a byl popsán v podstatě již při vysvětlování vztahu Bézierových a NURBS křivek. Pro úplnost je zde však uveden.
Mějme libovolnou Bézierovu křivku C_B definovanou N+1 řídícími body P_0, ..., P_N, kde N je stupeň křivky. NURBS křivku C_N stejného stupně N získáme tak, že zkopírujeme řídící body P_0, ...,P_N a vytvoříme uzlový vektor U odpovídající uniformnímu dělení, tzn. U je tvořeno posloupností N+1 nul a N+1 jedniček (U = (0, ...,0, 1, ..., 1)). Algoritmus zapsaný v pseudokódu tedy vypadá následovně:

```
for ( všechny body křivky Cb )
    vytvoř kopii pro Cn
vytvoř U z N+1 nul a N+1 jedniček
stupeň Cn je N
```



Obrázek 4.1: Příklad rozkladu NURBS křivky – původní křivka.

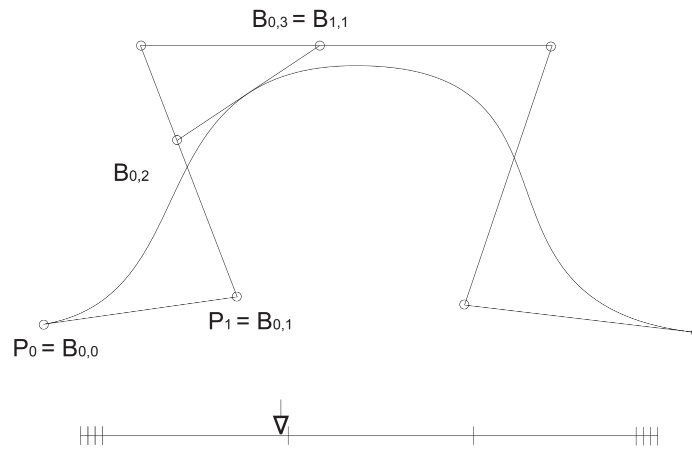
Pro plochy platí algoritmus analogicky.

```
for ( všechny body plochy Sb )
    vytvoř kopii pro Sn
vytvoř U z Nu+1 nul a Nu+1 jedniček
vytvoř V z Nv+1 nul a Nv+1 jedniček
stupeň Sn ve směru U je Nu
stupeň Sn ve směru V je Nv
```

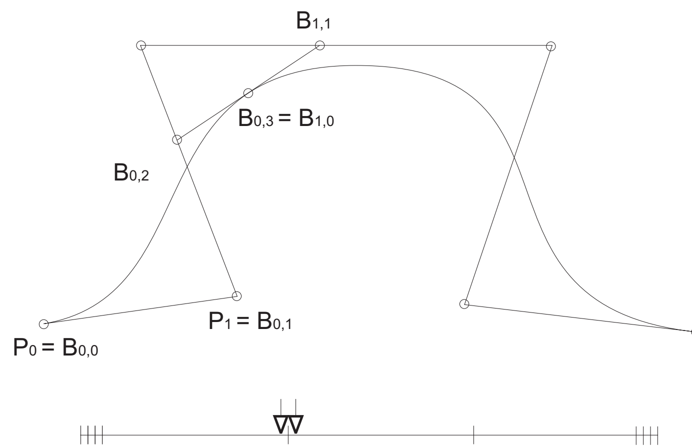
S_B značí Bézierovu plochu definovanou $(N_U+1).(N_V+1)$ řídícími body, kde N_U je stupeň plochy ve směru U, N_V pak ve směru V. S_N značí NURBS plochu, U uzlový vektor směru U a V uzlový vektor směru V.

4.4.2 NURBS – Bézier

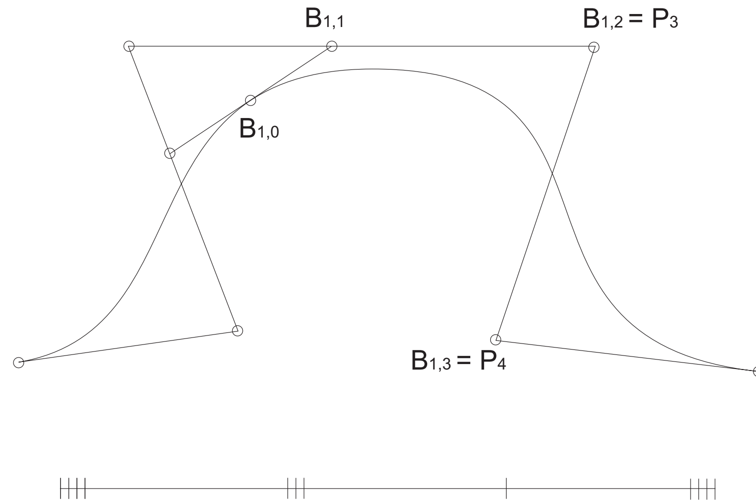
Převod NURBS křivky na Bézierovu je v podstatě rozklad křivky na jednotlivé části. Princip tohoto algoritmu je založen na násobném vkládání vnitřních uzlů do uzlového vektoru. Cílem je dosáhnout stejné násobnosti všech uzlů v uzlovém vektoru odpovídající stupni křivky. Vkládání uzlů do uzlového vektoru s sebou samozřejmě přináší vznik nových řídících bodů, díky čemuž je možné získat požadovaný rozklad na Bézierovy křivky. Počet těchto křivek odpovídá počtu segmentů (*spans*) NURBS křivky a stupeň jednotlivých křivek je roven stupni NURBS křivky.
Nyní zavedeme potřebné značení. Mějme NURBS křivku C_N stupně N, definovanou P řídícími body a uzlovým vektorem U obsahujícím P+N+1 uzlů. NURBS křivka má tedy P-N segmentů. Označení $B_{j,k}$ znamená k-tý řídící bod Bézierovy křivky j-tého segmentu (řídících bodů jednoho segmentu je N+1). Příklad rozkladu ukazují obr. 4.1 – 4.4. Zobrazená NURBS křivka je složena ze tří segmentů, pod ní je naznačena struktura uzlového vektoru resp. četnost jednotlivých uzlů a jejich přidávání. Následuje vlastní algoritmus:



Obrázek 4.2: Příklad rozkladu NURBS křivky – vložení prvního uzlu.



Obrázek 4.3: Příklad rozkladu NURBS křivky – vložení druhého uzlu.



Obrázek 4.4: Příklad rozkladu NURBS křivky – příprava na druhý segment.

```
for ( prvních N+1 bodů křivky Cn )
    ulož bod do prvního segmentu B
počínaje prvním vnitřním uzlem
počínaje prvním segmentem S
while ( nebyly probrány všechny uzly ) {
    spočítej četnost uzlu
    if ( četnost uzlu není N ) {
        for ( počet chybějících uzlů )
            vypočítej příslušný koeficient alfa pro interpolaci
        for ( počet chybějících uzlů ) {
            for ( počet přidávaných bodů ) {
                získej příslušné alfa
                Bs,i = alfa * Bs,i + (1 - alfa) * Bs,i-1
            }
            if ( ještě zbývají uzly )
                ulož potřebný řídící bod do dalšího segmentu
        }
    }
    posun na další segment S
    if ( ještě zbývají uzly )
        inicializuj řídící body dalšího segmentu
}
```

Algoritmus pro převod NURBS plochy na jednotlivé Bézierovy plochy je analogický. V podstatě se aplikuje výše uvedený algoritmus postupně pro jednotlivé směry U a V. Je tedy zřejmé, že po průběhu ve směru U získáme Bézierovy pásy, které se po aplikaci ve směru V rozloží na požadované Bézierovy plochy.

4.5 Návrh modulu

Nyní tedy k vlastnímu návrhu modulu. Nejprve se budeme věnovat jeho základní struktuře. Následně pak provedeme návrh jednotlivých částí modulu. Při návrhu je vhodné se nechat inspirovat vlastnostmi stávajících uzlů a příkazů v Maye. Věnovat se budeme pouze návrhu důležitých částí, některé jsou totiž čistě implementační problém.

4.5.1 Základní struktura modulu

Vzhledem k vzájemné provázanosti jednotlivých části je vše zahrnuto do jediného modulu, což znamená, že jeho načtením je získána kompletní funkčnost všech příkazů, nástrojů atd. Jeho struktura pak v podstatě odpovídá dělení na jednotlivé cíle práce:

Obecné:

- inicializace a zrušení modulu v Maye, včetně modifikace GUI
- MEL skripty pro modifikaci GUI (externí soubory)

Bézierovy křivky:

- data křivek
- geometrie křivek
- iterátor geometrie křivek
- *shape* uzel křivek
- vykreslování a výběr křivek
- DG uzel definující geometrii čtverce ze čtyř křivek
- příkaz vkládající čtverec do scény
- DG uzel definující geometrii čtverce ze čtyř křivek
- příkaz vkládající čtverec do scény
- kontext pro manuální vytváření křivky
- DG uzel pro převod NURBS – Bézier
- příkaz pro převod všech vybraných NURBS křivek na Bézierovy křivky
- DG uzel pro převod Bézier – NURBS
- příkaz pro převod všech vybraných Bézierových křivek na NURBS křivky

Bézierovy plochy:

- data ploch
- geometrie ploch
- iterátor geometrie ploch
- *shape* uzel ploch
- vykreslování a výběr ploch
- DG uzel definující geometrii plochy ve tvaru čtverce
- příkaz vkládající plochu do scény
- DG uzel definující geometrii krychle ze šesti ploch
- příkaz vkládající krychli do scény
- DG uzel pro vytvoření plochy rozšířením dvou křivek
- příkaz pro rozšíření dvou vybraných křivek na plochu
- DG uzel pro převod NURBS – Bézier
- příkaz pro převod všech vybraných NURBS ploch na Bézierovy plochy
- DG uzel pro převod Bézier – NURBS
- příkaz pro převod všech vybraných Bézierových ploch na NURBS plochy

4.5.2 Návrh DG uzlů

Návrh těchto uzlů, zejména pak *shape* uzlů, je v této práci nejdůležitější cíl. Hlavní problém je návrh jejich rozhraní. Uvedena jsou i jména uzlů, která jsou registrována do Mayi.

Shape uzel křivek

- *bezierCurveShape*

Uzel musí být schopen uchovávat svoji geometrii, jelikož bez ní by se nemohla vykreslit geometrie, ani by uzel nemohl posílat svá geometrická data jiným uzlům. Atribut k tomu určeny je standardně nazýván *cached* a jeho datový typ odpovídá ukládané geometrii, v našem případě tedy datům Bézierových křivek. Tento atribut, ještě spolu s předdefinovaným atributem *mControlPoints* (každý nově definovaný *shape* uzel obsahuje základní atributy), dále dovoluje aplikování tzv. offsetů (*tweaks*), což je mechanizmus používaný zejména deformátory. Představme si *shape* uzel v DG grafu obsahující geometrii nějaké Bézierovy křivky, na který aplikujeme nějaký deformátor. Deformátor v podstatě pracuje tak, že nejprve vytvoří kopii *shape* uzlu (originál se stane neviditelným), od originálu přejme geometrii, zdeformuje ji a kopii předá pouze offsety, které lze získat právě z atributu *mControPoints*. Výslednou geometrii tedy zkopírovaný

uzel získá, pokud sečte obsah dat atributu *cached* a *mControlPoints* (této operaci se říká aplikace offsetů). Aplikace offsetů je problémem zejména z hlediska implementace, čímž se zabývá následující kapitola.
Aby byl uzel schopen získávat geometrii od jiných uzlů, jak to umožňují existující uzly Mayi, musí obsahovat atribut datového typu odpovídajícímu geometrii Bézierových křivek, který umožní napojení jiného uzlu definujícího geometrii. Atribut nese jméno *create*.
Stejně jako je potřeba geometrii přijímat, je potřeba ji i předávat dál. Možnosti předání jsou ovšem dvě, jelikož objekt existuje ve dvou souřadnicových systémech – v lokálním neboli objektovém a ve světovém. Z tohoto důvodu je potřeba vytvořit dva atributy, opět datových typů odpovídajících geometrii Bézierových křivek, a to *local* a *worldSpace*. Druhý atribut ovšem nemůže být jednoduchý a to z důvodu možné existence instancí (každá instance může být ve světových souřadnicích umístěna jinam). Atribut *worldSpace* je tedy polem a počet jeho prvků je dán počtem instancí.
Dobrým pomocným atributem, který poskytuje dalším uzlům v případě potřeby informace, je atribut *degree* datového typu *short*. Jak již název napovídá, jedná se o atribut obsahující stupeň dané křivky.
Jedním z požadavků na nové objekty byla možnost explicitního povolení/zakázání vykreslování komponent křivky či křivky jako takové. Trojice atributů *dispCV*, *dispHull* a *dispGeometry* to umožňuje. Všechny jsou typu *boolean*.
Posledním atributem datového typu *short* je *curvePrecision*, který řídí jemnost rozkladu křivky při vykreslování.
Pokud geometrický objekt obsahuje komponenty, Maya umožňuje také jejich mazání. U křivek je princip odstraňování jednotlivých řídících bodů vcelku jasný, libovolný vybraný bod bude prostě z geometrie vyjmut.

