Evolute - Dokumentace

Jan Kolomazník

Obsah

Část I

Uživatelská dokumentace

Kapitola 1

Uživatelské rozhraní

Uživatelské rozhraní se skládá z hlavního menu (pokud neobsahuje žádné položky, není zobrazeno) a pracovní plochy, do které se umísťují okna modulů grafického rozhraní.

Rozměry a polohy oken modulů lze libovolně měnit, dle zvyklostí v jiných programech tažením za okraje a záhlaví, vždy však pokrývají celou pracovní plochu (viz. náhled).

V záhlaví každého okna se po kliknutí pravým tlačítkem myši zobrazí kontextové menu. Část menu s možnostmi manipulace s daným podoknem je shodná pro všechny druhy oken, navíc však může každý modul přidat své vlastní příkazy (podrobnosti u každého modulu).

Výčet společných položek:

- *Split* - podmenu pro dělení okna.
  - *Split vertically left* - svisle oddělí z daného okna nové, původní zůstane vlevo.
  - *Split vertically left* - svisle oddělí z daného okna nové, původní zůstane vpravo.
  - *Split horizontally top* - vodorovně oddělí z daného okna nové, původní zůstane nahoře.
  - *Split horizontally bottom* - vodorovně oddělí z daného okna nové, původní zůstane dole.
- *Choose module* - podmenu pro výběr aktivního modulu zobrazeného v daném okně. Starý se ukončí a je nahrazen nově vybraným.

1.1 Viewport

Viewport je modul sloužící k zobrazování scény a interaktivní práci s objekty scény pomocí myši.

Funkčnost jednotlivých tlačítek myši se, až na prostřední, liší dle právě prováděných úkonů.

Prostřední tlačítko myši slouží k manipulaci s kamerou příslušející viewportu, nad kterým je kurzor:

- *Rotace okolo lokálních os X a Y* - tažením myši se stisknutým prostředním tlačítkem myši.
- *Posun ve směru lokálních os X a Y* - tažením myši se stisknutým prostředním tlačítkem myši a drženou klávesou *Shift*.
- *Přibližování a oddalování kamery* - otáčení kolečka na myši.

Více o ovládání bude popsáno v části o práci s programem 2.

1.2 Console

Modul pro komunikaci se skriptovacím rozhraním, které využívá jazyk Python, jehož libovolné příkazy lze použít.

Pro snadnější práci je přítomna historie příkazů, přístupná, stejně jako v jiných konzolových aplikacích, pomocí šipky nahoru - posun zpět a šipky dolů - posun kupředu v historii.

Více o skriptování v 3.

1.3 Edit object

Rozhraní pro editaci objektů a to jak jejich vztahy ke scéně (pozice, natočení, ...), tak parametry a pro geometrické objekty správu *zásobníku modifikátorů* a jejich editaci.

Rozhraní je rozděleno do několika záložek. První slouží k určování globálních parametrů jako je barva použitá k vykreslení ve viewportu, pozice objektu ve scéně a natočení podle základních souřadných os.

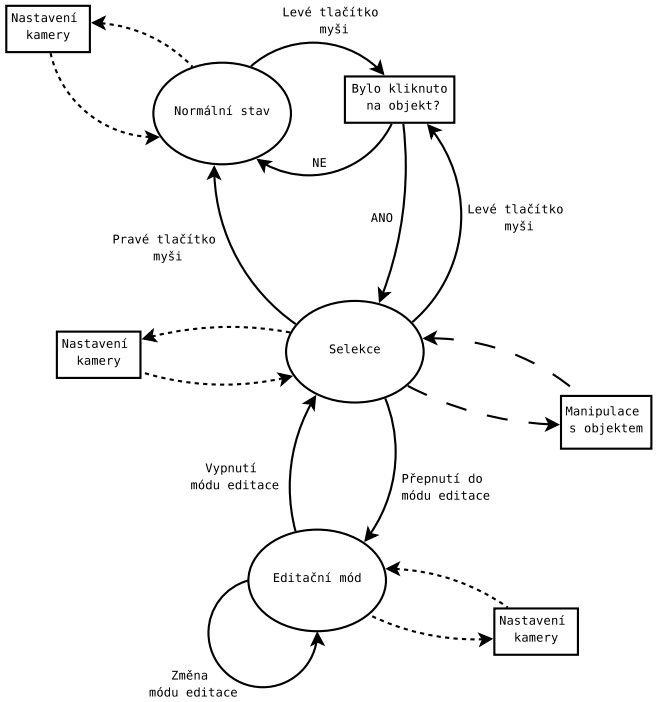
Druhá záložka slouží k práci se záobníkem modifikátorů. První položkou je vždy zdrojová geometrie - tou může být obecný geometrický objekt nebo nějaká speciální geometrie řízená vlastními parametry (koule, válec, ...). Dalšími položkami jsou modifikátory geometrie seřazené podle pořadí aplikace na objekt. Pokud má nějaký modifikátor nějaký speciální mód editace je vedle názvu modifikátoru malá ikonka plus, která rozbaluje seznam editačních módů, ke kterým se přistupuje stejně jako k jednotlivým modifikátorům selekcí dané položky v seznamu.

Při výběru modifikátoru nebo jeho módu editace se pod zásobníkem modifikátorů zobrazí komponenty pro nastavení parametrů právě vybraného modifikátoru. Při výběru módu editace se navíc celá aplikace přepne do speciálního editačního módu, kdy je většina uživatelských událostí (práce s myší, apod.) posílána k ošetření modifikátoru ve vybraném editačním módu.

Více v sekci o práci s programem 2.

Kapitola 2

Práce s programem



Obrázek 2.1: Stavy programu a přepínání mezi nimi

2.1 Manipulace s objekty scény

Po spuštění je program v *Normálním* editačním stavu. V tomto stavu lze pouze měnit nastavení kamer, příslušných jednotlivým viewportům (viz. 1.1).

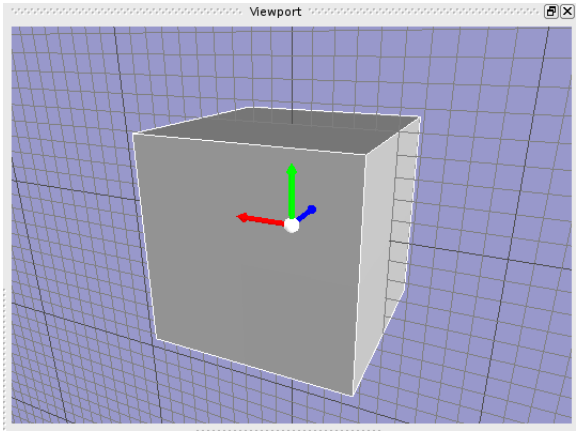
Kliknutím levého tlačítka myši do viewportu se otestuje zda pod kurzorem myši leží nějaký objekt scény. Pokud je odpoveď kladná je daný objekt vybrán a program se přepne do stavu *Selekce*. V tomto stavu je možná manipulace s kamerou, přesouvání vybraného objektu pomocí myši, nastavování pozice a natočení zadáním souřadnic, měnit barvu objektu a pracovat se zásobníkem modifikátorů.

Když je objekt vybrán je zobrazován jeho *manipulátor* [1] na souřadnicích *pivotu* [2] příslušného

[1] Objekt sloužící k interaktivnímu nastavování přidruženého objektu, příp. objektů.
[2] Bod objektu, ke kterému jsou vztaženy veškeré transformace a lokální soustava souřadnic.

objektu. Manipulátor je vykreslen jako šipky ve směrech základních souřadných os. Kliknutím na některou z nich a tažením myši dojde k přesouvání objektu ve směru vybrané osy podle pohybu myši. Při přesunu je vykreslována pouze tak osa v jejímž směru dochází k posunu.

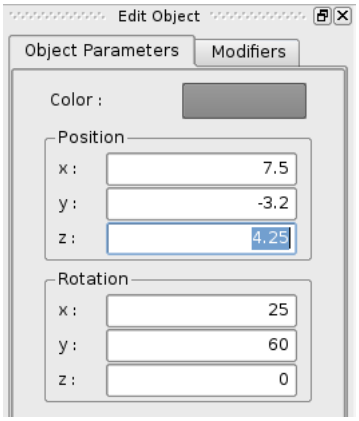
Pokud v průběhu tažení stisknete pravé tlačítko myši, je přesun zastaven a objekt se vrátí do původní polohy před začátkem manipulace.



Obrázek 2.2: Vybraný objekt s manipulátorem

Na první záložce GUI modulu *Edit object* lze číselně nastavit pozici a natočení vybraného objektu ve scéně. Kliknutím na barevné tlačítko v položce *Color* se zobrazí dialogové okno pro výběr barvy, která bude použita při vykreslování objektu.

Na druhé záložce pak lze manipulovat se zásobníkem modifikátorů (více viz 2.2).



Obrázek 2.3: Nastavování barvy, pozice a natočení

2.2 Modelování

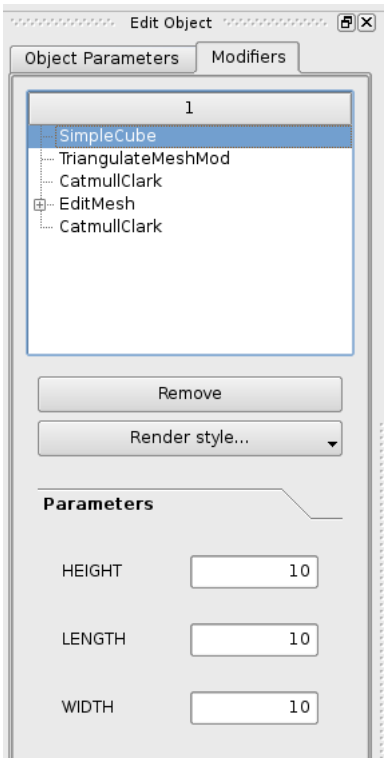
Pokud chceme začít modelovat potřebujeme vytvořit nějakou základní geometrii, která se stane základem naší práce. Kupříkladu mnoho modelovacích tutoriálů začíná s obyčejným kvádrem.