Shape uzel ploch

- *bezierSurfaceShape*

Rozhraní tohoto uzlu je velice podobné předchozímu. Atributy *cached*, *create*, *local* a *worldSpace* jsou jen jiného datového typu odpovídajícího geometrii Bézierových ploch. Atribut obsahující stupně plochy v jednotlivých směrech se jmenuje *degreeUV* a jedná se o složený atribut ze dvou jednoduchých atributů *degreeU* a *degreeV* (oba typu *short*). *DispCV*, *dispHull* a *dispGeometry* jsou identické.
Podstatnější rozdíl je v atributech umožňujících nastavení kvality vykreslování. *CurvePrecisionU* a *curvePrecisionV* definují jemnost rozkladu křivek v drátovém modelu, *shadedPrecisionU* a *shadedPrecisionV* pak jemnost rozkladu vykreslovaných ploch. A konečně *DivisionsU* a *divisionsV* určují hustotu drátového modelu. Všechny tyto atributy jsou typu *short*.
Odstraňování komponent u ploch již není tak jednoznačné jako u křivek. Byl zvolen způsob, který dovoluje mazat řídící body po sloupcích, řadách nebo jejich kombinaci. Před vlastním odstraněním je nutné vybrat všechny body dané řady či sloupce. V opačném případě nebude odstraněno nic.

DG uzel definující geometrii čtverce

- *makeBezierSquare*

Pro funkci tohoto uzlu je zásadní, aby byl schopen dodávat potřebnou geometrii připojeným *shape* uzlům (konkrétně čtyřem). Z tohoto důvodu musí obsahovat čtyři atributy, z nichž každý obsahuje data jedné z obvodových křivek. Atributy jsou pojmenovány *outputCurve1*, *outputCurve2*, *outputCurve3* a *outputCurve4*. Datový typ atributů odpovídá geometrii Bézierových křivek.
Pokud chceme umožnit změnu generované geometrie, je potřeba vytvořit další atributy. *SideLength1* a *sideLength2* typu *double* dovolují definovat délku obou rozměrů čtverce (výsledný objekt může být tedy i obdélník). Atribut *degree* definuje stupeň obvodových křivek, je typu *short*. Aby bylo možné objekt umístit a orientovat v prostoru, zavedeme také atributy *center* a *normal* typu *3double*.

DG uzel definující geometrii plochy

- *makeBezierPlane*

Geometrie definovaná tímto uzlem je jen jedna, tudíž postačí atribut *outputSurface* datového typu pro Bézierovy plochy. Šířku plochy a poměr délky ku šířce lze nastavovat pomocí atributů *width* a *lengthRatio* typu *double*. *DegreeU* a *degreeV* typu *short* dovolují nastavit stupeň plochy v jednotlivých směrech. Pro umístění a orientaci plochy je zde opět dvojice atributů *center* a *normal*.

DG uzel definující geometrii krychle

- *makeBezierCube*

Pro definici geometrie krychle je potřeba šesti Bézierových ploch, tomu odpovídá šestice atributů *outputSurface1 – 6* příslušného datového typu. *Width*, *lengthRation* a *heightRatio* pak definují šířku, poměr délka/šířka a poměr výška/šířka dané krychle. Všechny tři atributy jsou opět typu *double*. Atribut *degree* typu *short* umožňuje nastavit stupeň všech šesti ploch a to jednotně pro směr U i V. *Center* a *normal* jsou samozřejmostí.

DG uzel pro rozšíření dvou křivek na plochu

- *extendToBezierSurface*

Tomuto uzlu postačuje velice jednoduché rozhraní. Pro načtení geometrie dvou Bézierových křivek má atributy *inputCurve1* a *inputCurve2* příslušného typu. Výsledná geometrie Bézierovy plochy je pak v atributu *outputSurface*.

Převodní DG uzly

- *bezierCurveToNurbsConvertor*
- *bezierCurveFromNurbsConvertor*
- *bezierSurfaceToNurbsConvertor*
- *bezierSurfaceFromNurbsConvertor*

Všechny čtyři převodní uzly mají v podstatě identická rozhraní. Jeden atribut slouží pro vstup převáděné geometrie, druhý pak pro vstup geometrie převedené. Atributy se pouze liší v pojmenování a datových typech. U křivek jsou atributy nazvány *inputCurve* resp. *outputCurve*, u ploch pak *inputSurface* resp. *outputSurface*. Jak ovšem vyplývá z algoritmu pro převod NURBS–Bézier, výsledný počet křivek může být větší než jedna. Z toho důvodu je u příslušných převodních uzlů nutné, aby výstupní atributy outputCurve resp. outputSurface měly strukturu pole.

Závěrem je v tabulce 4.1 uveden přehled všech atributů jednotlivých uzlů, včetně jejich typů. Datovým typem křivka resp. plocha je míněna Bézierova křivka resp. plocha.

4.5.3 Návrh dat

Jména dat:

- *bezierCurve*
- *bezierSurface*

Vlastní geometrie křivek a ploch obsažená v datech bude v zásadě velice jednoduchá. V tuto chvíli totiž nebude potřeba ukládat žádné jiné informace než jen vlastní řídící body křivky resp. plochy a jím odpovídající stupně.
Geometrie křivky bude tedy uložena v podobě jednorozměrného pole bodů o čtyřech souřadnicích x, y, z, w. C++ API obsahuje třídy, které umožňují reprezentaci téměř libovolných dat připadajících v úvahu. V případě pole bodů lze využít strukturu, která obsahuje i informaci o počtu řídících bodů. Stupeň křivky tedy není třeba ukládat ve zvláštní proměnné.
Co se týče plochy, její řídící body budou uloženy také v podobě jednorozměrného pole bodů o čtyřech souřadnicích x, y, z, w. Takto jsou zřejmě uloženy i řídící body NURBS ploch. Řídící body NURBS plochy se zadávají a získávají v podobě jednorozměrného pole, kde jsou body uloženy po sloupcích. Představme si například tuto NURBS plochu zadanou maticí bodů:

P_{00} P_{01} P_{02} P_{03}
P_{10} P_{11} P_{12} P_{13}
P_{20} P_{21} P_{22} P_{23}
P_{30} P_{31} P_{32} P_{33}

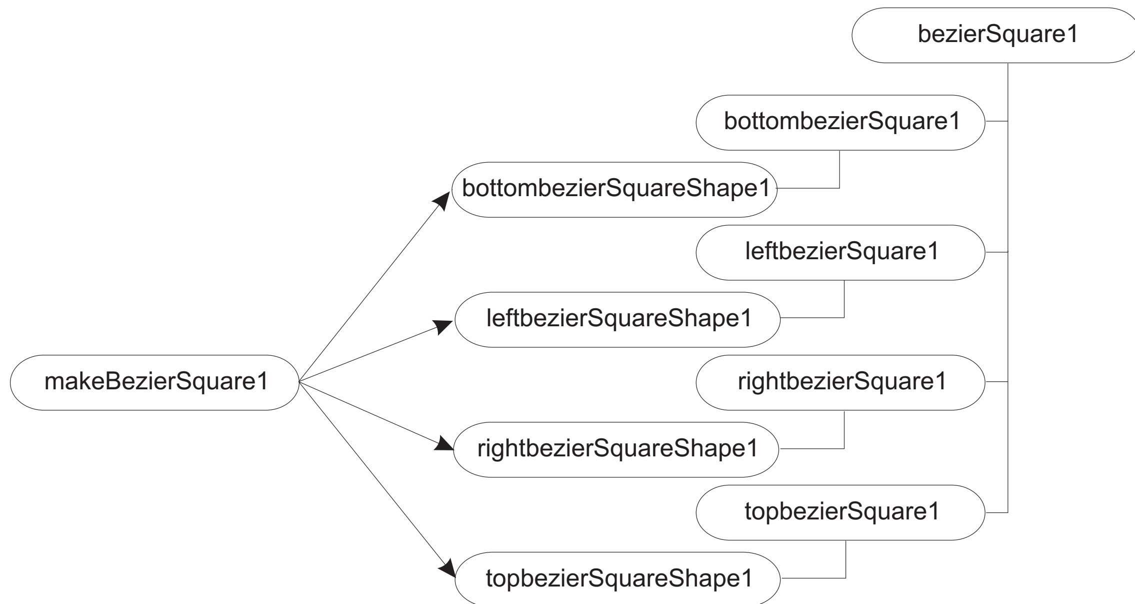
Bod P_{00} leží v počátku plochy, P_{01} pak určuje směr U a P_{10} směr V. Výsledná reprezentace v Maye potom vypadá následovně:

P_{00} P_{10} P_{20} P_{30} P_{01} P_{11} P_{21} P_{31} P_{02} P_{12} P_{22} P_{32} P_{03} P_{13} P_{23} P_{33}

Stejné uspořádání tedy bylo zvoleno i pro Bézierovy plochy. Aby bylo možné se v posloupnosti bodů orientovat, obsahuje geometrie také počet bodů v jednotlivých směrech, což je vlastně řád plochy ve směru U a V.

Uzel	**Atributy**
Shape křivek	cached (křivka), create (křivka), local (křivka), worldSpace (křivka), degree (short), dispCV (boolean), dispHull (boolean), dispGeometry (boolean), curvePrecision (short)
Shape ploch	cached (plocha), create (plocha), local (plocha), worldSpace (plocha), degreeUV (2short), degreeU (short), degreeV (short), dispCV (boolean), dispHull (boolean), dispGeometry (boolean), curvePrecisionU (short), curvePrecisionV (short), shadedPrecisionU (short), shadedPrecisionV (short), divisionsU (short), divisionsV (short)
Geometrie čtverce	outputCurve1 – 4 (křivka), sideLength1 (double), sideLength2 (double), degree (short), center (3double), normal(3double)
Geometrie plochy	outputSurface (plocha), width (double), lengthRatio (double), degreeU (short), degreeV (short), center (3double), normal(3double)
Geometrie krychle	outputSurface1 – 6 (plocha), width (double), lengthRatio (double), heightRatio (double), degree (short), center (3double), normal(3double)
Rozšíření křivek na plochu	inputCurve1 (křivka), inputCurve2 (křivka), outputSurface (plocha)
Křivka NURBS – Bézier	inputCurve (NURBS křivka), outputCurve (křivka)
Křivka Bézier – NURBS	inputCurve (křivka), outputCurve (NURBS křivka)
Plocha NURBS – Bézier	inputSurface (NURBS plocha), outputSurface (plocha)
Plocha Bézier – NURBS	inputSurface (plocha), outputSurface (NURBS plocha)

Tabulka 4.1: Přehled atributů navržených uzlů.



Obrázek 4.5: Příkaz vložení čtverce – struktura *Dependecy Graph* vč. hierarchie.

4.5.4 Návrh příkazů a kontextu

Návrh příkazů je relativně jednoduchý a není potřeba u nich vymýšlet nic nového. Stačí si jen rozmyslet jaké uzly musí vkládat do DG grafu, jak je napojit a vlastně také pojmenovat. Důležité je si uvědomit, že pokud příkazy vkládají uzly do grafu, mění tím vlastně stav Mayi a tudíž by měli implementovat metody pro podporu *undo* a *redo* mechanismu. u každého příkazu je uvedeno i jeho jméno, pomocí kterého může být v Maye volán.

Příkaz pro vložení čtverce

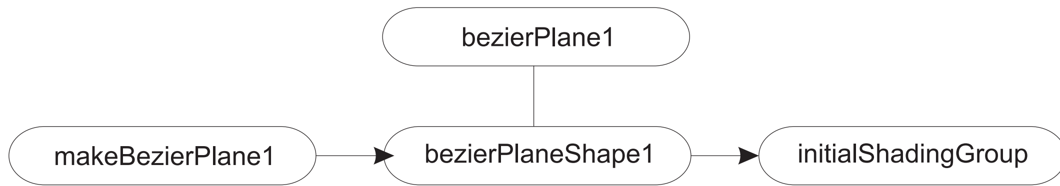
- *bezierSquare*

Úkolem tohoto příkazu je vytvoření uzlu pro definici příslušné geometrie, čtyřech *shape* uzlů Bézierových křivek a jejich vzájemné propojení. Způsob propojování, výčet atributů a jejich význam byl již popsán. Je tedy zřejmé, že uzly budou vzájemně propojeny přes atributy *outputCurve1 – 4* uzlu definujícího geometrii a *create* jednotlivých *shape* uzlů. Dalším úkolem příkazu je vytvořit příslušnou hierarchi *transform* uzlů nad novými *shape* uzly. Strukturu včetně příkladu pojmenování lze vidět na obr. 4.5.

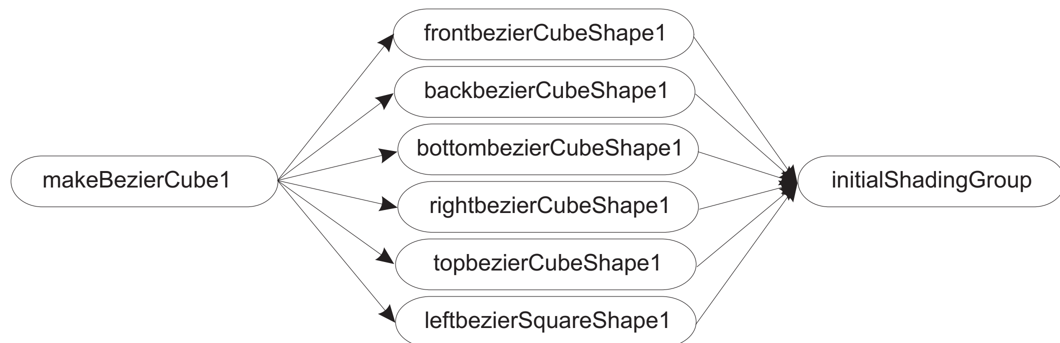
Příkaz pro vložení plochy ve tvaru čtverce

- *bezierPlane*

Tento příkaz musí vytvořit uzel pro definici dané geometrie a napojit ho na *shape* uzel Bézierovy plochy pomocí atributů *outputSurface* a *create*. Nově vytvořené plochy mají v Maye přiřazen uzel, který definuje počáteční stínování objektu (jeho jméno je *initialShadingGroup*). Napojení *shape* uzlu do tohoto uzlu je provedeno pomocí atributu *shape* uzlu *instObjGroups* (ten má standardně každý *shape* uzel) a *dagSetMembers* uzlu *initialShadingGroup*. Obr. 4.6 ukazuje výslednou strukturu.



Obrázek 4.6: Příkaz vložení plochy – struktura *Dependecy Graph* vč. hierarchie.



Obrázek 4.7: Příkaz vložení krychle – struktura *Dependecy Graph*.

Příkaz pro vložení krychle

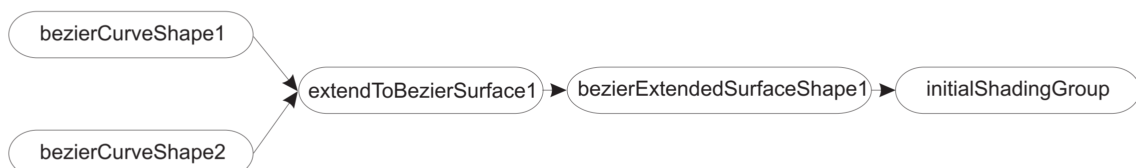
- *bezierCube*

U toho příkazu se nevyskytuje již nic nového. Vytváří a napojuje uzel s geometrií a šestici *shape* uzlů Bézierových ploch. Využívá atributy *outputSurface1 – 6* a *create*. Dále jednotlivé plochy napojuje na *initialShadingGroup*, jak ilustruje obr. 4.7. Hierarchie *transform* uzlů je obdobná jako u vložení čtverce, pro jednoduchost není na obrázku vyznačena.

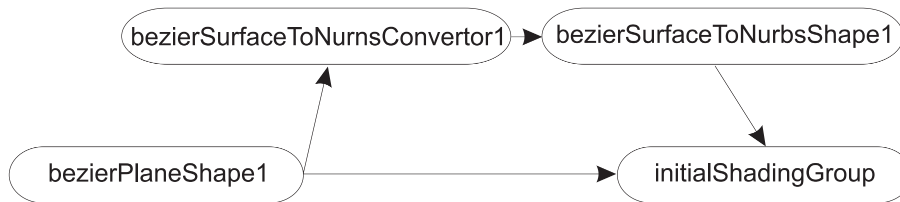
Příkaz pro rozšíření dvou křivek na plochu

- *extendToBezierSurface*

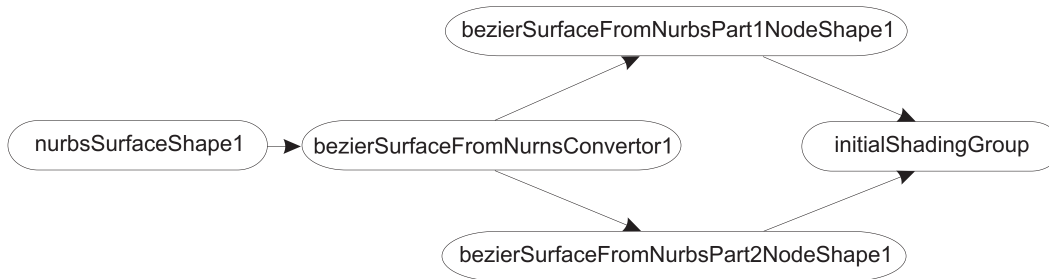
Předpokladem správné funkce příkazu je vybraná dvojice Bézierových křivek ze scény. Pokud vybrány nejsou, příkaz vypíše varování a nic se neprovede. V opačném případě vezme příslušné shape uzly obou křivek a napojí je do nově vytvořeného uzlu, který je zodpovědný za výpočet plochy. Toto napojení je realizováno pomocí atributů křivek *worldSpace* a atributů nového uzlu *inputCurve1*, *inputCurve2*. Nový uzel je pak následně



Obrázek 4.8: Příkaz rozšíření dvou křivek – struktura *Dependecy Graph*.



Obrázek 4.9: Příkaz převodu Bézier–NURBS – struktura *Dependecy Graph*.



Obrázek 4.10: Příkaz převodu NURBS–Bézier – struktura *Dependecy Graph*.

napojen do nového *shape* uzlu Bézierovy plochy (*outputSurface* – *create*) a ten opět do *initialShadingGroup*. Vše je vidět na obr. 4.8. *Transform* uzly jednotlivých objektů jsou opět na obrázku vynechány.

Příkazy převodu Bézier–NURBS

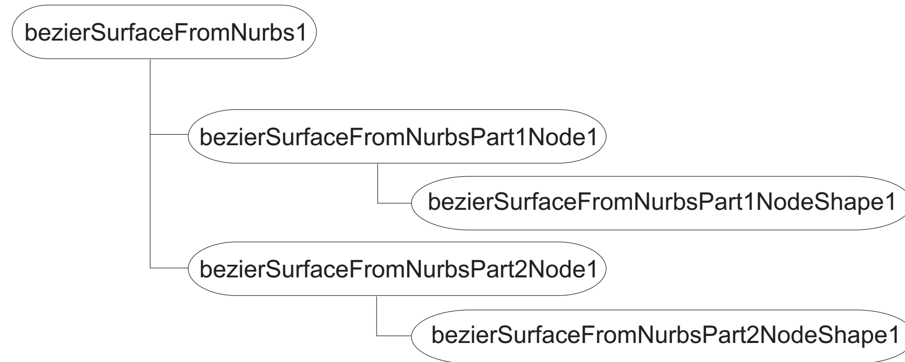
- *bezierCurveToNurbs*
- *bezierSurfaceToNurbs*

Příkazy pro převod křivek i ploch jsou v zásadě stejné. Oba vytvoří příslušný převodní uzel, který propojí s převáděným *shape* uzlem přes atributy *worldSpace* a *inputCurve* resp. *inputSurface*. Dále vytvoří nový *shape* uzel NURBS křivky resp. plochy a napojí ho na převodní uzel pomocí atributů *outputCurve* resp. *outputSurface* a *create*. V případě plochy ještě následuje napojení na *initialShadingGroup*. Příkazy vyžadují, aby byla vybrána nejméně jedna Bézierova křivka resp. plocha. Pokud jich je vybráno více, provede se převod všech. Dobrou vlastností je ignorování neodpovídajících objektů. Struktura DG grafu pro převod ploch je na obr. 4.9, opět bez *transform* uzlů.

Příkazy převodu NURBS–Bézier

- *bezierCurveFromNurbs*
- *bezierSurfaceFromNurbs*

Princip funkce těchto příkazů je v zásadě stejný s předchozími, rozdíl ovšem spočívá v počtu vytvořených *shape* uzlů. Algoritmus pro převod NURBS–Bézier vytváří obecně několik Bézierových křivek resp. ploch. Je tedy potřeba vytvořit příslušný počet *shape* uzlů. Aby bylo možné jednotně manipulovat se všemi převedenými křivkami resp.



Obrázek 4.11: Příkaz převodu NURBS–Bézier – hierarchie uzlů.

plochami, vytvoří se potřebná hierarchie. Příklad výsledného DG grafu pro plochy je na obr. 4.10, hierarchie je pro přehlednost oděleně na obr. 4.11.

Kontext a příkaz pro manuální vytváření křivky

- *bezierCurve*

Chování a vlastnosti výsledného nástroje by měli co nejvíce odpovídat nástroji pro vytváření NURBS křivek pomocí definice řídících bodů, jelikož uživatel pak bude moci oba nástroje využívat bez citelného rozdílu.
Tento kontext musí v zásadě reagovat na několik událostí myší či klávesnice. Co se týče myši, nejdůležitějšími jsou stisk pravého tlačítka, tažení při stisku a uvolnění. U klávesnice pak *enter* a *backspace*.
Chování nástroje lze popsat následovně. Kliknutím myši definuje uživatel řídící bod a výsledná křivka je ihned vykreslena do scény. Dalším kliknutím přidá bod do stejné křivky atd., dokud nestiskne *enter*, čímž danou křivku ukončí a může definovat další. Pokud uživatel při zadávání bodu stiskne pravé tlačítko, podrží ho a bude táhnout, přidaný bod se bude posouvat podle myši, dokud uživatel tlačítko neuvolní. Stiskem *backspace* lze smazat posledně zadaný bod, pokud ovšem nebyla křivka ukončena.
Součástí tohoto kontextu je příkaz *bezierCurve*, který je pro vytváření křivky využíván. Lze ho tedy volat i pomocí MELu, přičemž řídící body se zadávají v podobě argumentů. Vytvoření křivky ze tří bodů může vypadat takto:

```
bezierCurve -p 1.2 0 1.5 -p 0 0 0 -p 10.5 -3.5 0.0
```

Více argmentů než *-point* (*-p*) tento příkaz nebude potřebovat.
Se zadáváním bodů pomocí myši souvisí problém umístění bodu do prostoru, jedná se ale o implementační problém, tudíž je rozebrán v následující kapitole.

Tabulka 4.2 uvádí jména všech výše popsaných příkazů, případně jejich argumentů a jejich logické zařazení podle toho zda jsou využívány pro křivky či plochy.

4.5.5 Návrh modifikací GUI

Možnost volání všech příkazů z grafického uživatelského rozhraní je v podstatě standardní požadavek. Strukturu menu je vhodné rozdělit podle způsobu použití jednotlivých

Příkaz	**Použití**	**Argumenty**
bezierSquare	křivky	-
bezierPlane	plochy	-
bezierCube	plochy	-
extendToBezierSurface	plochy	-
bezierCurveToNurbs	křivky	-
bezierSurfaceToNurbs	plochy	-
bezierCurveFromNurbs	křivky	-
bezierSurfaceFromNurbs	plochy	-
bezierSurfaceAllToNurbs	plochy	-
bezierCurve	křivky	-point (-p)

Tabulka 4.2: Přehled navržených příkazů.

příkazů. Přihrádky pro ikony pak budou dvě, každá pro jeden typ objektu. Součástí ikon a položek menu budou nápovědné popisy. Ty se zobrazí ve stavovém řádky Mayi při najetí myší na ikonu či položku menu.

Menu nazvané *Bezier* bude mít následující strukturu:

- Bezier Primitives

  Square

  Plane

  Cube

- Bezier Conversions

  Bezier Curve To NURBS

  NURBS Curve To Bezier

  Bezier Surface To NURBS

  NURBS Surface To Bezier

  All Bezier Surfaces To Nurbs

- Extend To Surface

Příhrádky pro ikony nazvané *BezierCurves* a *BezierSurfaces* budou obsahovat následující položky:

- BezierCurves

  Bezier Square

  Convert To Bezier

  Convert To NURBS

  CV Bezier Curve Tool

- BezierSurfaces

  Bezier Plane

  Bezier Cube

  Extend To Surface

  Convert To Bezier

  Convert To NURBS

  Convert All To NURBS

5 Popis implementace

Následující text se zabývá podrobnostmi ohledně implementovaného modulu. Postupně je popsána struktura programu, jaké části C++ API byly využity a v poslední části kapitoly je věnována pozornost vlastní implementaci, zejména nejpodstatnějším problémům. Důležitou informací je, že programový modul byl implementován a zkompilován pro Mayu verze 7 pro operační systém Windows.

5.1 Základní struktura programu

Celý program reprezentující modul je v zásadě rozdělen na části odpovídající jednotlivým cílům práce. Každá implementovaná třída v programu vlastně představuje jeden konkrétní příkaz, uzel atd. Výjimkou jsou jen shape uzly, které mají třídy dvě, a nástroj, který má tři. Tato vlastnost je v podstatě dána C++ API.
Co se týče rozdělení programu do souborů, každá jednotlivá část má svůj *.h* a *.cpp* soubor. Toto umožňuje, při případném dalším rozšiřování modulu, použití jen potřebných specifických částí. Modifikace GUI jsou pak uloženy ve dvou *.mel* souborech a potřebné ikony v několika souborech *.bmp*.
Výsledný modul (tedy po zkompilování) se pak skládá z dynamické knihovny *.mll*, která obsahuje veškeré nové příkazy, uzly a kontexty. Dále se skládá z již zmíněných MEL skriptů pro modifikaci GUI a potřebných ikon. Postup instalace je podrobně posán v příloze B.