Několik standartních objektů (koule, kvádr) by mělo být přístupno z hlavního menu aplikace, případně je lze vytvořit příkazem z konzole (viz. 3).

Pokud již máme nějaký objekt ve scéně (a není vybrán) kliknutím ve viewportu levým tlačítkem myši na žádaný objekt dojde k jeho selekci. Pravým pak naopak k jeho deselekci.

Když máme vybrán objekt změní se způsob jeho vykreslování a navíc se zobrazí manipulátor objektu.

V GUI modulu *Edit object* se přepneme na záložku *Modifiers*, v horní části je umístěn zásobník modifikátorů s tlačítkem *Remove* pro odstranění vybraného modifikátoru a *Render style...*, jež mění způsob vykreslování objektu v závislosti na editaci.



Obrázek 2.4: Zásobník modifikátorů a editor parametrů

Nejvyšší záznam v zásobníku modifikátorů odpovídá zdrojové geometrii, další záznamy pak jednotlivým modifikátorům v pořadí jak byly aplikovány. Kliknutím na některý z nich se pod zásobníkem modifikátorů zobrazí seznam parametrů vybraného modifikátoru, jež je možné přímo editovat. Množina parametrů je pro každý modifikátor jiná.

Nový modifikátor se aplikuje na vybraný objekt, buď příkazem z hlavního menu, nebo z konzole (viz. 3). Výsledný vzhled geometrie je okamžitě přepočítán. Aktuálně vybraný modifikátor (s výjimkou zdrojové geometrie) lze jednoduše odstranit stisknutím již zmiňovaného tlačítka *Remove*.

Při editaci nějakého modifikátoru, který je někde hluboko v zásobníku, je často výhodné vidět nejen geometrii vybraného modifikátoru, ale i výslednou geometrii. To lze nastavit pomocí menu, jež se zobrazí po stisknutí tlačítka *Render style...* :

Selected modifier - vykreslí pouze geometrii aktuálně vybraného modifikátoru. Při editací se nebudou přepočítávat modifikátory nad ním. Výhodné při editaci složitého objektu.

Result - vykreslí pouze výslednou geometrii.

Selected & Blend result - vykreslí geometrii vybraného modifikátoru standartně a výslednou geometrii průhledně.

Result & Blend selected - výslednou geometrii vykreslí standartně a geometrii aktuálně vybraného modifikátoru vykreslí průhledně.

Editační módy

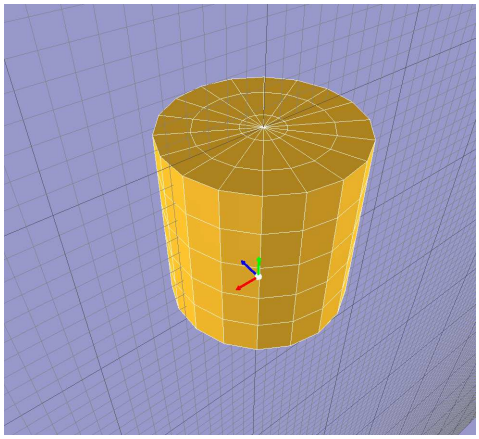
Některé z modifikátorů mají vedle svého názvu v zásobníku modifikátorů nakreslené malé +. Kliknutím na něj se rozbalí seznam editačních módů daného modifikátoru. Přepnutím se do editačního módu se aplikace dostane do stavu s jiným ovládáním, které je závislé na daném modifikátoru.

Důvodem pro existenci editačních módů je častá nutnost složitějšího nastavování modifikátorů. Například modifikátor *EditMesh* potřebuje myš pro selekci a přesuny vrcholů, hran a plošek.

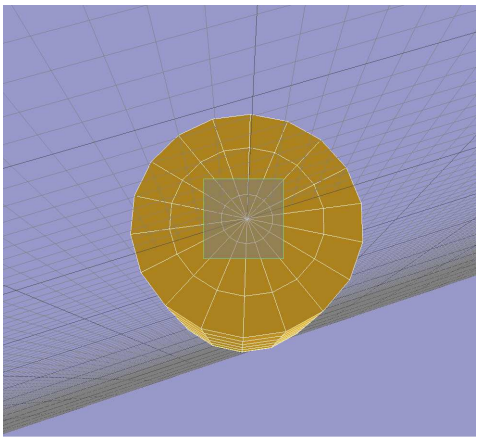
Návod na vymodelování hrnečku

Jako ukázku modelovací techniky uvedu jednoduchý návod na vymodelování hrníčku na kávu. Obrázky ke každému z kroků jsou uvedeny za návodem.

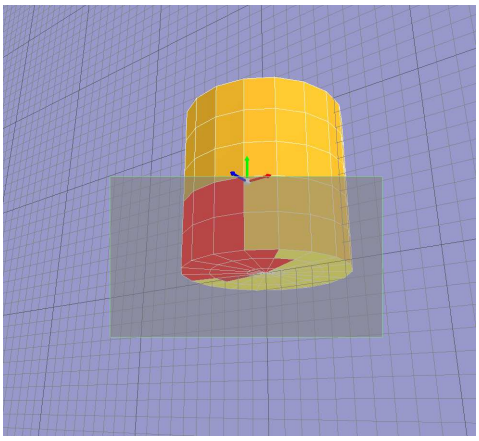
1. Nejdříve vytvoříme válec a nastavíme mu hustotu dělení podstav (SEG_C) na 3, hustotu výškového dělení (SEG_H) na 5 a hustotu obvodového dělení (SEG_R) na 18.
2. Aplikujeme na válec modifikátor *Edit mesh* a přepneme se do módu editace *Faces*. Tažením levým tlačítkem myši vytvoříme výběrové okno jak je vidět na obrázku a vybereme polygony ve střední části horní podstavy. Vybrané polygony zčervenají.
3. Okenní výběr však vybere polygyny i na druhé straně válce. To je nežádoucí proto opět použijeme okno výběru, ale tentokrát držíme stisknuté tlačítko *Ctrl*.
4. Dalším krokem je roztažení vybraných polygonů. Nastavíme v GUI parametr modifikátoru *Scale* na 1.3 (nezapomeňte stisknout Enter) a stiskneme tlačítko *Scale*.
5. Nastavíme parametr *Extrude* na -9 (opět stisknout Enter) a aplikujeme pomocí tlačítka.
6. Podle obrázku vybereme polygony na straně (můžeme upravit mírně jejich pozici pomocí manipulátoru). Výběr provádíme kliknutím, přidání do selekce se provádí se stisknutým tlačítkem *Shift*.
7. Vybrané polygony postupně vysuneme pomocí Extrude s hodnotami 1.2, 3.0, 1.5.
8. Vysuneme dva polygony na boku námi vytvořených výčnělků podle obrázku.
9. Naposled vytažené polygony odstraníme pomocí tlačítka *Delete selection*.
10. Přepneme se do módu editace *Vertices*. Postupným výběrem protilehlých vrcholů a aplikováním *Collapse vertices* spojíme oba otevřené konce.
11. Po ukončení ucha hrníčku se můžeme přepnout do módu editace *Edges* a doladit tvar hrníčku.
12. Posledním krokem je aplikace několika iterací vybraného SubSurf schématu. Ten jsme mohli aplikovat již dříve a sledovat jak se nám mění výsledná geometrie v závislosti na prováděných změnách



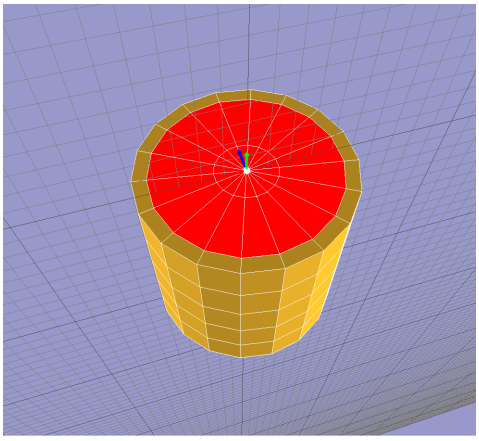
Obrázek 2.5: Krok 1.



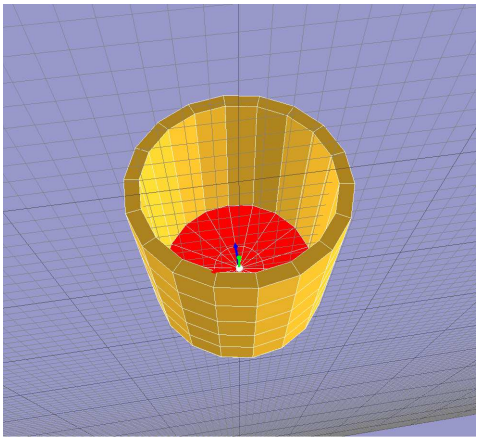
Obrázek 2.6: Krok 2.



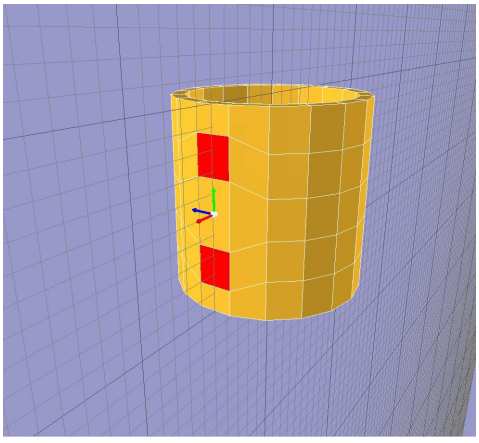
Obrázek 2.7: Krok 3.