5.2 Přehled a popis použitých MPx tříd

Jak již bylo řečeno, implementace výše uvedených částí modulu v podstatě spočívá v implementaci tříd, které jdou potomky tzv. proxy tříd poskytovaných C++ API. Následuje tedy výčet a stručný popis použitých tříd. Popis metod je značně rozsáhlý a proto zde není uveden, lze ho nalézt v [1].

MPxComponentShape

Základní třída pro všechny uživatelsky definované *shape* uzly, tedy DG a zároveň DAG uzly. Obsahuje metody pro vykreslování, výběr a práci s komponentami. Díky této třídě tedy lze vytvořit nové geometrické objekty s možností výběru a manipulace komponent.

Použití – *shape* uzly křivek i ploch

MPxSurfaceShapeUI

Definuje metody pro vykreslování a interaktivní vybírání objektů i jejich komponent. Obsahuje tedy metody pro definici uživatelsky závislého chování daného *shape* uzlu. Tato třída je v podstatě nedílnou součástí třídy předchozí, jelikož bez ní by nebylo možné objekty vykreslovat a vybírat.

Použití – *shape* uzly křivek i ploch

MPxGeometryData

Tato třída poskytuje jakýsi obal, umožňující existenci libovolné geometrie uvnitř DG grafu v podobě dat. Vlastní geometrie je implementována v nezávislé třídě. Tato varianta umožňuje iteraci komponent a také použití deformátorů spolu s třídou *MPxGeometryIterator*.

Použití – data křivek i ploch

MPxGeometryIterator

Umožňuje použití iterátorů nezávisle na vlastní geometrii. Spolu s třídou *MPxComponentShape* dovoluje iteraci komponent uživatelsky definovaných geometrických objektů.

Použití – iterátory geometrie křivek i ploch

MPxNode

Třída pro vytváření obecných DG uzlů. Definuje pouze základní vlastnosti, které umožňují vytváření atributů, jejich napojování a výpočet v případě nějaké změny.

Použití – všechny uzly pro definici geometrie, převodní uzly

MPxCommand

Třída dovolující tvorbu nových MEL příkazů podporujících *undo* a *redo* mechanismus. Jakýkoli příkaz nějakým způsobem modifikující vnitřní stav Mayi by měl *undo* implementovat.

Použití – všechny příkazy vyjma příkazu obsaženého v kontextu

MPxToolCommand

Základní třída pro definici interaktivních příkazů volaných v určeném kontextu. Stejně jako příkazy vytvořené pomocí třídy *MPxCommand* je lze volat z MELu, obsahují však navíc některé metody pro použití v interaktivním kontextu.

Použití – příkaz pro manuální definování křivky

MPxContext

Tato třída slouží k vytváření uživatelských kontextů, díky kterým lze v Maye používat interaktivní nástroje. Lze definovat reakce na události myši či klávesnice. Je to např. stisk levého či prostředního tlačítka, táhnutí myší, podržení tlačítka myši, stisk klávesy *enter* či *backspace* atd.

Použití – kontext nástroje pro manuální vytvoření křivky

MPxContextCommand

Kontext je potřeba nejprve vytvořit. K tomu slouží příkaz, který lze implementovat právě pomocí této třídy.

Použití – vytvoření instance kontextu pro manuální vytvoření křivky

5.3 Implementace jednotlivých částí

Obsahem této části kapitoly je popis jednotlivých implementovaných tříd se zaměřením na problematické části. Vysvětleny jsou zde základní vazby mezi některými metodami a také důležité souvislosti. Na veškeré podrobnosti zde není z důvodu značného rozsahu prostor, lze je nalézt např. v knize [5] nebo v [1]. Kompletní přehled tříd, metod a jejich popis je uveden v dokumentaci zdrojového kódu na přiloženém CD. Před popisem vlastních částí modulu je ještě uvedeno několik základních informací.
Pro zavedení modulu do Mayi je nutné implementovat metodu *initializePlugin()*, která je zodpovědná za registraci všech potřebných uzlů, dat, příkazů atd. O zrušení se pak stará metoda *uninitializePlugin()*, která je volána při odstraňování modulu.
Téměř všechny metody poskytované C++ API vracejí mimo potřebných dat také návratové kódy. Maya totiž nevyužívá systém výjimek. Třída použitá pro návratové kódy se nazývá *MStatus*.
Maya identifikuje jednotlivé uzly a data pomocí *id* čísel. Aby tato identifikace mohla spolehlivě fungovat a nedocházelo ke konfliktům, musí mít každý objekt unikátní *id*. Pro potřeby vývoje a možnosti interního použití je vyhrazena oblast 0 – 0x7ffff, což je 524288 *id* čísel. Pokud by bylo potřeba využívat moduly globálně, musí se získat příslušná globálně unikátní *id*, která lze obdržet od vývojářů Mayi. Pro tento implementovaný modul byly zvoleny následující rozsahy:

Křivky: 0x15300 – 0x15304

Plochy: 0x26400 – 0x26406

Konkrétně zvolená *id* nejsou až tak podstatná, případně je lze nalézt ve zdrojovém kódu.

5.3.1 Shape uzly křivek a ploch

Třídy:

- Křivky

  CBezierCurve

  CBezierCurveUI

- Plochy

  CBezierSurface

  CBezierSurfaceUI

Nejprve se budeme věnovat Bézierovým křivkám.

Vlastnost	**Význam**
Readable	může být zdrojem spojení
Writable	může být cílem spojení
Connectable	může být napojován
Storable	je ukládán do souboru scény
Keyable	může být animován
Hidden	je skrytý
Cached	je ukládán do vyrovnávací paměti
Array	je pole
ArrayDataBuilder	využívá tvůrce dat
Internal	je interní
Worldspace	obsahuje data ve světových souřadnicích

Tabulka 5.1: Přehled obecných vlastností atributů.

Vytváření atributů

Rozhraní uzlu již bylo popsáno při návrhu, nyní popíšeme, jak je potřeba ho inicializovat. Inicializace spočívá ve vytvoření jednotlivých atributů včetně definování jejich názvů, datových typů a implicitních hodnot. Kromě toho se musí atributu definovat obecné vlastnosti. Tabulka 5.1 uvádí přehled nejdůležitějších vlastností a jejich stručný popis. V tabulce je vidět, že lze atributy označit jako interní. Toho se využívá ve dvou případech. Zaprvé pokud chceme nějakým způsobem reagovat na změnu atributu a zadruhé pokud potřebujeme ukládat data atributu ve své speciální struktuře odlišné od *dataBlock*. U těchto atributů je v případě nastavování resp. čtení hodnoty volána metoda *setInternalValueInContext()* resp. *getInternalValueInContext()*, ve které lze potřebné věci implementovat. Tohoto mechanizmu je zde využito jen pro implicitní atributy *mControlPoints* (pole bodů) a *mHasHistoryOnCreate* (příznak), jejich význam bude popsán dále.
Po vytvoření atributů je ještě potřeba určit závislosti atributů (tzn. jak se atributy navzájem ovlivňují). Definuje se vždy dvojice, která určuje jaký atribut vyžaduje přepočítání při změně druhého. Např. dále uvedený příkaz specifikuje závislost atributu *local* na atributu *create*:

```
attributeAffects(create, local);
```

Ukládání atributů

Metoda *shouldSave()* umožňuje definovat, které atributy budou ukládány do souboru. To je z důvodu efektivity, protože některé atributy se vypočítávají na základě jiných a není tedy potřeba jejich data ukládat. V našem případě jsou ukládány pouze atributy *mControlPoints* a *cached*.

Výpočetní funkce

Jakmile je uzel inicializován, je plně funkční, může být vložen do DG grafu a napojen na ostatní uzly. V případě, že Maya vyžaduje vyhodnocení nějakého atributu, je zavolána metoda *compute()* daného uzlu. Jako argumenty je jí předán tzv. *plug*, který dovoluje identifikovat příslušný atribut, a *dataBlock* umožňující přístup k datům.

Výpočet atributů, jako je například *degree*, není v zásadě nijak složitý. Spočívá v podstatě jen v získání geometrie obsažené v *cached* atributu, určení stupně křivky, uložení do atributu a zrušení příznaku *dirtyBit*. K získávání dat, přesněji řečeno ovladače dat (*dataHandle*), slouží zmíněný *dataBlock*, který dovoluje v zásadě dvojí způsob obdržení. Data je možno otevřít pro čtení (metoda *inputValue()*) nebo pro zápis (metoda *outputValue()*). První zmíněná metoda zajistí aktuálnost dat (tedy i jejich případně vyhodnocení) před vlastním obdržením. Druhá metoda nic nevyhodnocuje, jelikož bude hodnota dat přepsána. Výpočtu podobných atributů jako je *degree* se již dále zabývat nebudeme, princip je stále stejný.
Důležitější je mechanizmus výpočtu atributů *cached*, *local* a *worldSpace*. Jak již bylo vysvětleno, použití deformátorů s sebou přináší i nutnost implementovat aplikování offsetů. Začněme u atributu *create*, jehož hodnota je vyžadována na počátku výpočtu atributu *local*. Atribut *create* slouží pro načítání geometrie z jiných uzlů. Možné je ovšem také přímé definovaná geometrie, přičemž není tento atribut využíván (geometrie je vložena přímo do atributu *cached*). V těchto případech se o uzlu říká, že tzv. má nebo nemá konstrukční historii. V případě že historii nemá, *create* atribut se ignoruje. V opačném případě se zjistí jeho aktuální hodnota a uloží se do atributu *cached*. Indikace historie je zařízena pomocí implicitního atributu *mHasHistoryOnCreate*.
Nyní tedy známe dvě možnosti, jak lze nastavit hodnotu atributu *cached* (přímo nebo pomocí atributu *create*). Třetí možnost souvisí s aplikací offsetů přes atribut *mControlPoints*. Jak bylo řečeno, *mControlPoints* je interní atribut, do kterého se ukládají offsety (např. při deformacích). Na změny tohoto atributu je reagováno uložením právě do *cached* atributu, přičemž nastavovat lze jak jednotlivé řídící body geometrie, tak dokonce jejich jednotlivé souřadnice.
Průběh výpočtu atributu *local* je následující. Nejprve je v případě historie nastavena hodnota atributu *cached* pomocí *create*. Data *cached* jsou následně získána a pokud má uzel historii, jsou na ně aplikovány offsety z atributu *mControlPoints*. Výsledek je pak uložen do atributu *local*.
Zbývá atribut *worldSpace*. Jeho výpočet začíná výpočtem atributu *local* na jehož data je aplikována transformace do světových souřadnic. Matice pro tuto transformaci je získána v závislosti na instanci, pro kterou je atribut *worldSpace* vypočítáván. Přetransformovaná data jsou opět uložena jako výsledek.

Bounding box

Pro urychlení výpočtů Mayi je vhodné implementovat metodu *boundingBox()*, která na základě aktuální geometrie vypočítá omezující kvádr.

Aplikace nástrojů move, rotate a scale

Aby bylo možné používat tyto standardní nástroje Mayi, je potřeba implementovat metody *transformUsing()* a v případě možnosti napojení tzv. *tweak* uzlu pro aplikování offsetů i metodu *tweakUsing()*. Tyto metody jsou implementovány tak, aby umožňovaly jak transformaci celého objektu, tak jednotlivých řídících bodů. Podstatná je u nich zejména podpora ukládání a načítání bodů do vyrovnávací paměti poskytované Mayou (*pointCache*). Metoda *transformUsing()* ukládá, v případě existence historie, vzniklé rozdíly oproti původní geometrii do atributu *mControlPoints*. Tyto rozdíly jsou pak při výpočtu výstupní geometrie aplikovány jako offsety.

Režim	Význam
Bounding box	jen obalové kvádry objektů
Wireframe	jen drátové modely
Points	jen drátové modely z bodů
Flat shaded	neinterpolované stínovaní ploch
Smooth shaded	interpolované stínování ploch

Tabulka 5.2: Přehled režimů vykreslování.

Posun komponent ve směru normály nebo UV

Pro podporu nástroje Mayi dovolující posun jednotlivých řídících bodů v uvedených směrech je potřeba implementovat metodu *vertexOffsetDirection()*. Pro její implementaci je s výhodou využita funkční sada pro NURBS křivky, přičemž je Bézierova křivka nejprve převedena na NURB křivku (tento převod je triviální a nemůže nijak výrazně ovlivnit rychlost výpočtu směrů).

Reprezentace komponent

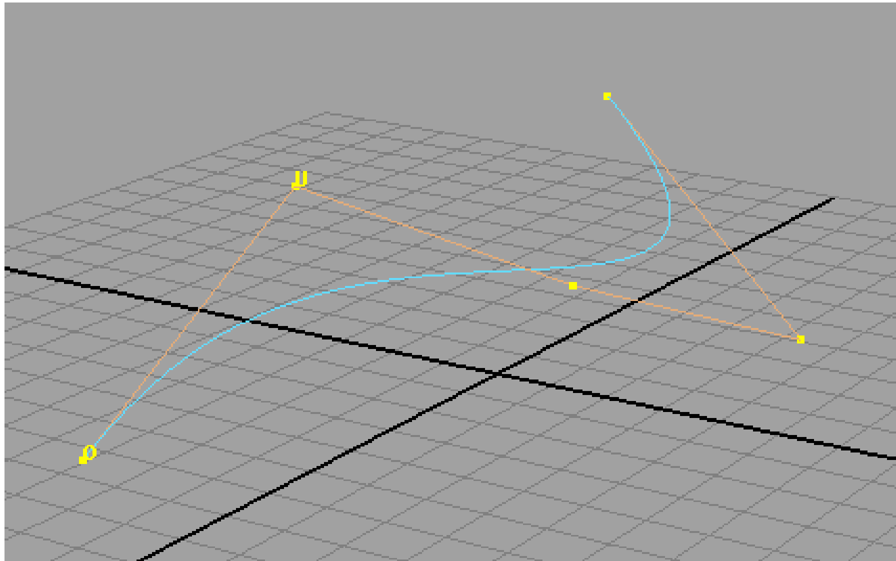
Za zmínku také stojí, jakým způsobem jsou reprezentovány komponenty tohoto uzlu. C++ API poskytuje celkem tři možnosti reprezentace. Jedná se o jednoduše, dvojitě a trojitě indexované komponenty. Pro křivky se evidentně hodí první varianta. S použitým typem reprezentace souvisí i další metoda *matchComponent()*, která slouží k převodu textového popisu komponent (při volání z MELu) na skutečné komponenty.

Mazání komponent

Pro mazání řídících bodů v geometrii, která byla definována přímo (tzn. uzel nemá historii), je potřeba implementovat metody *deleteComponents()* a *undeleteComponents()*. Ty na základě předaných komponent odstraní resp. vrátí řídící body. Rušení komponent pro uzly s historií je pak implementováno v rámci vlastních dat.

Vykreslování objektu

Maya definuje několik režimů vykreslování objektů ve scéně. Jejich přehled a stručný popis uvádí tabulka 5.2. Pro každý režim je pak potřeba implementovat příslušnou metodu, která pomocí *OpenGL* vykreslí geometrii do scény. Vlastnímu vykreslování se zde zabývat nebudeme, používány jsou běžné metody *OpenGL*, které lze najít např. na www stránkách [3]. Za zmínku stojí, že není potřeba nastavovat žádné parametry okna či scény, o vše se stará Maya. Vývojář musí pouze dbát na to, aby nezměnil nějaké atributy či transformační matice. Vhodné je tedy využívat zásobníky pro jejich ukládání a obnovu.
Vykreslování z hlediska volání metod probíhá ve dvou krocích. Nejprve jsou pomocí metody *getDrawRequests()* vytvořeny veškeré požadavky pro vykreslení. Může se jednat například o dva požadavky na vykreslení křivky a jejích komponent. Do daného požadavku je uložena příslušná geometrie, případně materiál vč. textury (u ploch) a také definice barev pro kreslení v závislosti na stavu objektu (pasivní, vybraný atd.). Vytvořený požadavek je pak zařazen mezi ostatní požadavky a předán Maye.



Obrázek 5.1: Příklad vykreslené křivky včetně řídících bodů a obalu.

Jakmile bude potřeba požadavek zpracovat, zavolá Maya metodu *draw()* a předá jí příslušný požadavek. Tato metoda na základě předaných informací objekt vykreslí. Obr. 5.1 ukazuje příklad vykreslené křivky včetně jejích řídících bodů a obalu (*hull*). Za povšimnutí stojí také vyznačení počátečního bodu O a určení směru U.

Výběr objektu

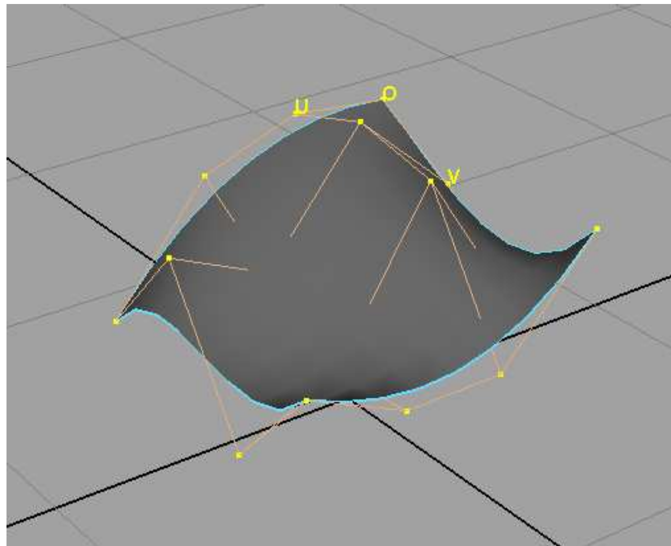
Pro zjednodušení výpočtů při rozhodování o vybraném objektu využívá Maya již zmíněný *bounding box*. Pokud rozhodne, že objekt může být teoreticky vybrán, zavolá metodu *select()*. Úkolem této metody je pak ověřit, zda byl daný objekt skutečně vybrán. Pokud ano, musí ho, příp. jeho komponenty, umístit do seznamu vybraných objektů (*selectionList*). Rozhodování zda je objekt vybrán, které je také rozděleno podle režimů vykreslování, lze realizovat opět pomocí *OpenGL*. C++ API definuje metody umožňující snadné nastavení *OpenGL* do výběrového režimu a pomocí běžného vykreslení je pak vše ověřeno.

Bézierovými plochami se zde budeme zabývat pouze stručně, jelikož jejich implementace je velice podobná implementaci křivek. Liší se pouze v některých jednoduchých atributech, jejichž výpočet se provádí standardním způsobem. Důležitější informací je snad jen fakt, že komponenty byly z důvodu jednoduchosti implementovány stejně jako u křivek, tedy jednoduše indexované. Tento druh komponent je totiž standardně připraven pro použití a není potřeba implementovat další metody. Příklad vystínované Bézierovy plochy včetně řídících bodů a *hullu* je na obr. 5.2.

5.3.2 Geometrická data

Třídy:

- Křivky



Obrázek 5.2: Příklad vykreslené vystínované plochy včetně řídících bodů a obalu.

CBezierCurveGeom

CBezierCurveGeomIterator

CBezierCurveData

- Plochy

  CBezierSurfaceGeom

  CBezierSurfaceGeomIterator

  CBezierSurfaceData

Data nových geometrických objektů jsou v podstatě implementována pomocí tří tříd. První definuje geometrii, druhá iterátor geometrie a třetí vlastní data, což je v podstatě jakýsi obal na geometrii.
Implementace geometrie křivek je velice jednoduchá a byla již v podstatě popsána při návrhu. Řídící body jsou uloženy v poli *MPointArray*, které je k dispozici v C++ API. Jediná metoda je přetížený operátor = pro snadné kopírování geometrie. Řídící body ploch jsou také uloženy v poli *MPointArray*, řád plochy v jednotlivých směrech pak ve dvou proměnných (*uOrder*, *vOrder*). Kromě přetíženého operátoru = dále obsahuje metodu pro výpočet bodu na ploše při zadaných parametrech U a V, metodu pro výpočet všech křivek drátového modelu a pomocnou metodu, která využívá algoritmus *deCasteljau* pro výpočet bodu na křivce. Zmíněný algoritmus lze nalézt např. v knize [7]. Varianta pro plochy je uvedena např. na stránkách [8].
Iterátory geometrií ploch i křivek jsou v zásadě stejné. Implementují klasické metody např. pro vynulování, posun iterátoru, získání počtu komponent nebo získání či nastavení dané komponenty. Metody jsou velice triviální.
Vlastní data křivek i ploch jsou také velice podobná. Obsahují metody pro ukládání a načítání geometrie do binárních a ACSII souborů, které se implementují pomocí proudů předaných v podobě argumentů. Způsob, jakým vývojář data do souborů uloží je čistě na něm, jelikož načítání realizuje také on. Dále jsou zde metody pro identifikaci, kopírování a

nastavení iterátoru. Asi nejdůležitější metodou z hlediska uživatele je mazání komponent v případě, že *shape* uzel má historii. Mazání v tomto případě probíhá tak, že Maya vloží uzel pojmenovaný *deleteComponent* do cesty vstupní geometrie hned před *shape* uzel. Cílem této metody pak je smazat příslušné řídící body v přiřazené geometrii. Její implementace je velice podobná implementaci metody pro mazání komponent *shape* uzlu.

5.3.3 Ostatní DG uzly

Třídy:

- Křivky

  CBezierCurveSquare

  CBezierCurveFromNurbs

  CBezierCurveToNurbs

- Plochy

  CBezierSurfacePlane

  CBezierSurfaceCube

  CBezierSurfaceExtend

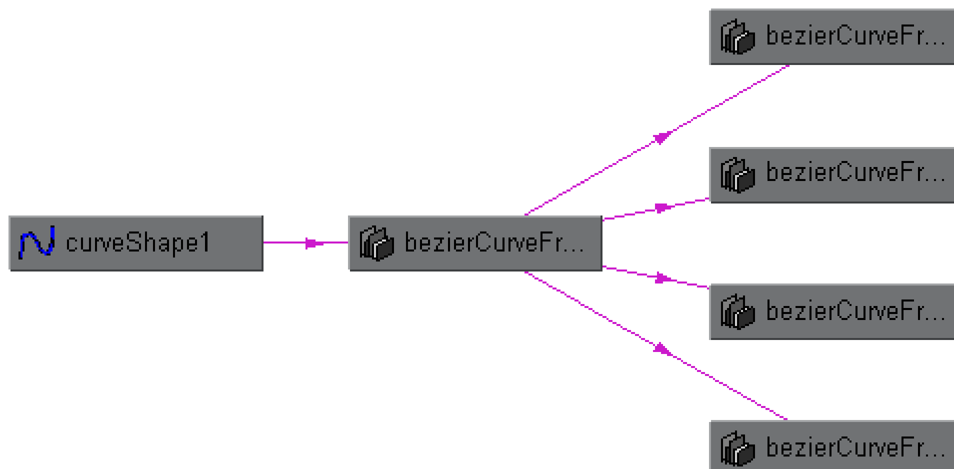
  CBezierSurfaceFromNurbs

  CBezierSurfaceToNurbs

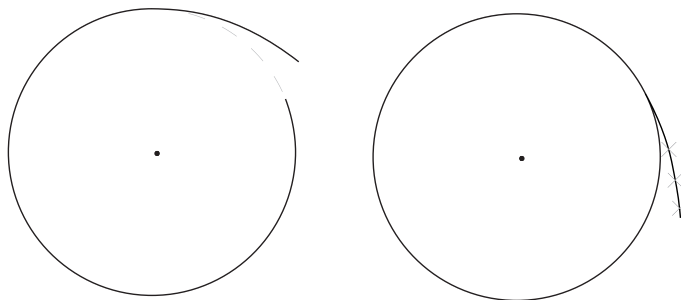
Tyto uzly jsou v zásadě velice jednoduché. Postup inicializace atributů se nijak neliší (až na vlastní atributy samozřejmě) od postupu uvedeného pro *shape* uzly. Jádro těchto uzlů spočívá ve výpočetní funkci *compute()*.
Úkol této metody je vypočítat výstupní geometrii v závislosti na vstupních atributech. Problém tedy spočívá čistě ve výpočtu řídících bodů. Za zmínku stojí jen výpočetní funkce převodních uzlů NURBS–Bézier.
Jak již bylo popsáno při návrhu, převod NURBS křivek resp. ploch může obecně vést na několik výsledných Bézierových křivek resp. ploch. S tímto faktem souvisí problém, který se vyskytne v případě modifikace původního objektu. Běžné transformace objektu a komponent nevadí, co ovšem vadí jsou úpravy vedoucí na změnu počtu segmentů původního NURBS objektu.

Představme si následující situaci. Máme nějakou NURBS křivku se 4 segmenty, kterou převedeme na 4 Bézierovy křivky. Reprezentace v DG grafu je na obr. 5.3. Tuto strukturu vytvoří příslušný příkaz. Nyní provedeme odebrání jednoho bodu NURBS křivky, čímž se změní počet segmentů a tím pádem by měla jedna Bézierova křivka přestat existovat, jelikož již není výsledkem převodu. Jenže příkaz, který tuto strukturu vytvořil již není platný a samotný uzel sám sebe odstranit nemůže.
Tento problém byl vyřešen tak, že se sleduje počet segmentů převedeného NURBS objektu a pokud dojde ke změně, vypíše se varování a nový převod se neprovede. Ve scéně tedy zůstanou původní převedené Bézierovy objekty.

Další komplikací jsou periodické NURBS křivky či plochy. Algoritmus převodu uvedený v kapitole 4 nepočítal s možností převodu např. kružnic či koulí. Periodické NURBS křivky



Obrázek 5.3: Převod NURBS křivky – reprezentace v *Dependency graph*.