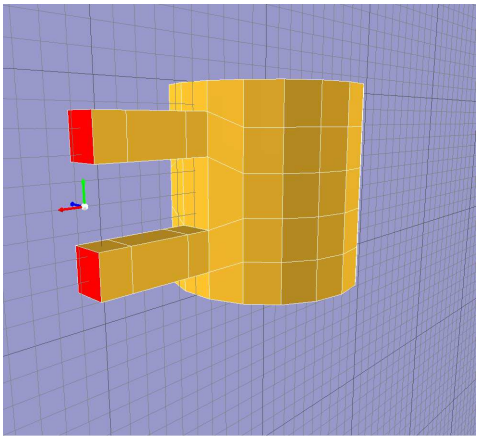
Obrázek 2.8: Krok 4.



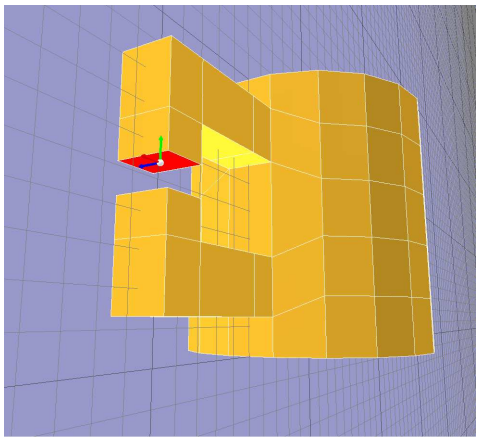
Obrázek 2.9: Krok 5.



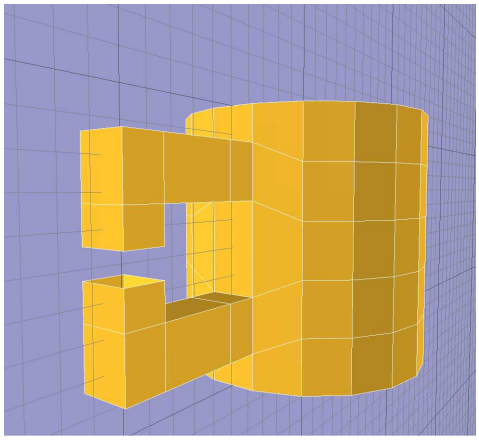
Obrázek 2.10: Krok 6.



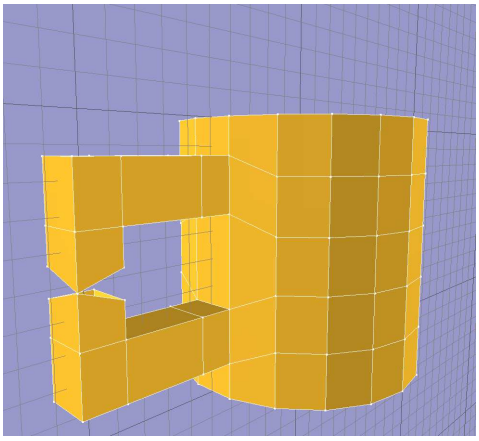
Obrázek 2.11: Krok 7.



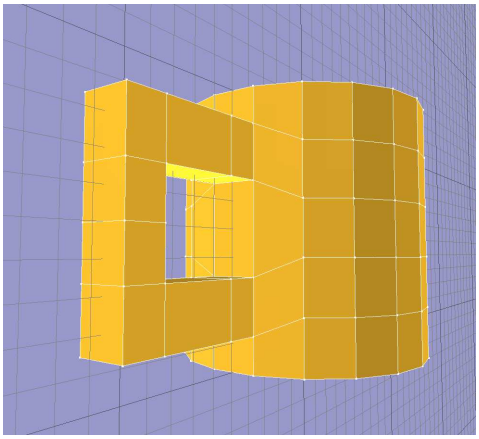
Obrázek 2.12: Krok 8.



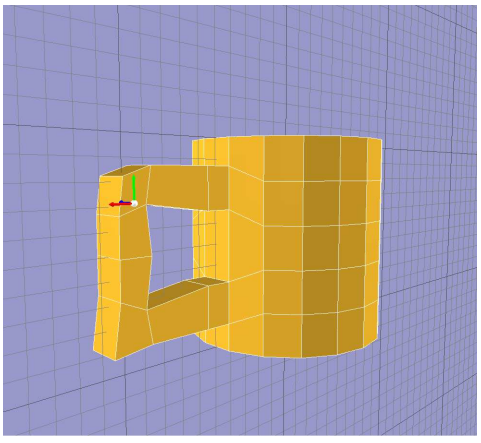
Obrázek 2.13: Krok 9.



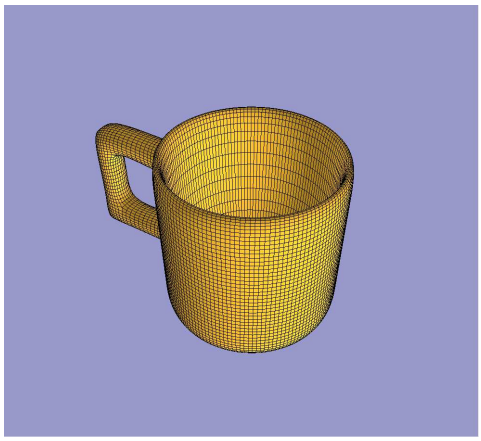
Obrázek 2.14: Krok 10.



Obrázek 2.15: Krok 11a.



Obrázek 2.16: Krok 11b.



Obrázek 2.17: Krok 12.

Kapitola 3

Skriptování a přidávání pluginů

Program Evolute využívá interpretu skriptovacího jazyka Python. S tímto získává i veškerou jeho sílu. Doporučuji si přečíst nějaký z manuálů k tomuto jazyku (např. [3]).

Není problém využít kteroukoliv pythonovskou knihovnu a to jak v konzoli, tak v prováděných skriptech a pluginech.

Pro práci s programem je vytvořena hierarchie specializovaných knihoven, většinou odpovídajících stejnojmenému modulu v programu.

3.1 Knihovny jazyka python

Evolute

`logErr( string )` - funkce vypisující chybové hlášky.

`_create_stdout()` - funkce vracející novou instanci objektu pro standartní výstup.

`_create_stderr()` - funkce vracející novou instanci objektu pro standartní chybový výstup.

Evolute.Geometry

`ModifierStack` - zásobník modifikátorů

- `GeomObject GetSource()` - vrátí zdrojovou geometrii

`GeomObject` - obecný objekt geometrie.

- `string GetParNames()` - vrátí názvy parametrů objektu.
- `value [string]/[string]=value` - operátor přístupu a nastavení parametrů.

`MeshGeomObject` - geometrický objekt polygonální sítě.

- `Mesh GetMesh()` - vrátí jeho polygonální síť.

`Modifier` - objekt modifikátoru.

- `GeomObject GetSource()` - vrátí objekt, na který byl aplikován.

`ModifierMesh` - modifikátor, jehož data jsou polygonální síť.

- `GetMesh()` - vrátí jeho polygonální síť.

Evolute.Geometry.Mesh

`Mesh NewMesh()` - vrátí nový objekt polygonové sítě.

`TriangulateMesh( Mesh )` - rozdělí všechny plošky zadné polygonální sítě s valencí větší než 3 na trojúhelníky pomocí úhlopříček.

`IsolateFaceList( FaceSequence, Mesh )` - "vykrojí zadané plošky z daného mnohostěnu. Zůstanou stále v geometrii, hrany na okraji jsou zdvojeny.

`ExtrudeFaceList( FaceSequence, Mesh, double amount )` - vysune zadané plošky ve směru průměrné normály o hodnotu `amount`.

`DeleteVertexList( VertexSequence, Mesh )` - odstraní vrcholy v seznamu, spolu s incidenčními subobjekty (hrany, plošky).

`DeleteEdgeList( EdgeSequence, Mesh )` - odstraní hrany v seznamu, spolu s incidenčními ploškami.

`DeleteFaceList( FaceSequence, Mesh )` - odstraní plošky v seznamu ze zadané polygonální sítě.

`ContractEdgeList( EdgeSequence, Mesh )` - zkontrahuje (nahradí jedním bodem) všechny hrany ze seznamu.

`CollapseVertexList( VertexSequence, Mesh )` - zadanou posloupnost vertexů spojí do jenoho jediného, který pak použije jako návratovou hodnotu.

`ScaleVertexList( Sequence, double x, double y, double z, double factor)` - provede změnu měřítka seznamu vertexů dle zadaného faktoru. Změna bude provedena relativně k zadaným souřadnicím v prostoru.

`Vertex InsertVertexToEdge( Edge, double, double, double, Mesh )` - vloží vrchol do předané hrany a na zadanou pozici. Tento vrchol pak použije jako návratovou hodnotu.

`Face SplitFace( Face, Vertex v1, Vertex v2, Mesh )` - rozdělí zadaný face na dva pomocí hrany spojující dva předané vertexy. Nově vytvořený face vrátí jako návratovou hodnotu.

Datové objekty s metodami:

`Vertex` - vrchol, základní prvek polygonálních sítí.

- `SetPos( double, double, double )` - nastaví pozici vrcholu na předané koordináty.
- `(double,double,double) GetPos()` - jako návratovou hodnotu použije trojici s koordináty vrcholu.
- `integer GetID()` - vrátí ID číslo vrcholu.

`Edge` - hrana, jeden ze stavebních kamenů mnohostěnu.

- `Vertex GetV1()` - vrátí počáteční vrchol.
- `Vertex GetV2()` - vrátí koncový vrchol.
- `Face GetF1()` - vrátí plošku nalevo od hrany.
- `Face GetF2()` - vrátí plošku npravo od hrany.
- `Edge GetSym()` - vrátí tuto hranu opačně orientovanou.
- `integer GetID()` - vrátí ID této hrany.