Obrázek 5.4: Ukázka špatného převodu NURBS kružnice a principu opravy.

jsou vytvořeny tak, že definují více řídících bodů než je ve skutečnosti potřeba s tím, že několik prvních a posledních bodů se překrývá. Tento počet je dán stupněm křivky. Pokud aplikujeme standardní algoritmus pro převod např. na již zmíněnou kružnici, vznikne několik Bézierových křivek, z nichž první křivka nenavazuje na poslední, jak je ilustrováno na levém obr. 5.4. Problém lze odstranit tak, že původní NURBS křivku rozšíříme na jejím počátku o řídící body z jejího konce (včetně uzlového vektoru) a to v počtu odpovídajícímu stupni křivky. Následně aplikujeme algoritmus pro převod a u výsledku zanedbáme první Bézierovu křivku, jak ukazuje pravý obr. 5.4. Pro plochy to platí analogicky.

Možnost uživatele manuálně měnit DG graf znamená, že může bez problémů vymazat uzly, kvůli kterým vlastně tyto uzly existují. Chování uzlů tedy bylo nastaveno tak, aby se v takovém případě automaticky odstranili z DG grafu.

5.3.4 Příkazy

Třídy:

- Křivky

CBezierCurveSquareCmd

CBezierCurveFromNurbsCmd

CBezierCurveToNurbsCmd

- Plochy

  CBezierSurfacePlaneCmd

  CBezierSurfaceCubeCmd

  CBezierSurfaceExtendCmd

  CBezierSurfaceFromNurbsCmd

  CBezierSurfaceToNurbsCmd

Struktura všech příkazů je v zásadě stejná. Pro modifikaci DG resp. DAG grafu využívají modifikátor poskytovaný C++ API. Tento modifikátor umožňuje tzv. nahrát příkazy a následně je provést. Lze si tedy předpřipravit požadovanou strukturu (vytvoření a propojení uzlů a jejich hierarchie) a tu následně vytvořit. Tohoto principu lze s výhodou využít při implementování metod pro podporu *undo* a *redo* mechanizmu. Základní metody příkazu jsou *doIt()*, *redoIt()* a *undoIt()*. Pokud příkaz zavoláme, provede se metoda *doIt()*, která nahraje příkazy do modifikátoru a strukturu vytvoří. Pokud uživatel provede *undo*, metoda *undoIt()* vše vytvořené zruší. *RedoIt()* pak umožnuje opětovné vytvoření, ale tentokrát bez nutnosti příkazy nahrávat, jelikož se využije modifikátor nahraný v metodě *doIt()*.
Co se týče *undo* mechanizmu, Maya provádí veškerou režii. Lze se tedy spolehnout, že při volání *undoIt()* resp. *redoIt()* je Maya v očekávaném stavu.
Důležité je, že příkaz nesmí ukládat ukazatele na objekty v podobě třídy *MObject*. Maya totiž nezaručuje jeho stálou platnost mezi jednotlivými voláními příkazu. Místo toho je vhodné si ukládat DAG cesty, ty jsou platné vždy.

5.3.5 Kontext

Třídy:

- CBezierCurveCVTool
- CBezierCurveCVContext
- CBezierCurveCVCmd

Jak již bylo popsáno, nástroj je vytvořen pomocí tří vzájemně provázaných tříd. Jedna se stará o kontext, druhá o jeho vytvoření a třetí reprezentuje prováděný příkaz.
Popišme si tedy jak spolu jednotlivé třídy spolupracují. Jestliže chceme v Maye vytvořit nástroj, musíme nejprve vytvořit kontext umožňující interaktivní práci uživatele s příkazem realizujícím požadovanou práci. Za vytvoření kontextu je zodpovědný zmíněný příkaz. Ten pouze vrací požadovaný objekt kontextu, který umístíme v podobě ikony do přihrádky nástrojů
Jakmile máme kontext, lze ho aktivovat (např. kliknutím na ikonu v přihrádce). Od této chvíle kontext reaguje na činnost uživatele. Funkce kontextu byla již popsána při jeho návrhu, zde se zaměříme na implementační problémy. Pokud uživatel definuje body, v

pozadí je vlastně prováděn příkaz, který vytvoří příslušný *shape* uzel v DG grafu (při definování prvního bodu) a nastaví geometrii do atributu *cached shape* uzlu. Při dalším zadávání bodů, jejich posouvání nebo mazání se pouze aktualizuje geometrie přímo v *shape* uzlu. Metody *doPress()*, *doDrag()* a *doRelease()* umožňují implementovat reakce na stisk, táhnutí a uvolnění tlačítka. *DeleteAction()* a *completeAction()* pak na stistk kláves *backspace* a *enter*.
Metody *doPress()*, *doDrag()* i *doRelease()* jsou, až na drobnosti související s nastavením pomocných příznaků, implementovány stejně. Jejich úkolem je zjistit pozici posledně definovaného bodu. Uvědomme si, že uživatel zadává body ve 2D prostoru (obrazovka resp. daný pohled – *viewport*) a příkaz vyžaduje body v 3D prostoru. Tento problém je vyřešen pomocí metod třídy *M3dView* poskytované C++ API. Jedna z jeho metod dokáže vypočítat souřadnice bodů, které vzniknou protnutím paprsku definovaného stiskem myší s blízkou resp. vzdálenou rovinou daného promítání. Pokud máme tyto body, lze již pomocí průniku s nějakou rovinou získat požadovaný 3D bod. Zmíněná rovina je určena na základě kamery, která definuje pohled, do kterého uživatel klikl:

standardní kamera *sideShape*: rovina yz

standardní kamera *frontShape*: rovina xy

jakákoli jiná kamera: rovina xz

Tato rovina je ještě posunuta v závislosti na posledním zadaném bodu (pokud ještě nebyl žádný zadán, tak do počátku). Například pokud byl posledně zadán bod (0.0 0.0 1.0) a nyní definuje uživatel bod v pohledu kamery *frontShape*, bude výsledná rovina xy posunuta do $z = 1.0$. Toto chování bylo vypozorováno z nástrojů pro NURBS. Nyní tedy zpět k našim metodám. V momentě kdy máme získán 3D bod, nezbývá než modifikovat (či prvně vytvořit) geometrický objekt, prostřednictvím příkazu. Po vytvoření objektu je ještě potřeba aktualizovat všechny pohledy Mayi.
Metoda *deleteAction()* pouze odebírá řídící body geometrie prostřednictvím příkazu a v případě smazání všech řídících bodů odstraní i vytvořený *shape* uzel.
Při dokončení zadávání má metoda *completeAction()* za úkol nastavit kontext do počátečního stavu pro nové zadávání a také volání metody *finalize()* příkazu pro vytvoření objektu. Tato metoda má za úkol registrovat posledně realizovaný příkaz do Mayi. Toho je nutné z důvodu korektní funkce *undo* mechanizmu, jelikož při práci kontextu nejsou žádné příkazy opticky prováděny (na příkazovou řádku nebylo nic vloženo), *finalize()* toto napravuje. Ukončení daného kontextu se provede automaticky vybráním jiného nástroje Mayi.
Vlastní příkaz se až na metodu *finalize()* a použití argumentů nijak neliší od ostatních příkazů. Vytváření a zpracování argumentů je implementačně velice jednoduché, proto nemá cenu se jím podrobněji zabývat.

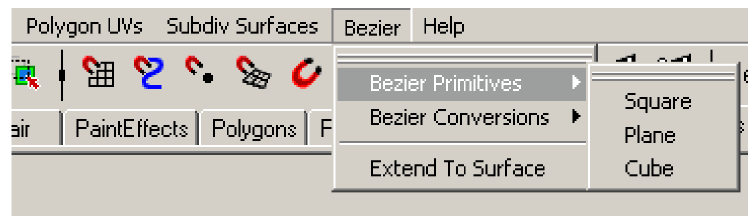
5.3.6 Úprava GUI

Skripty:

- shelf_Bezier_Load.mel
- shelf_Bezier_Unload.mel

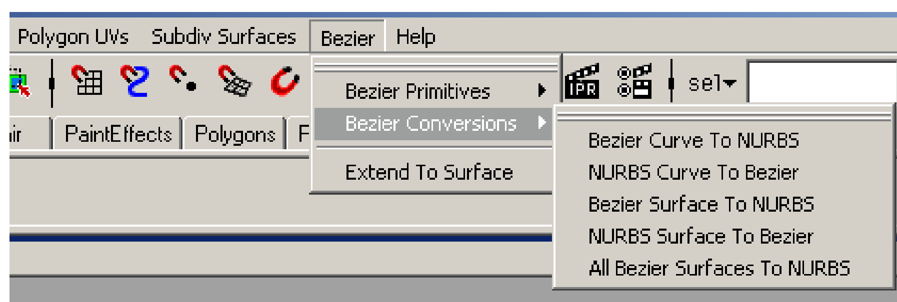
Příkaz	Použití
addNewShelfTab	přidání nové přihrádky
shelfButton	přidání příkazu do přihrádky vč. ikony a popisky
toolButton	přidání nástroje do přihrádky vč. ikony a popisky
menu	vytvoření nového menu
menuItem	přidání položky menu
setParent	nastavení objektu, který bude modifikován

Tabulka 5.3: Přehled MEL příkazů použitých k modifikaci GUI.

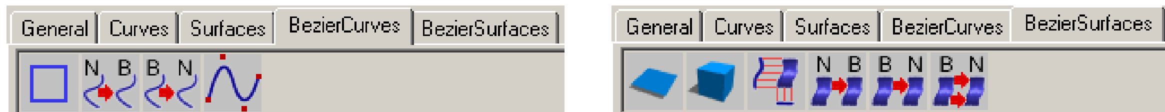


Obrázek 5.5: Vytvořené menu obsahující nové příkazy – první část.

Grafické uživatelské rozhraní je upravováno při inicializaci a odstraňování modulu. C++ API totiž poskytuje metodu pro volání MEL skriptu z programu. Jedná se tedy o výše zmíněné MEL skripty, z nichž první má za úkol vytvořit menu a přihrádky s ikonami, druhý pak zrušení přihrádek (vyžaduje potvrzení uživatelem). Menu se při vypnutí Mayi zruší automaticky. Použity jsou standardní příkazy MELu, jejich přehled je v tabulce 5.3. Podrobný popis jejich použití a argumentů lze nalézt v [1].
S modifikacemi GUI ovšem souvisí jeden problém. Nástroj pro vytvoření křivky, lze podle dokumentace C++ API do přihrádky umístit jen způsobem, který vyžaduje předchozí inicializaci modulu (naproti tomu obyčejné příkazy lze umístit i bez toho, jen nebudou funkční). Vložení ikony nástroje tedy funguje, pokud modul načteme po nahrání Mayi ručně. Její zrušení je také bez problému. Pokud ale vypneme Mayu bez zrušení modulu resp. přihrádek, Maya si přihrádky zapamatuje (menu si neukládá) a při příštím spuštění se je pokusí znovu načíst. Jenže v tuto chvíli není inicializován potřebný modul, takže nástroj nelze vytvořit, což dá Maya patřičně najevo.



Obrázek 5.6: Vytvořené menu obsahující nové příkazy – druhá část.



Obrázek 5.7: Vytvořené přihrádky pro Bézierovy křivky a plochy.

Ikony pro reprezentaci příkazů mohou být vytvořeny v libovolném grafickém editoru, přičemž lze využít např. formát *.bmp*. Pro dodržení rozměrů je vhodné vyjít z některé ze stávajících ikon, které lze nalézt v instalačním adresáři Mayi. Obr. 5.5 – 5.7 ukazují realizované menu a přihrádky.

6 Zhodnocení a návrh dalšího rozvoje práce

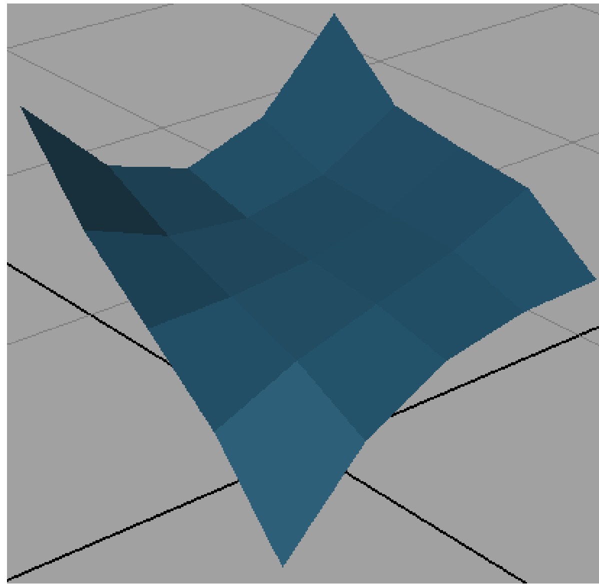
Následující text má za úkol zhodnotit výsledné vlastnosti implementovaného modulu, potažmo jeho nových uzlů, příkazů, nástrojů atd. Zahrnuto je celkové zhodnocení funkčnosti modulu a jeho jednotlivých částí včetně uvedení známých nedostatků. Obsaženo je i hrubé porovnání s adekvátními nástroji a prvky pro NURBS. Z uvedených nedostatků pak vycházejí návrhy na jejich odstranění, což může být obsahem dalších navazujících prací.