`Face` - ploška (polygon), stěna mnohostěnu.

- `RecalcNormal()` - znovu přepočítá normálu k této plošce.

`integer GetID()` - vrátí ID číslo této plošky.

`integer EdgeCount()` - vrátí počet hran tvořících tento polygon.

`Edge [integer]` - vrátí hranu se zadaným indexem tvořící tento polygon.

`Mesh` - objekt mnohostěnu.

`Vertex AddVertex( double, double, double )` - přidá do sítě nový vrchol o zadaných souřadnicích.

`Edge AddEdge( Vertex, Vertex )` - z dvou vrcholů sítě vytvoří novou hranu, přidá jí do sítě a použije jako návratovou hodnotu.

`Face AddFace( [Vertex, ...] )` - ze zadaného seznamu vrcholů vytvoří polygon, přidá ho do sítě a vrátí jako návratovou hodnotu.

`Vertex GetVertex( integer )` - získá vrchol ze sítě se zadaným pořadovým číslem.

`Vertex GetVertexID( integer )` - získá vrchol sítě se zadaným ID.

`Edge GetEdge( integer )` - získá hranu se zadaným pořadovým číslem.

`Edge GetEdgeID( integer )` - získá hranu se zadaným ID.

`Face GetFace( integer )` - získá polygon se zadaným pořadovým číslem.

`Face GetFaceID( integer )` - získá polygon se zadaným ID.

`integer VerticesCount()` - vrátí počet vrcholů v síti.

`integer EdgesCount()` - vrátí počet hran v síti.

`integer FacesCount()` - vrátí počet plošek v síti.

`MeshReset()` - vymaže obsažená data (vrcholy, hrany, polygony).

`RecalcNormals()` - přepočítá normálové vrcholy všech plošek.

`CloneToMesh( Mesh )` - překopíruje svoji geometrii do jiného objketu sítě.

Evolute.Geometry.SubSurf

`NewPrimalSubSurf ( Function even_vert, Function odd_edge, Function odd_face, integer max_valence, integer ref )` - vytvoří novou výpočetní strukturu pro primární subdivision schéma. První tři parametry jsou funkce repozicovacích pravidel (sudý vrchol, lichý hranový, lichý ploškový). Zadáním None místo některých z nich znamená ponechání defaultních pozic vrcholů daného typu. Tyto funkce musí přijímat dva parametry - první z nich je repozicovaný vrchol a druhý je pomocný objekt.

Parametr `max_valence` udává maximální povolenou valenci polygonů vstupní sítě (nevyhovující budou automaticky rozděleny). Poslední parametr je konstanta určující druh repozicovacího pravidla.

`NewPrimalSubSurfRefinement ( Function even_vert, Function odd_edge, Function odd_face, integer max_valence, Function ref )` - stejné jako předchozí fce, až na poslední parametr, kterým je v tomto případě dělící pravidlo

`SubdivisionVertex` - vrchol sítě v průběhu aplikace SubSurf.

`SetNPos( double, double, double )` - nastaví novou pozici vrcholu.

`(double, double, double) GetNPos()` - získá novou pozici vrcholu.

`(double, double, double) GetOPos()` - získá starou pozici vrcholu

`bool Crease()` - zjistí zda je vrchol součástí ostré hrany.

`VertexRepositioningHelper` - pomocný repozicovací objekt. Zná celou topologii sítě.

`( double, double, double ) GetNeighboursPosSum( SubdivisionVertex )` - vrátí součet pozic sousedních vrcholů zadaného vrcholu.

`( double, double, double ) GetOppositePosSum( SubdivisionVertex )` - vrátí součet pozic vrcholů ležících proti hraně, na kterou byl vložen zadný hranový vrchol.

`( double, double, double ) GetEvenOppositePosSum( SubdivisionVertex )` - vrátí součet pozic vrcholů, které leží v polygonech, do kterých náleží i zadaný sudý vrchol. Nezapočítává se vrchol sám a jeho sousedé.

`( double, double, double ) GetEdgeOppositePosSum ( SubdivisionVertex, SubdivisionVertex )` - vrátí součet pozic vrcholů naproti hraně zadané krajními vrcholy.

`[SubdivisionVertex, SubdivisionVertex] GetEdgeParents ( SubdivisionVertex )` - vrátí dvojici krajních vrcholů hrany, na kterou byl vložen zadaný vrchol.

`[SubdivisionVertex, ...] GetNeighbours( SubdivisionVertex )` - vrátí seznam sousedů zadaného vrcholu.

`[SubdivisionVertex, ...] GetOpposite( SubdivisionVertex )` - vrátí seznam vrcholů ležících proti hraně, na kterou byl vložen zadný hranový vrchol.

`[SubdivisionVertex, ...] GetEvenOpposite( SubdivisionVertex )` - vrátí seznam vrcholů, které leží v polygonech, do kterých náleží i zadaný sudý vrchol. Nezapočítává se vrchol sám a jeho sousedé.

`[SubdivisionVertex, ...] GetEdgeOpposite ( SubdivisionVertex, SubdivisionVertex )` - vrátí seznam vrcholů naproti hraně určené zadanými vrcholy.

`integer GetCreasesCount( SubdivisionVertex )` - zjistí počet vrcholů, do kterých vede ostrá hrana.

`[SubdivisionVertex, ...] GetCreases( SubdivisionVertex )` - vrátí seznam vrcholů, do kterých vede ostrá hrana ze zadaného vrcholu.

`PrimalSubdivisionBuilder` - výpočetní struktura pro primární subdivision schémata.

`Reset()` - vymaže data v ní uložená.

`Iterate( integer )` - nechá proběhnout zadaný počet iterací.

`SetSourceMesh( Mesh )` - zadá vstupní síť.

`CreateMesh( Mesh )` - do zadané sítě vyplní data získáná aplikací SubSurf na vstupní síť.

Evolute.Kernel

`Update` - aktualizuje všechny moduly.

`EventHandler NewSelWinHandler()` - vytvoří pomocný objekt ošetřující události, jež jsou spojené se selekčním oknem (nastavování jeho velikosti apod.).

`EventHandler NewHitTestHandler()` - vytvoří pomocný objekt ošetřující události, které jsou vyvolávány při klepnutí myší na objekty.

`EventHandler NewPivotManipulatorHandler()` - vytvoří pomocný objekt ošetřující události vyvolávající změny v aktuálním manipulátoru.

`EventHandler` - rozhraní objektů, které dokáží ošetřovat události.

`bool HandleEvent( Event )` - metoda, jež zkusí ošetřit předanou událost. Pokud se jí to povede vrátí True.

`Event` - Bázová třída událostí.

`Dispose()` - zničí záznam o události.

`integer Type()` - vrátí typ události.

`MouseDownEvent` - událost vyvolaná stisknutím tlačítka myši.

`Button()` - tlačítko, které bylo stisknuto.

`MouseUpEvent` - událost vyvolaná uvolněním tlačítka myši.

`Button()` - tlačítko, které bylo uvolněno.

`MouseMoveEvent` - událost vyvolaná pohybem myši.

`MouseWheelEvent` - událost vyvolaná kolečkem myši.

`ObjectHitEvent` - událost vyvolaná klepnutím na nějaký objekt.

`integer HitID()` - vrátí ID objektu, na který bylo kliknuto.

`bool IsAdd()` - zjistí zda bylo žádáno o přidání objektu do selekce.

`IsSwitch()` - zjistí zda bylo žádáno o přepnutí výběrového stavu objektu (pokud byl vybrán, bude jeho výběr zrušen, jinak bude přidán do selekce).

`GroupHitEvent` - událost vyvolaná selekcí většího množství objektů.

`integer IsAdd()` - zda mají být přidány do selekce.

`IsSubtraction()` - zda mají být vyjmuty ze selekce.

`length()/integer [integer]` - operátor zjišťující délku seznamu vybraných objektů a operátor přístupu k ID číslům vybraných objektů.

Konstanty typů událostí : `VIEWPORT_MOUSE_DOWN_TYPE`

`VIEWPORT_MOUSE_UP_TYPE`

`VIEWPORT_MOUSE_MOVE_TYPE`

`VIEWPORT_MOUSE_WHEEL_TYPE`

`KEYBOARD_DOWN_TYPE`

`KEYBOARD_UP_TYPE`

`OBJECT_HIT_EVENT_TYPE`

`GROUP_HIT_EVENT_TYPE`

Evolute.Modeler

`BaseObject CreateObj( string )` - vytvoří nový objekt scény zadaného typu.

`BaseObject CreateObjFromMesh( Mesh )` - ze zadané polygonální sítě vytvoří nový geometrický objekt scény.

`string GetCreatorsTypes()` - vrátí jména typů objektů scény, které lze vytvořit.

`ApplyModifier( GeometryContainer, string )` - aplikuje modifikátor zadaného jména na předaný objekt.

`AddCreatorMod( string, CREATION_PLUGIN, integer )` - přidá nový geometrický plugin pro tvorbu zdrojové geometrie. První parametr je název, druhý plugin samotný a třetí je konstanta odpovídající typu geometrie (zatím může být pouze `MESH`).

`AddModifierMod( string, MODIFIER_PLUGIN, integer, integer )` - přidá nový modifikátor. První parametr je název, druhý plugin samotný, třetí je konstanta typu geometrie (zatím může být pouze `MESH`) a čtvrtý je konstanta typu geometrie na který ho lze aplikovat.

Evolute.Objects

`Pivot CreatePivot()` - vytvoří pivot.

`Pivot CreateReportingPivot( Function )` - vytvoří pivot, který při změně své pozice, nebo natočení zavolá předanou funkci.