6.1 Funkčnost modulu a jeho částí

Modul lze celkově ohodnotit jako funkční a dobře použitelný. To znamená, že ho lze bez obtíží integrovat do Mayi a využívat jeho prvků. Během ověřování jeho funkcí, které spočívalo v simulování běžné práce s Mayou, se nevyskytly žádné závažné chyby, což ovšem nedokládá jeho bezchybnost. Vlastní program obsahuje řádově tisíce řádků kódu, z čehož vyplývá, že chyby nelze stoprocentně vyloučit. Maya je velice složitý a komplexní systém, do kterého je poměrně obtížné proniknout. Dokumentace C++ API sice popisuje poskytované třídy a vlastní rozhraní, nepopisuje však způsob práce s některými podstatnými prvky API a se systémem Mayi jako takovým. S tím souvisí možný výskyt těžko odhalitelných chyb, které mohou nastat jen v určitých situacích. I přes tyto komplikace se ovšem zdá, že je modul v praxi použitelný. Vědomé nedostatky jsou popsány dále u jednotlivých prvků tohoto modulu.
Vlastní testování modulu nemůže v podstatě probíhat jinak, než prováděním běžných úkonů spojených s modelováním. Ověřeny byly veškeré příkazy a nástroje, které ve spojitosti s novými objekty mohou připadat v úvahu. Jako příklad lze uvést vytvoření plochy pomocí základního tvaru, modifikace jejích řídících bodů, aplikace materiálu či textury, převod na NURBS plochu a převod zpět. Další možností je například použití deformátorů na nový objekt, čímž se potvrdí korektní funkčnost navrženého rozhraní s ohledem na aplikaci offsetů. Veškeré kombinace, které mohou vzniknout při modelování, není samozřejmě možné postihnout. Nyní se zaměřme na jednotlivé části modulu.

Shape uzly křivek a ploch

Rozhraní těchto uzlů se zdá být dobře navržené a implementované, jelikož mají všechny předpokládané vlastnosti. Práce s objektem i komponentami je možná v plném rozsahu možností. Lze tedy, jak již bylo zmíněno, aplikovat i standardní deformátory Mayi.
Pokud porovnáme tyto vlastnosti s NURBS, zjistíme, že se v zásadě nijak neliší, až na již zmíněnou reprezentaci komponent u Bézierových ploch. Ty byly totiž implementovány jako jednoduše indexované z důvodu zjednodušení. NURBS plochy naproti tomu využívají dvojité indexování. Tato odlišnost však není nijak zásadní. Pro uživatele je tato změna při práci s grafickým rozhraním téměř neviditelná. Podstatný rozdíl se objeví až při označování komponent pomocí příkazové řádky MELu, kde se jednotlivé řídící body Bézierovy plochy indexují souvisle po sloupcích narozdíl od jednoho indexu pro sloupce a jednoho pro řádky, jak je tomu u NURBS.
Co se týče vykreslování, nově vytvořené geometrické objekty podporují veškeré potřebné režimy Mayi, stejně jako NURBS. Možná jsou i nastavení kvality vykreslovaného objektu. Jediný a v zásadě nepodstatný rozdíl je u režimu *Flat Shaded*, kde mají Bézierovy plochy



Obrázek 6.1: Vystínovaná NURBS plocha – normály ploch.

definované normály ve vrcholech, kdežto NURBS v plochách. Na obr. 6.1 lze vidět vystínovanou NURBS plochu, na obr. 6.2 je pak zobrazena Bézierova plocha.
Problém nastává pokud chceme Bézierovy plochy renderovat. Geometrický objekt složený z tohoto typu ploch totiž nebude vykreslen. Dokumentace C++ API se o tomto problému nijak nezmiňuje, dokonce ani přiložené ukázkové příklady. Jeden z příkladů dokonce ilustruje vytváření nových objektů včetně nové geometrie, ale výsledné objekty nejsou také při renderováni vykresleny. Díky převodním nástrojům však lze tento problém obejít. Implementovaný příkaz pro převod všech Bézierových ploch ve scéně dovoluje jediným kliknutím vše převést na NURBS a scéna může být v zápětí renderována.

Geometrická data

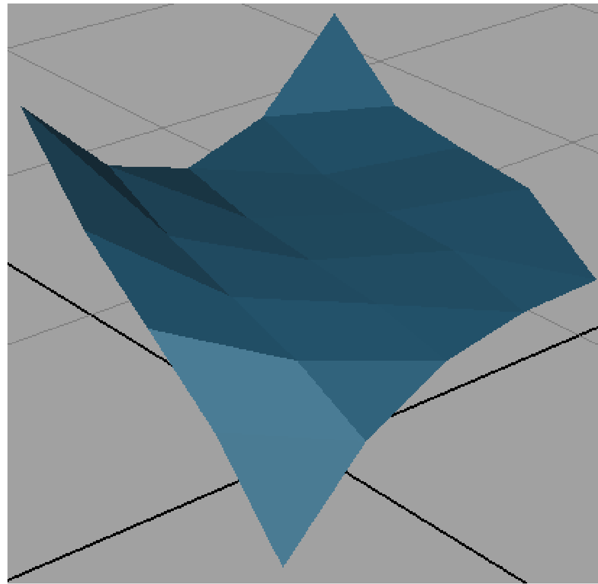
Nově vytvořená data mohou být bez omezení využívána všemi DG uzly. Lze je tedy využívat stejně, jako jakákoli jiná data v Maye. Jediný problém spočívá v již zmíněném renderování Bézierových ploch.

DG uzly definující geometrii

Tyto uzly jsou v podstatě stejné jako uzly pro definici geometrie NURBS, zejména co se týče rozhraní. Poskytují tedy téměř stejné možnosti nastavení výsledné geometrie.

Převodní DG uzly

Uzly sloužící pro převod Bézierových křivek a ploch na NURBS a zpět, vycházejí ze standardního chování jiných převodních uzlů, které Maya obsahuje. Jejich funkčnost je tedy v souladu s chováním ostatních uzlů podobného typu. Možnost konverze mezi jednotlivými typy parametrických křivek a ploch s sebou přináší jednu nesmírnou výhodu. Pro modelování pomocí Bézierových křivek a ploch lze podle potřeby využívat i nástroje

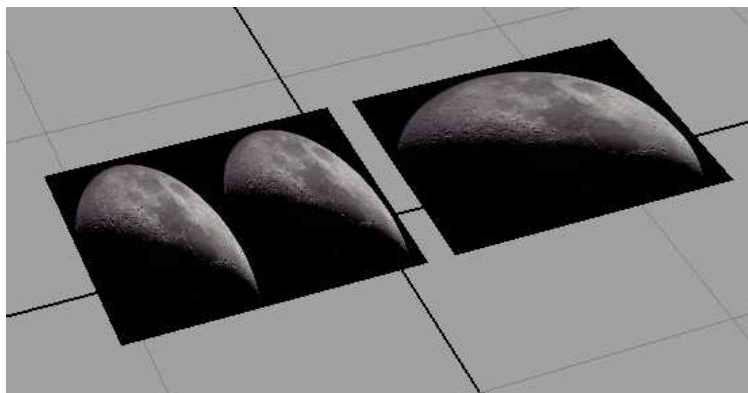


Obrázek 6.2: Vystínovaná Bézierova plocha – normály vrcholů.

pro NURBS. Stačí např. Bézierovu plochu převést na NURBS (převod se provede i včetně materiálů atd.) aplikovat potřebné nástroje a výslednou plochu převést na původní typ. Jedinou nevýhodou je, že zpětný převod může vést na více než jednu Bézierovu plochu. U převodu NURBS–Bézier je potřeba zmínit jeho neefektivitu spojenou se způsobem výpočtu atributů typu pole v Maye a principem běhu algoritmu pro převod. Jak již bylo vysvětleno, výstupní geometrie konverzního uzlu tohoto typu definuje obecně více než jednu křivku. Je tedy potřeba, aby výstupní atribut obsahoval daný počet složek. Maya při vyhodnocování pak volá výpočet právě těchto složek jednotlivě podle potřeby. V danou chvíli je tedy potřeba jen ten segment Bézierovy křivky či plochy, který odpovídá požadované složce. Algoritmus převodu ovšem musí probíhat vždy od začátku NURBS plochy či křivky. Je tedy zřejmé že dochází ke zbytečným opakovaným výpočtům. Pokud je např. vyžadován první segment a v zápětí druhý, první segment bude vypočítáván v podstatě dvakrát. Vzhledem k rychlosti výpočtu se ovšem znatelné zpomalení objeví až u NURBS ploch skládajících se z mnoha desítek segmentů.
Dalším problémem převodu ploch NURBS–Bézier je aplikování textur. Pokud totiž budeme převádět NURBS plochu s texturou, která obsahuje více než jeden segment, aplikuje se textura stejně na všechny výsledné Bézierovy segmenty. Příklad je na obr. 6.3, kde je vpravo zobrazena NURBS plocha skládající se ze dvou segmentů a vlevo dvě převedené Bézierovy plochy. To je spojeno s tím, že textura je součástí materiálu definovaného v příslušném *shading* uzlu, ten je napojen na všechny převedené segmenty a texturovací souřadnice jsou generovány při vykreslování u všech objektů identicky.

Příkazy

Jednotlivé příkazy byly realizovány tak, aby umožňovaly základní funkčnost, tedy hlavně vytvoření potřebných struktur v DG grafu. Kromě příkazu, který využívá kontext pro manuální vytváření křivky, nemají definovány žádné argumenty. To tedy například znamená, že příkaz pro vložení čtverce neumožňuje přímo modifikovat parametry výsledného



Obrázek 6.3: Ilustrace problému s texturováním při převodu NURBS–Bézier.

objektu. To lze provést až následně pomocí příkazů MELu nebo přes editor atributů. Plná funkčnost je tedy k dispozici, jen není dosažitelná přímo.

Modifikace GUI

Vytvořená menu a přihrádky s ikonami jsou plně funkční a nelze je po načtení v zásadě odlišit od standardních. Ovšem vzhledem k již dříve uvedeným problémům, které jsou spojeny s vytvářením ikony pro nástroje, nelze v podstatě tento modul automaticky načítat při spuštění Mayi. V opačném případě bude kromě výpisu varovného hlášení také nefunkční ikona daného nástroje. Správného chování tedy docílíme ručním načtením modulu a ručním zrušením před ukončením Mayi.

6.2 Další možná rozšíření práce

Možnosti rozšiřování této práce v zásadě vycházejí z nedostatků jednotlivých částí a také uvedených neefektivit. Pokud tedy provedeme shrnutí několika předchozích odstavců, získáme následující návrhy na rozšíření.
Shape uzly křivek i ploch by bylo vhodné v budoucnu rozšířit o další atributy, které budou poskytovat data potřebná k různým výpočtům. V tuto chvíli se jedná kromě vlastních geometrických dat jen o stupeň dané křivky či plochy. Dále, pokud by byla vyžadována vysoká efektivita prováděných výpočtů, bylo by vhodné využívat vnitřních atributů pro ukládání potřebných dat. Například výpočet obálky (*bounding box*) by se pak prováděl jen pokud by skutečně došlo ke změně geometrie.
Jelikož v tuto chvíli nelze přímo renderovat Bézierovy plochy, bylo by rovněž vhodné prozkoumat problematiku renderování nových geometrických objektů v Maye. Tím by se odstranila nutnost převodu na NURBS.
Zmíněná neefektivita převodu NURBS–Bézier spojená s opakovaným výčtem některých segmentů by mohla v rozsáhlých scénách způsobit znatelné zpomalení. Z tohoto důvodu by bylo rozumné výpočet zefektivnit. Jeden z možných způsobů by mohl opět spočívat v použití vnitřních atributů. Problém s texturováním by se pak mohl vyřešit separátním výpočtem texturovacích souřadnic pro jednotlivé segmenty Bézierových ploch. V tuto chvíli také není možný převod NURBS ploch s otvory, což není jistě triviální problém.

Pro snadnější a rychlejší používání příkazů definujících geometrii by bylo vhodné je doplnit o potřebné argumenty, které by přímo nastavovaly parametry generované geometrie. Rozšiřování modulu jako takového je jinak samozřejmě dále možné. Doplnit lze nástroje umožňující různé spojování křivek a ploch nebo naopak jejich dělení. Možností je opravdu hodně, záleží spíše na praktických požadavcích koncových uživatelů.

7 Závěr

Jak již bylo uvedeno, výsledný modul je plně funkční a použitelný. Po načtení modulu je v zásadě možné využívat všechny jeho části stejně, jako lze využívat standardní nástroje. Chování nových nástrojů je z uživatelského hlediska téměř stejné jako u ostatních, což usnadňuje zejména jejich používání. Pro možné nasazení je ovšem potřeba jeho další vývoj, jelikož ne vše je zcela dokončeno. V zásadě lze ale prohlásit, že byly realizovány všechny části specifikované v zadání této diplomové práce.

Podstatným přínosem této práce není jen vlastní modul, který umožňuje využití nových typů objektů k modelování, ale rovněž vysvětlení některých principů a postupů spojených s vytvářením takto rozsáhlého modulu pro program Maya. Proniknutí do systému tohoto programu není jednoduché a zabere relativně mnoho času. Z tohoto důvodu je pak vývoj takového modulu poměrně náročný, což by mohl tento text v kombinaci s vlastním kódem programu značně zjednodušit. Výsledky této práce by měly také objasnit nebo přinejmenším zviditelnit některé skryté problémy spojené s rozšiřováním vlastností zmíněného programu.