`Pivot SetPos( double, double, double )` - nastaví pozici.

`SetRot( double, double, double )` - nastaví natočení.

`( double, double, double ) GetPos()` - zjistí pozici.

`( double, double, double )GetRot()` - zjistí natočení.

`BaseObject SetPos( double, double, double )` - nastaví pozici.

`SetRot( double, double, double )` - nastaví natočení.

`Move( double, double, double )` - posune objekt o zadaný vektor.

`Rotate( double, double, double )` - orotuje objekt o zdané úhly.

`Orbit( double, double, double, double, double, double )` - orotuje objekt okolo zadaného bodu. První tři douřadnice určují střed rotace, zbylé určují úhly.

`( double, double, double ) GetPos()` - zjistí pozici.

`( double, double, double )GetRot()` - zjistí natočení.

`GeometryContainer GeometryStack GetStack()` - vrátí zásobník modifikátorů.

`Rebuild()` - přepočítá celou geometrii (všechny modifikátory).

Evolute.Plugin

`ParamList NewParamSet()` - vytvoří nový seznam parametrů.

`ParamList` - objekt zastřešující seznam parametrů. Každý z nich je jednoznačně identifikován textovým řetězcem.

`AddDouble( string name, double value, double min, double max )` - přidá floating-point parametr s rozsahem.

`AddInteger( string name, integer value, integer min, integer max )` - přidá celočísleny parametr s rozsahem.

`AddBoolean( string name, bool value )` - přidá boolean parametr.

`string GetParNames()` - vrátí jména parametrů

`value [string]/[string] = value` - přístup k jednotlivým parametrů - zjišťování i nastavování hodnot.

`GuiCreator` - pomocný objekt poskytující metody pro tvorbu uživatelských rozhraní pluginů.

`NewGroup( string )` - založí novou skupinu komponent.

`AddParamList( ParamList )` - vytvoří komponenty pro nastavování parametrů ze zadaného seznamu.

`AddButton( string, Function )` - přidá tlačítko se zadným názvem a spouštějící zadanou funkci.

`AddLineEditInt ( string name, Function(integer) onchange, Function update, double min, double max)` - přidá komponentu pro nastavování neceločíselných hodnot. Funkce `onchange` je spouštěna při změně hodnoty (předá jí do funkce jako parametr). Funkce `update` vrací hodnotu, na kterou má být komponenta nastavena v případě aktualizace. Poslední dva parametry určují rozsah povolených hodnot.

`AddLineEditDouble (string name, Function(integer) onchange, Function update, integer min, integer max)` - jako předchozí metoda, jen pro celočíselné hodnoty.

`AddButtonLineEditInt (string name, Function onclick, Function onchange, Function update, integer min, integer max)` - jako předchozí metody. Pro celočíselné hodnoty navíc ve skupině s tlačítkem.

`AddButtonLineEditDouble (string name, Function onclick, Function onchange, Function update, double min, double max)` - jako předchozí metoda, pro neceločíselné hodnoty.

`ReportChange()` - metoda, kterou může zavolat plugin, když chce dát vědět o změně svých dat..

`UpdateInterface()` - aktualizuje GUI.

Evolute.Selection

`BaseObject GetSelected()` - vrátí vybraný objekt.

`SetSelected( BaseObject )` - daný objekt nastaví jako selektovaný.

`RemoveSelected()` - smaže vybraný objekt ze scény.

`Deselect()` - zruší výběr objektu scény

`AssignManipulator( Pivot, bool )` - nastaví zadaný pivot na aktuální manipulátor, kterým lze pohybovat pomocí myši. Pokud je druhý parametr True pohybuje se v absolutních souřadnicích, jinak relativně k aktuálně vybranému objektu.

Evolute.FileWorker

`AddLoadingPlugin( Function, string )` - přidá nový plugin pro načítání souborů identifikovaný jednoznačným názvem. Jako parametr bere funkci, která bere za parametr cestu k souboru a vrací seznam objektů scény.

`AddSavingPlugin( Function, string )` - přidá nový plugin ukládající objekty do souboru. Prvním parametrem je funkce, jež dokáže do cestou zadaného souboru uložit seznam objektů. Druhý parametr je řetězec jednoznačně identifikující plugin.

`LoadFileType( string, string )` - Nahraje scénu ze souboru zadaného prvním parametrem. Druhý parametr určuje plugin, který se má použít k načítání.

`ImportFileType( string, string )` - Importuje ze souboru, jež zadaný prvním parametrem, objekty do scény. Druhý parametr určuje plugin, který se má použít k načítání.

Evolute.GUI

`Update()` - aktualizuje GUI.

`AddMenuItem( string, Function )` - přidá do hlavní menu položku, jež je zadána cestou. Při jejím výběru se spustí zadaná funkce.

`ChooseFileDlg( integer )` - zobrazí dialogové okno pro výběr souboru. Celočíselný parametr určuje, zda se jedná o ukládací dialog nebo otevírací. Po zavření okna v případě úspěchu vrátí absolutní cestu k souboru, jinak vrátí `None`.

Evolute.Renderer

`Renderer` - objekt rendereru, který je předáván do funkcí při vykreslovaní.

`RenderMesh( Mesh )` - vykreslí celou síť.

`RenderVertices( Mesh )` - vykreslí vrcholy sítě.

`RenderVerticesID( Mesh )` - vykreslí vrcholy sítě. Zároveň bude nastavovat jejich ID pro kreslení do select bufferu.

`RenderVertex( Vertex )` - vykreslí vrchol.

`RenderVertexID( Vertex )` - vykreslí vrchol a nastaví jeho ID do select bufferu.

`RenderEdges( Mesh )` - vykreslí hrany sítě.

`RenderEdgesID( Mesh )` - vykreslí hrany sítě. Zároveň bude nastavovat jejich ID pro kreslení do select bufferu.

`RenderFaces( Mesh )` - vykreslí polygony sítě.

`RenderFacesID( Mesh )` - vykreslí polygony sítě. Zároveň bude nastavovat jejich ID pro kreslení do select bufferu.

`RenderEdge( Edge)` - vykreslí hranu.

`RenderFace( Face )` - vykreslí zadaný polygon.

`RenderPivotRelatively( Pivot )` - vykreslí manipulátor.

`SetColorTemp( integer, integer, integer )` - nastavi barvu (RGB), která se použije při kreslení.

Evolute.Scene

`string GetInfo` - vrátí nějaké informace o scéně.

`Reset()` - vymaže všechny objekty ze scény.

`AddObject( BaseObject )` - přidá objekt do scény.

GeometryPlugin, MeshFunctions, MeshFunctions

Tyto knihovny jsou psány přímo v Pythonu. Províce informací o nich použijte dokumentační řetězce ve zdrojových souborech.

3.2 Jednoduchý geometrický plugin

Zde ukážu základy, jak napsat plugin vytvářející kvádr. Nezapomeňte, že kód je psán v jazyce Python, kde je důležité odsazení řádků.

Na začátku si naimportujeme potřebné knihovny funkcí.

```
from GeometryPlugin import *
from Evolute import Modeler
from Evolute import Plugin
```

Náš plugin bude vytvářet zdrojovou geometrii, proto musí dědit ze třídy `CREATION_PLUGIN`, která je uvedena v knihovně `GeometryPlugin`.

```
class CUBE_PLUGIN( CREATION_PLUGIN ):
```

V konstruktoru našeho pluginu musíme vytvořit záznamy v seznamu parametrů. Na ty se můžeme při konstrukci objektu dotazovat a mohou být zobrazovány uživatelským rozhraním.

```
def __init__( self ):
    CREATION_PLUGIN.__init__(self)
    self.Parameters.AddDouble( "WIDTH", 10.0, 0.0, 100000.0 )
    self.Parameters.AddDouble( "HEIGHT", 10.0, 0.0, 100000.0 )
    self.Parameters.AddDouble( "LENGTH", 10.0, 0.0, 100000.0 )
```

V této metodě dochází k tvorbě geometrie objektu, který je předán jako parametr `obj`.

```
def Build( self, obj ):
```

Nejdříve se dotážu jakou hodnotu mají jednotlivé parametry objektu nutné k vytvoření geometrie.

```
    sizeA = obj[ "WIDTH"  ] / 2.0
    sizeB = obj[ "HEIGHT" ] / 2.0
    sizeC = obj[ "LENGTH" ] / 2.0
```

Získám odkaz na objekt polygonální sítě zadaného objektu a smažu v něm uložená data.

```
    m = obj.GetMesh()
    m.MeshReset()
```

Zde postupně vytvoříme všech osm vrcholů kvádru na pozicích zjištěných z rozměrů kvádru. Zároveň si ukládáme dané vrcholy do proměných.

```
    v0 = m.AddVertex( sizeA, sizeB, sizeC )
    v1 = m.AddVertex( sizeA, -sizeB, sizeC )
    v2 = m.AddVertex( -sizeA, -sizeB, sizeC )
    v3 = m.AddVertex( -sizeA, sizeB, sizeC )
    v4 = m.AddVertex( sizeA, sizeB, -sizeC )
    v5 = m.AddVertex( -sizeA, sizeB, -sizeC )
    v6 = m.AddVertex( -sizeA, -sizeB, -sizeC )
    v7 = m.AddVertex( sizeA, -sizeB, -sizeC )
```

Posledním krokem tvorby geometrie je vytvoření šesti stěn. K tomu potřebujeme již vytvořené vrcholy, jimiž definujeme jednotlivé stěny. Všechny stěny musí být souhlasně orientovány (závisí na pořadí vrcholů). V opačném případě dostaneme nežádoucí výsledek.

```
    m.AddFace( [ v0, v3, v2, v1 ] )
    m.AddFace( [ v7, v6, v5, v4 ] )
    m.AddFace( [ v1, v7, v4, v0 ] )
    m.AddFace( [ v4, v5, v3, v0 ] )
    m.AddFace( [ v6, v2, v3, v5 ] )
    m.AddFace( [ v1, v2, v6, v7 ] )
```

Tímto jsme ukočili definici nového pluginu. Nyní ho předáme modulu `Modeler` spolu s jeho názvem a typem jeho geometrie.

```
Modeler.AddCreatorMod( "SimpleCube", CUBE_PLUGIN, Modeler.MESH )
```

3.3 Geometrický plugin - nové subsurf schéma

Nyní ukážu jak vytvořit jednoduché subdivision schéma.

Jako v předchozím příkladu nejdříve naimportuji potřebné knihovny.

```
from Evolute import Modeler
from Evolute import Plugin
from Evolute.Geometry.Mesh import *
from Evolute.Geometry.SubSurf import *
from GeometryPlugin import *
```

Nadefinujeme repozicovací pravidla pro sudé vrcholy a hranové vrcholy (stejným způsobem se pro jiná schémata definují pravidla pro ploškové vrcholy)

```
def even_vertex_rule( vertex, helper ):
    pos = vertex.GetOPos()
    if( vertex.Crease() ):
        creases = helper.GetCreases( vertex )
        res = []
        pos1 = creases[0].GetOPos()
        pos2 = creases[1].GetOPos()
        for i in range(3):
            res.append( pos[i]*3.0/4.0 + 1.0/8.0 *(pos1[i]+pos2[i]) )
        vertex.SetNPos( res[0], res[1], res[2] )
        return


    pom1 = helper.GetNeighboursPosSum( vertex )
    pom2 = helper.GetEvenOppositePosSum( vertex )
    res = []
    beta = 2.0/8.0
    gamma = 1.0/8.0
    if( pom2[3] == 0 ):
        beta = beta + gamma
        for i in range(3):
            res.append( pos[i]*( 1-beta ) + pom1[i]*beta/pom1[3] )
    else:
        for i in range(3):
            res.append( pos[i]*( 1-beta-gamma ) + pom1[i]*beta/pom1[3] + pom2[i]*gamma/
                pom2[3] )

    vertex.SetNPos( res[0], res[1], res[2] )

def odd_edge_vertex_rule( vertex, helper ):
    par = helper.GetEdgeParents( vertex )
    pos1 = par[0].GetOPos()
    pos2 = par[1].GetOPos()

    if( vertex.Crease() ):
        res = []
        for i in range(3):
            res.append( 1.0/2.0*( pos1[i]+pos2[i] ) )
        vertex.SetNPos( res[0], res[1], res[2] )
        return

    pom = helper.GetOppositePosSum( vertex )
    res = []
    b = 1.0/8.0 * 2.0/pom[3]
    for i in range(3):
        res.append( 3.0/8.0 * ( pos1[i]+pos2[i] ) + b * pom[i] )

    vertex.SetNPos( res[0], res[1], res[2] )
```

Inicializace jako v předchozím příkladu. Navíc musíme vytvořit datovou strukturu pro výpočet subdivision surfaces - té předáme námi vytvořená pravidla (pro ploškové None - nebudeme ho potřebovat). Dále jí předáme maximální valenci polygonů, tím zajistíme že nám struktura ty nevyhovující upraví na správnou hodnotu. Posledním parametrem si vybereme dělící pravidlo - v tomto případě budeme dělit čtyřúhleníky na čtyřúhelníky.

```
class CATMULL_CLARK( MODIFIER_PLUGIN ):
    builder = None

    def __init__( self ):
        MODIFIER_PLUGIN.__init__(self)
        self.Parameters.AddInteger( "DEPTH",    1, 1, 20 )
        self.builder = NewPrimalSubSurf( even_vertex_rule, odd_edge_vertex_rule,
            None, 4, PQQ )
```

Začátek konstrukční metody je podobný minulému příkladu.

```
    def Build( self, obj ):
```

```
        d = obj[ "DEPTH"   ]

        m = obj.GetMesh()
        m.MeshReset()
        o = obj.GetSource()
```

Ve zbytku však pracujeme se strukturou pro SubSurf. Vynulujeme ji, nastavíme vstupní síť a necháme proběhnout požadovaný počet iterací. Na závěr vytvoříme výstupní síť.

```
        self.builder.Reset()
        self.builder.SetSourceMesh( o.GetMesh() )
        self.builder.Iterate( d )
        self.builder.CreateMesh( m )
```

Ještě nesmíme zapomenout přidat nový plugin do programu. Nyní přidáváme modifikátor, proto musíme zadat nejen typ geometrie, ale i typ geometrie, na který lze tento modifikátor aplikovat.

```
Modeler.AddModifierMod( "CatmullClark", CATMULL_CLARK, Modeler.MESH, Modeler.MESH )
```

3.4 Složitější geometrický plugin

Vytvořit základní pluginy již umíme. Nyní si ukážeme některá rozšíření, která můžeme využít při psaní složitějších pluginů. Zde zobrazené části kódu jsou z modifikátoru *Edit mesh*, který může pracovat ve třech různých módech editace.

Nejdříve jsou vytvořeny objekty usnadňující ošetřování událostí. Každý z nich má svou specializaci.

```
SelWinHandler   = Kernel.NewSelWinHandler()
HitTestHandler      = Kernel.NewHitTestHandler()
PivotManipHandler = Kernel.NewPivotManipulatorHandler()
```

Nadefinujeme si konstanty identifikující jednotlivé módy editace. A nadeklarujeme třídu modifikátoru.

```
ID_VERTICES = 1
ID_EDGES    = 2
ID_FACES    = 3

class EDIT_MESH_MOD( MODIFIER_PLUGIN ):
    mesh = None
    selection = []
    vertices = []
    pivot = None
    creator = None

    ExtrudeAmount = 0.0
    ScaleFactor = 1.0
    isPivotDrawing = False  #Zda se ma vykreslovat Pivot
    isPivotHandling = False #Zda se ma vyuzit zpravy o zmene parametru pivotu
    oldPosition = ( 0.0, 0.0, 0.0 )
          .
          .
          .
```

Metoda, kterou musíme vytvořit, pokud má modifikátor více módů editace. Vrátí seznam dvojic `(název, ID)`.

```
    def GetModNames( self ):
       return [ ( "Vertices", ID_VERTICES ), ( "Edges", ID_EDGES ), ( "Faces",
           ID_FACES ) ]
          .
          .
          .
```

Metoda nutná pokud potřebujeme v módu editace využívat select buffer nebo si vykreslovat objekt po svém.

```
def Render( self, renderer, selection ):
    if selection:
        self.SelectRender( renderer )
    else:
        self.ViewRender( renderer )
    #if self.isPivotDrawing:
    #  renderer.RenderPivotRelatively( self.pivot )
```

Pokud si modifikátor nevystačí jen se seznamem parametrů (pro nějž bylo GUI vytvářeno automaticky již v rodičovské třídě) může si vytvořit GUI podle svého. Slouží k tomu předávaný pomocný objekt `creator`. Ten poskytuje rozhraní pro tvorbu základních komponent.

Jak je vidět GUI může být závislé na módu editace.

```
def CreateGUI( self, creator, mod ):
    #Selection.AssignManipulator( self.pivot )
    #print "Creating GUI mod: " + str(mod)
    self.ActualMod = mod
    self.creator = creator
    self.selection = []
    self.vertices = []
    self.UpdatePivot()
    if mod != 0:
        creator.NewGroup( "Basic" )
        creator.AddButton( "Delete selection", self.DeleteSelection )
        creator.AddButton( "Recalculate normals", self.RecalcNormals )
        creator.AddButtonLineEditDouble( "Scale", self.Scale, self.SetScaleFactor,
            self.GetScaleFactor, -10000.0, 10000.0 )

    if self.ActualMod == ID_FACES :
        creator.NewGroup( "Extruding" )
        creator.AddButtonLineEditDouble( "Extrude", self.ExtrudeFaces, self.
            SetExtrudeAmount, self.GetExtrudeAmount, -10000.0, 10000.0 )
        creator.AddButton( "Isolate", self.IsolateFaces )
    elif self.ActualMod == ID_EDGES :
        creator.AddButton( "Contract edges", self.ContractEdges )
    elif self.ActualMod == ID_VERTICES :
        creator.AddButton( "Collapse vertices", self.CollapseVertices )
        .
        .
        .
```

Důležitá metoda ošetřující události posílané modifikátoru v některém z módů editace. Zde jsou ošetřovány požadavky na vytváření výběrových oken, selekcí objektů či změny manipulátorů. Metoda vrátí True pokud ošetřila událost (v tom případě je povinna zrušit záznam o události voláním `Dispose()`]"a False pokud nikoliv.

```
def HandleEvent( self, event ):
    if PivotManipHandler.HandleEvent( event ) :
        return True
    if SelWinHandler.HandleEvent( event ) :
        return True
    if HitTestHandler.HandleEvent( event ):
        return True
    if (event.Type() == Kernel.OBJECT_HIT_EVENT_TYPE):
        self.ObjectHitEvent( event )
        self.UpdatePivot()
        event.Dispose()
        return True
    if (event.Type() == Kernel.GROUP_HIT_EVENT_TYPE):
        self.GroupHitEvent( event )
        event.Dispose()
        return True

    return False
```