Vzhledem k poměrně dobré perspektivě modelovacího programu Maya, by jistě nebylo další rozšiřování této práce zbytečné a mohlo by vést na vytvoření velice užitečného modulu.

8 Seznam literatury

[1] Alias Systems Corp. Maya Help for Maya 7.0.
Nápověda programu obsažená v instalačním adresáři.

[2] Autodesk Maya: Product Information.
http://www.autodesk.com/maya.

[3] U. Behrens. OpenGL Reference, 1995.
http://www.mevis.de/opengl/opengl.html.

[4] D. A. D. Gould. Osobní stránky autora knih Complete Maya Programming.
http://www.davidgould.com/.

[5] D. A. D. Gould. *Complete Maya Programming (An Extensive Guide to MEL and the C++ API).* Morgan Kaufmann Publishers, 1st edition, 2003.

[6] D. A. D. Gould. *Complete Maya Programming Volume II (An In-depth Guide to 3D Fundamentals, Geometry and Modelling).* Morgan Kaufmann Publishers, 1st edition, 2005.

[7] L. A. Piegl and W. Tiller. *The NURBS Book (Monographs in Visual Communication).* Springer, 2nd edition, 1996.

[8] C. K. Shene. CS3621 Introduction to Computing with Geometry Notes, 2003.
http://www.cs.mtu.edu/~shene/COURSES/cs3621/NOTES/.

[9] J. Sloup. Maya 6 API Programming, 2004.
http://www.cgg.cvut.cz/~sloup/maya/html/.

A Seznam použitých zkratek

2D Two-dimensional

3D Three-dimensional

API Application Programming Interface

ASCII American Standard Code for Information Interchange

DAG Directed Acyclic Graph

DG Dependency Graph

GUI Graphical User Interface

MEL Maya Embedded Language

NURBS Non Uniform Rational Basis Spline

OpenGL Open Graphics Library

B Instalační a uživatelská příručka

Instalace modulu

Instalace modulu je velice snadná. Stačí nakopírovat potřebné soubory do adresářů definovaných Mayou. Na přiloženém CD je obsažena instalační dávka pro operační systém Windows, která vše potřebné provede. Tato dávka předpokládá standardní umístění potřebných adresářů. Pokud je tedy toto umístění změněno, musí se nakopírování provézt ručně nebo lze příslušné cesty v instalační dávce změnit. Standardní umístění potřebných adresářů je v domovském adresáři uživatele v podadresáři Dokumenty. Dávka provádí kopírování následujících souborů do uvedených adresářů:

Modul

Soubor: `bezier.mll.`

Adresář: `%HOMEDRIVE%%HOMEPATH%\Dokumenty\maya\7.0\plug-ins`

Skripty

Soubory: `shelf_Bezier_Load.mel, shelf_Bezier_Unload.mel.`

Adresář: `%HOMEDRIVE%%HOMEPATH%\Dokumenty\maya\7.0\scripts`

Ikony

Soubory: `bezierCurveBtN.bmp, bezierCurveCV.bmp, bezierCurveNtB.bmp, bezierSurfaceBtN.bmp, bezierSurfaceExtend.bmp, bezierSurfaceNtB.bmp, bezierSurfaceAllBtN.bmp.`

Adresář: `%HOMEDRIVE%%HOMEPATH%\Dokumenty\maya\7.0\prefs\icons`

Proměnné `%HOMEDRIVE%` a `%HOMEPATH%` definují umístění domovského adresáře uživatele. Pokud cílové adresáře neexistují, instalační dávka je vytvoří.

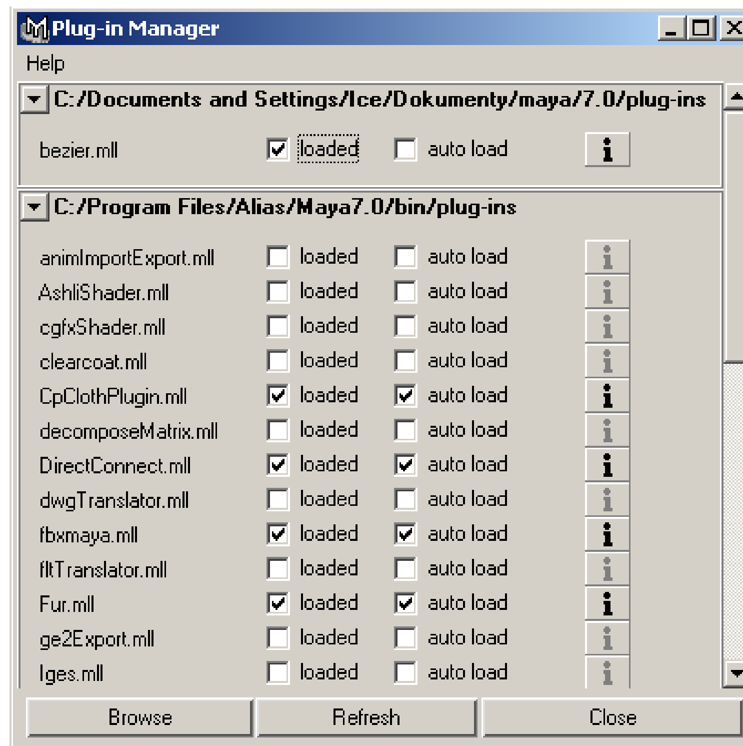
Načtení modulu

Nakopírování všech těchto souborů umožní přímé načtení modulu v Maye. To lze provést dvěma způsoby. První spočívá ve využití příkazu pro načtení modulu. Do příkazové řádky lze tedy napsat:

```
loadPlugin bezier;
```

Zrušení modulu se pak provádí příkazem:

```
unloadPlugin bezier;
```



Obrázek B.1: Manažer modulů – okno.

Důležité ovšem je, že aktuální scéna nesmí obsahovat žádné objekty definované tímto modulem. Scénu je nejprve potřeba zavřít.

Druhou možností je využití manažeru modulů. Ten lze nalézt v menu:

Window | Settings/Preferences | Plug-in Manager

Zde je možné požadovaný modul načíst a případně zrušit. Na obr. B.1 je zobrazeno okno tohoto manažeru. Automatické načítání při spuštění Mayi se z výše uvedených důvodů nedoporučuje. Ze stejných důvodů je potřeba před ukončením programu modul zrušit.

Používání příkazů a nástrojů

Po načtení lze využívat veškeré nástroje a příkazy definované modulem. Jejich chování se nijak neliší od standardních. Příkazy lze volat jak na příkazové řádce, tak pomocí menu či ikon v přihrádkách. Položky menu a ikony v přihrádkách samozřejmě obsahují popisky, které se při najetí myší zobrazí ve stavovém řádku Mayi.
Úkolem této příručky není popis ovládání programu Maya jako takového, to lze nalézt v nápovědě programu [1]. Vysvětleno je zde jen specifické použití jednotlivých příkazů a nástrojů.

Vytváření základních geometrických tvarů

- `bezierSquare`

- `bezierPlane`
- `bezierCube`

Příkazy pro vytvoření čtverce, plochy a krychle vytvoří po zavolání danou geometrii. Její parametry lze pak standardně měnit prostřednictvím editoru atributů nebo příkazů MELu pro změnu hodnoty atributu.

Rozšíření dvou křivek na plochu

- `extendToBezierSurface`

Tento příkaz vyžaduje, aby před jeho zavoláním byly vybrány dvě Bézierovy křivky. Na základě jejich geometrie pak vytvoří výslednou plochu.

Převod Bézier–Nurbs a opačně

- `bezierCurveToNurbs`
- `bezierSurfaceToNurbs`
- `bezierCurveFromNurbs`
- `bezierSurfaceFromNurbs`
- `bezierSurfaceAllToNurbs`

Uvedené příkazy mají identické chování. Před zavoláním je potřeba vybrat příslušné křivky resp. plochy, které budou převedeny. Vybrat jich lze libovolné množství. Objekty jiných typů budou ignorovány. Jedinou výjimkou je příkaz `bezierSurfaceAllToNurbs`, který převede všechny Bézierovy plochy ve scéně, není tedy potřeba nic vybírat.

Manuální vytvoření křivky

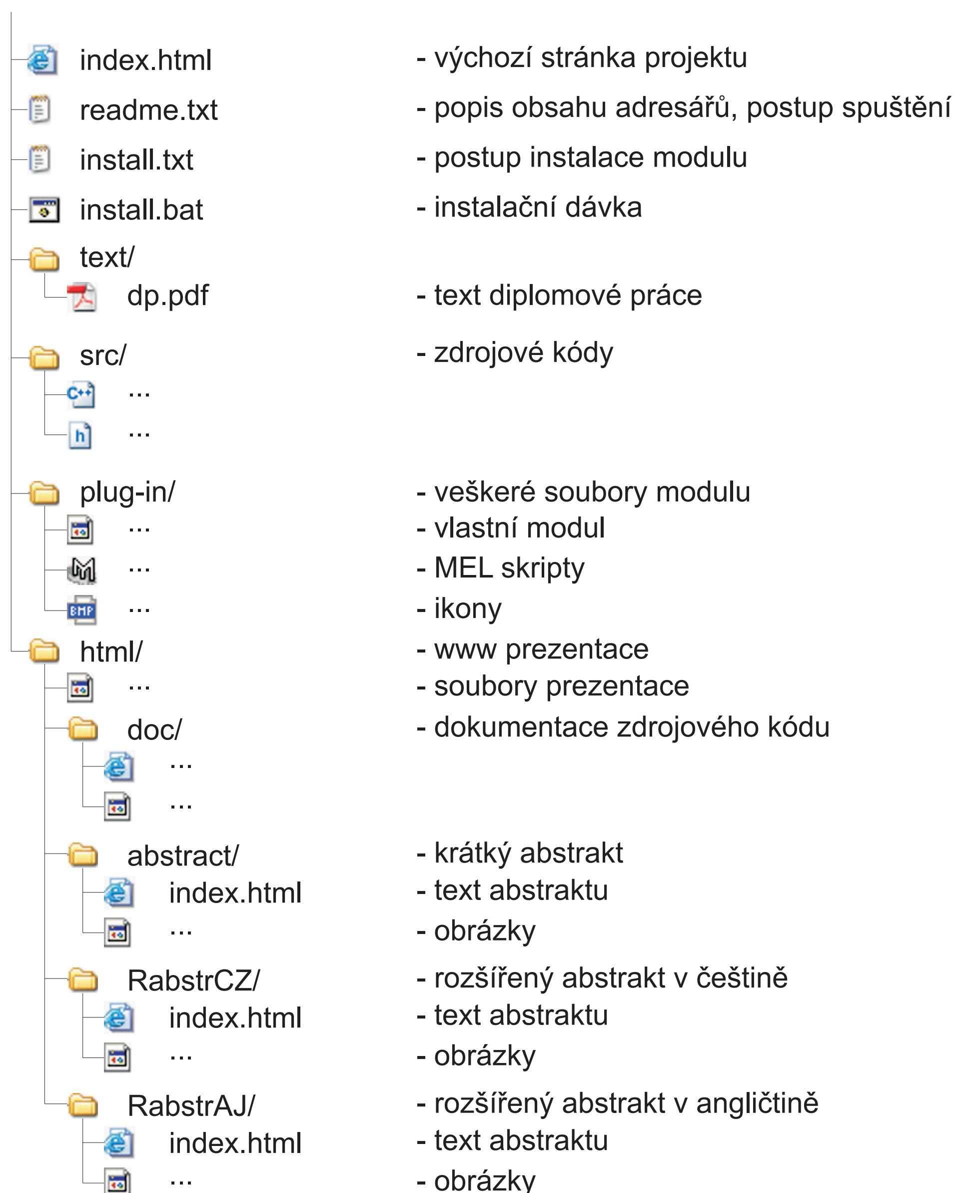
- `bezierCurve`

Tento nástroj lze aktivovat jen ikonou v přihrádce pro Bézierovy křivky. Následně lze pomocí myši přímo definovat řídící body vytvářené křivky. Kliknutím na pravé tlačítko myši je řídící bod přidán. Podržením tlačítka a táhnutím je pak možné plynule posouvat nový řídící bod. Uvolněním tlačítka je pak bod umístěn. Klávesou *backspace* lze zrušit posledně definovaný bod a klávesou *enter* se provede ukončení dané křivky. Následně lze vytvářet další křivku. Ukončení tohoto nástroje se provede vybráním libovolného jiného nástroje, např. výběru. Chování je velice podobné obdobným nástrojům pro NURBS a proto není nutné se jím zabývat do podrobností.
Křivku lze také vložit z příkazové řádky pomocí výše uvedeného příkazu. Řídící body se definují pomocí argumentů `-point` (`-p`) např. takto:

```
bezierCurve -p 0 0 0 -p 1 1.5 1 -p 2.5 0.0 3.2
```

C Obsah přiloženého CD

Přiložené CD obsahuje všechna data spojená s touto diplomovou prací. Jedná se zejména o vlastní modul, zdrojové kódy a text práce. Dále je pak zahrnuta www prezentace včetně dokumentace zdrojového kódu a instalační dávka. Na obr. C.1 je struktura zobrazena.



Obrázek C.1: Struktura přiloženého CD.