Část II

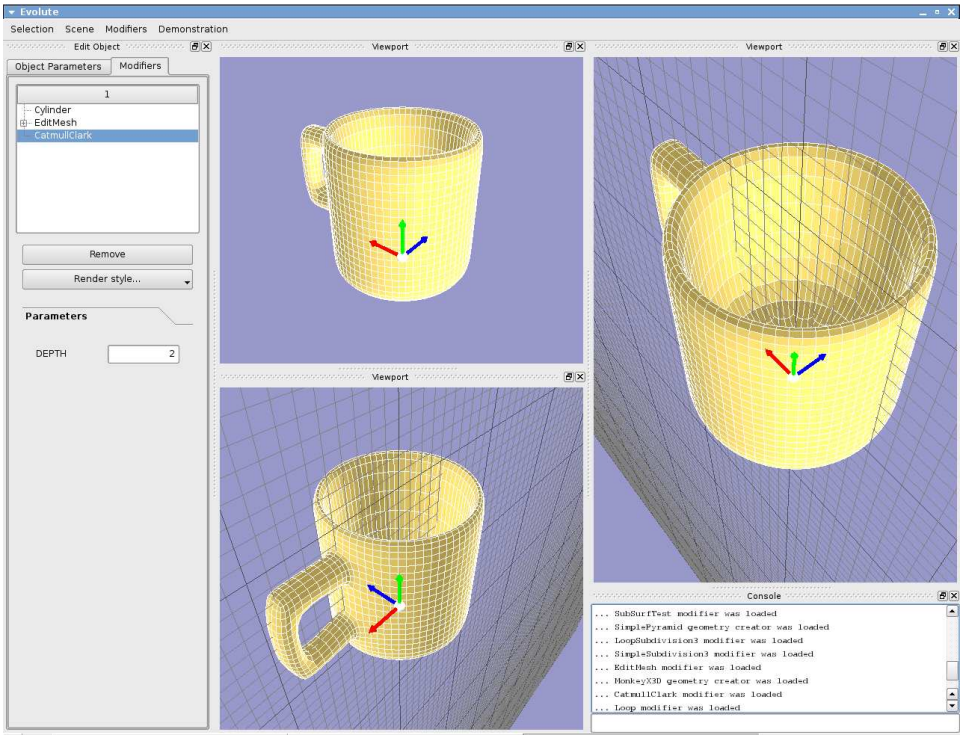
Programátorská dokumentace

Kapitola 4

Požadovaná funkčnost

Cílem bylo vytvořit program, umožňující vytváření a testování subdivision schémat. Rozhodl jsem se přistoupit k úkolu podobně, jako k návrhu obecného 3D editoru, ve kterém by byly požadované vlastnosti zastoupeny jako vysoce specializované nástroje. Ostatní funkce, které nemají s tématem přímou spojitost, buď nebudou v případě nedostatku času realizovány, nebo jen částečně implementovány.

Ve výsledku bych tedy měl získat základ programu, který je silným nástrojem podporující subdivision surfaces a je jednoduše rozšiřitelný o další vlastnosti, funkce, či nové okruhy využití.



Obrázek 4.1: Program Evolute - hrníček vymodelován z válce a vyhlazen pomocí Catmullova-Clarkova SubSurf schématu.

SubSurf pracují s polygonálními sítěmi jako vstupy i výstupy. Z tohoto důvodu je editor primárně zaměřen na práci s polygony. Návrh programových částí, které pracují s geometrií, je však připraven na pozdější rozšíření o další druhy geometrických dat (NURBS plochy, křivky, apod.).

SubSurf schémata mají často procedurální charakter, z toho tedy vyplynulo využití skriptovacího jazyka pro jejich popis. Výběr padl na jazyk Python, který je silným vysokoúrovňovým programovacím jazykem a jeho intepret lze bez velkých obtíží vložit do programu a rozšířovat jeho množinu built-in funkcí pomocí c/c++. To také zapříčinilo jeho hlubší začlenění a využitelnost ve většině částí programu, které lze z jeho rozhraní ovládat, případně rozšiřovat.

Editační a modelovací systém je navržen velmi obecně a subdivision surfaces do něj zapadají jen jako specializovaná množina funkcí. Tím nijak nelimitují zbytek programu, který je využitelný i pro jiný přístup k modelování. Stačí program jen odpovídajícím způsobem rozšířit.

Kapitola 5

Architektura programu

5.1 Modulární návrh

Po zkušenostech z jiného projektu, jsem se rozhodl rozvrhnout program do samostatných částí (modulů). Mimo ně neexistují žádná data, všechna jsou uložena v nějakém z modulů.

Toto rozdělení programu vyžaduje sice napsání velkého množství rutin, které budou zaručovat bezproblémový běh a vzájemnou komunikaci mezi částmi programu. Výhod tohoto schématu je však nepřeberné množství.

V prvé řadě je to vysoká rozšiřitelnost programu. Přidání nové množiny funkcí můžeme zajistit velice jednoduše rozšířením programu o nový modul, který je zastřešuje. Například budeme chtít program využít pro spolupráci s nějakým snímacím zařízením, nebo jinou aplikací pro práci s 3D objekty. Vytvoříme tedy nový modul schopný komunikace s cílovým zařízením, nebo aplikací. Tento modul pak vložíme do programu a máme de facto novou specializovanou aplikaci.

Pokud se budeme poctivě držet modularity a každý z modulů bude zpřístupňovat jen určitou specializovanou oblast. Získáme velice transparentní návrh, kde modifikace již hotových částí programu není problém. Já osobně jsem v průběhu vývoje byl nucen změnit GUI knihovnu, kterou jsem používal. Změna se prakticky dotkla jen jednoho modulu. Neměnil jsem funkční rozhraní a jen jsem přepsal těla metod, aby využívaly novou GUI knihovnu.

V neposlední řadě nám modulární návrh zjednodušuje (velmi výrazně) údržbu zdrojového kódu. Ladění programu je také velice snadné. Většina chyb je nalezena ve velice krátkém čase. Zůstávají uzavřeny ve svém modulu a pokud nějaký jiný modul dostane chybná data, je velice snadné najít jejich zdroj. To jsou mé zkušenosti z vývoje programu - neřešil jsem žádný závažný, nebo těžko nalezitelný problém.

5.2 Komunikace a propojení mezi moduly

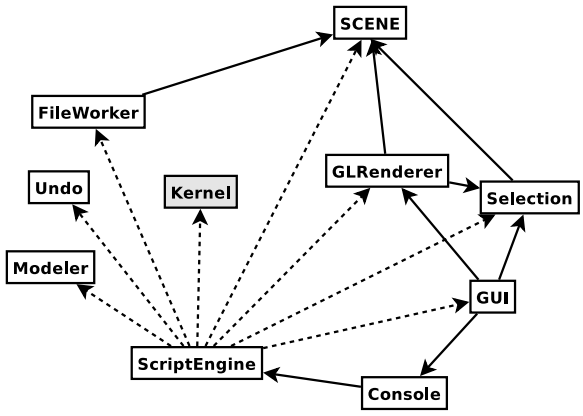
Komunikaci mezi dvěma různými moduly lze zajistit:

Přímé spojení je nejjednodušší na použití. Modul, který požaduje toto spojení musí znát rozhraní cílového, jelikož jediné co dostane jako odpověd od `Kernelu` na požadavek propojení (`get_module_by_type()`) je ukazatel na cílový modul, jehož typ byl zadán parametrem.

Takovéto spojení je většinou navazováno při inicializaci, kdy se `Kernel` dotazuje na požadavky modulu (např. modul `Console` se napojí na `ScriptEngine` apod.).

Nepřímé spojení je navazováno vždy přes `Kernel`. První možností je volání nějaké metody `Kernelu`, zprostředkovávající volání na modulech (např. metoda `Update()`).

Dále lze poslat zprávu o nějaké události, kterou je třeba ošetřit (kliknutí myši, změna módu editace,...), voláním `handle_event()`. Zprávou je ukazatel na datový objekt dědící z třídy `EV_EVENT`. Ty zpravidla nemají adresáta. O jejich příjemci rozhoduje jejich typ a stav, v jakém se nachází aplikace (viz. 5.3).



Obrázek 5.1: Vazby mezi moduly programu

V diagramu je znázorněno, jak jsou jednotlivé moduly standartně provázány.

5.3 Modul `KERNEL`

Jak již bylo zmiňováno, `Kernel` obstarává veškerou manipulaci s programovými moduly. Je prvním modulem, který je vytvářen při startu programu. Sám pak inicializuje ostatní programové moduly.

Inicializace

Inicializace probíhá v několika fázích. Nejdříve je vytvořena instance modulu Kernel a na ní je pak volána metoda `initialize()`.

Nejdříve jsou vytvořeny instance všech zaregistrovaných modulů. Pokud vše proběhne bez problémů, jsou pak postupně všechny moduly napojeny na modul Kernel a je zavolána jejich inicializační metoda.

Posledním krokem je dotázání se všech modulů na jejich požadavky (`send_requests_for_kernel()`). Zde většinou probíhá navazování přímých spojení mezi moduly.

V tuto chvíli je vše připraveno k použití, proto se zavolá metoda `enable_kernel()` a spustí se inicializační skript, který nahraje pluginy a uživatelské nastavení.

Stavy aplikace

Během práce s programem se celá aplikace nachází v nějakém ze čtyř stavů[1]. Podle nich se rozhoduje o aktuálním vzhledu GUI, způsobu renderování objektů scény a celkovém chování aplikace při interakci s uživatelem.

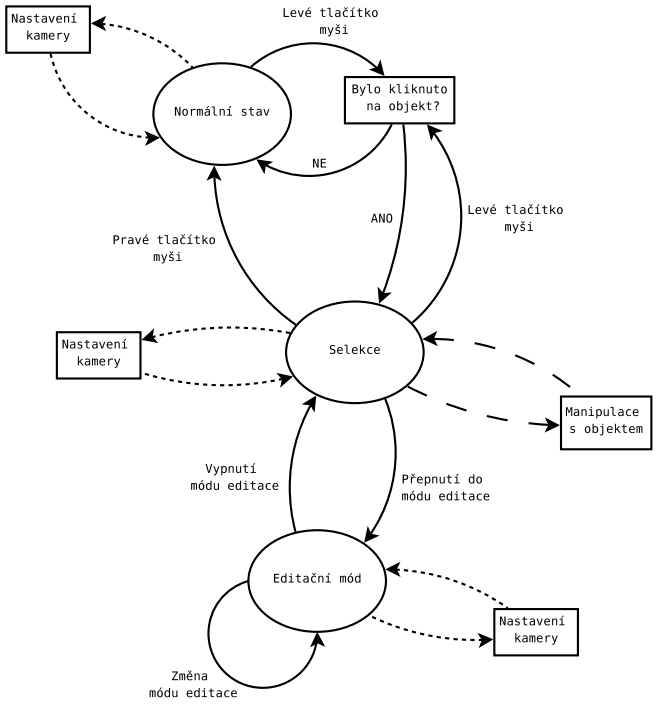
Normální (`AS_DEFAULT`) - jsou ošetřovány události spojené s nastavováním viewportů. GUI modul *Edit object* je neaktivní. Kliknutím na objekt scény dojde k jeho výběru a změně stavu aplikace.

Selekce (`AS_SELECTED`) - stav, ve kterém je vybrán jeden objekt scény. Jsou ošetřovány události viewportů, lze používat myš při práci s *manipulátorem* vybraného objektu. Lze používat GUI modul *Edit object*.

Skupinová (`AS_GROUP_SELECTED`) - v současnosti stav, do kterého se nelze přepnout. Je cílený pro správu skupinové selekce objektů scény (hromadné transformace apod.).

Speciální mód editace (`AS_EDIT_MODE`) - výběrem editačního módu nějakého modifikátoru se aplikace přepne do tohoto stavu. Chování programu pak závisí na konkrétním pluginu.

[1] V současnosti pouze ve třech - skupinová selekce objektů zatím není dostupná



Obrázek 5.2: Stavy programu a přepínání mezi nimi

Ošetřování událostí

Pro ošetření události používá Kernel členského objektu typu `STATE_MACHINE`, ten potom vše obstará sám.

Získané informace (`EV_EVENT`) se delegují objektům, které implementují rozhraní `EVENT_HANDLER_IFC`. Ty pokud zjistí, že danou událost dokážou zpracovat, tak učiní a zavolají destruktor záznamu o události a vrátí true, v opačném případě nedělají nic a vrátí false.

Událost je postupně předávána jednotlivým uvhandlerům, dokud se nenajde nějaký, který ji dokáže ošetřit (pokud žádný takový není - událost je ignorována a záznam zdestruován).

První handler v pořadí zajišťuje zpracování požadavků na změny nastavení kamery (poloha, natočení, přiblížení) patřící viewportu uvedeném v záznamu o události. Druhý zpracovává systémové (ovlivňující běh programu - např. změna stavu aplikace) události.

Výběr dalšího již závisí na aktuálním stavu aplikace.

Normální - ošetřují se události vztažené k selekci objektů scény.

Selekce - selekce jiného objektu, práce s manipulátorem vybraného objektu.

Speciální mód editace - ošetřování událostí je předáno pluginu, jehož mód editace je aktivní. V režii Kernelu zůstávají jen systémové události. Více viz. 7.2.

5.4 Modul `MODELER`

Modeler je pomocí pluginů rozšiřitelný modul vytvářející objekty scény (více o nich viz. 6). Pluginy prozatím rozšiřují jen množinu geometrických objektů, a to jak zdrojových geometrií, tak modifikátorů. Pokud by bylo nutné přidat nový negeometrický objekt, musí se tak učinit přímo ve zdrojovém kódu, nejlépe v inicializační metodě modulu.

Vytvoření nového objektu scény probíhá zavoláním metody `create_obj_by_typename()`. Ta zjistí, zda pro zadaný typ existuje objekt typu `CREATOR_MOD`, jež dokáže vytvořit žádaný objekt

scény. Pokud ano, vytvoří jej a odešle jako návratovou hodnotu. Vyhledává se podle unikátního názvu.

V případě, že máme k dispozici samotnou geometrii polygonální sítě (`MESH_DATA`), lze vytvořit objekt scény přímo z ní metodou `create_obj_from_mesh()`. Ta vrátí objekt typu `GEOMETRY_CONTAINER`, se zadanou zdrojovou geometrií. Lze na něj tedy dále aplikovat modifikátory.

Aplikace modifikátoru probíhá obdobně jako vytváření nového objektu. Až na to, že se podle názvu hledá objekt s rozhraním `MODIFIFICATION_APPLICATOR`. Když je nalezen, je ověřeno zda souhlasí typ geometrie, na níž má být aplikován, a typ modifikátoru. Pokud dojde k nějakému problému neděje se nic a modifikátor není aplikován.

Pluginy se dodávají ve formě objektů implementujících rozhraní `GEOMETRY_PLUGIN_FACTORY`. Pluginy pro zdrojové geometrie i pro modifikátory mají sice stejné rozhraní, je však nutné je vkládat pomocí dvou různých metod - `add_geometry_creator_mod()` pro zdrojovou geometrii a `add_geometry_modifier_mod()` pro modifikátory. Více o fungování pluginů viz. 7.2.

Plugin jde přidat pouze v případě, že již není zaregistrovaný jiný se stejným názvem.

5.5 Modul `GUI`

Tento modul vytváří a spravuje grafické uživatelské rozhraní. K tomu je využita knihovna *Qt* firmy Trolltech ([4]), tu používá například desktopové prostředí KDE.

Jde o jednoduše použitelnou, objektově navrženou knihovnu psanou v c++. Díky tomu je možné jednoduše doprogramovávat nové, či pouze modifikované komponenty. Qt přidává do jazyka c++ pomocí sady `#define` maker nový mechanismus *signálů* a *slotů*. Z toho důvodu je nutné soubory s kódem, který tohoto mechanismu využívá předzpracovat programem `moc` (MetaObject Compiler), ten vygeneruje navíc pro každý vstup jeden nový soubor s c++ kódem, jenž se musí také zkompilovat a přilinkovat k projektu (více v dokumentaci Qt).

Tento modul vytváří hlavní okno aplikace (instance třídy `MAIN_WINDOW`), hlavní menu a podokna zobrazující jednotlivé *GUI moduly* (potomci třídy `BASE_GUI_MOD`).

Hlavní menu je vytvářeno dynamicky většinou inicializačním skriptem. Uživatel si tedy může jeho modifikací uzpůsobit pracovní prostředí. K přidání nové položky v menu slouží metoda `add_menu_item()`.

GUI modul `Viewport`

Slouží k zobrazování scény. Každý viewport má své vlastní nastavení (pozice a natočení kamery, způsob vykreslování, ...), které při požadavku na překreslení posílá modulu GLRenderer, zároveň svoji plochu označí jako aktuální vykreslovací oblast.

Podobným způsobem jako vykreslování probíhá i selekce, resp. hledání objektů spadajících do určité oblasti. Buď je to oblast okolo kurzoru myši, nebo oblast ohraničená oknem výběru. Všechny nutné údaje jsou opět poslány modulu GLRenderer.

GUI modul `Console`

Je napojen na funkční rozhraní modulu Console a posílá jí zadané příkazy. Dále je zaregistrován jako jeden z příjemců záznamů logovacího rozhraní (LogFile). Zobrazuje tedy veškerou logovací historii od momentu inicializace konkrétní instance GUI modulu.

GUI modul `Edit object`

Zpřístupňuje grafické rozhraní pro editaci objektů scény. Je napojen na modul Selection, který zpřístupňuje vybraný objekt scény spolu s jeho atributy.

Celé rozhraní je rozděleno do několika částí. Každá z nich se stará o část úkolů. Prvním je instance třídy `COMMON_PARAM_EDIT`, ta obstarává nastavování atributů společných pro všechny objekty scény (pozice, natočení, barva).

Dalším je instance třídy `MODIFIER_STACK_EDIT`, ta pracuje se zásobníkem modifikátorů. Skládá se ze tří částí. `MODIFIER_STACK_VIEW` poskytuje náhled na celý zásobník modifikátorů, společně s módy editace jednotlivých modifikátorů. `MODIFIER_STACK_OPERATOR` poskytuje rozhraní pro nastavení zobrazování objektu při editaci a manipulaci s modifikátory (odstraňování apod.). `MOD_PARAM_EDITOR` zobrazuje rohraní pro editaci aktuálně vybraného modifikátoru. Sestavení rozhraní má na starost plugin obsahující daný modifikátor.

5.6 Modul `GLRENDERER`

Jak název napovídá modul dává k dispozici funkční rozhraní postavené nad knihovnou *OpenGL* ([5]), to slouží nejen k vykreslování aktuálního stavu scény, ale i výběru objektů pomocí select bufferu.

Celé rozhraní příliš neabstrahuje od funkčního rozhraní knihovny OpenGL, pouze některé úkoly zjednodušuje a automatizuje.

Modul se nestará o přepínání jednotlivých render bufferů, nýbrž vykresluje vždy do toho, který je aktuální (OGL funkce `glMakeCurrent()`). A je jen na tom, kdo požaduje vykreslení, aby jako aktuální buffer nastavil ten svůj.

Pro rendering je připraveno několik metod, které dokážou vykreslit objekty, nebo seznamy objektů využívané v programu (nejen geometrické objekty, ale i pomocné). Varianty končící na _ID jsou používány pro vykreslování do select bufferu (nastavují ID vykreslovaného objektu).

render() - vykreslí scénu.

render_base() - vykreslí bázi jako trojici šipek ve směrech základních souřadných os.

render_edge() - vykreslí zadanou hranu.

render_edges() - vykreslí hrany v zadaném seznamu.

render_edges_ID() - při vykreslování hran ze seznamu do bufferu průběžně nastavuje jejich ID, pro použití při selekci.

render_face() - vykreslí zadaný polygon.

render_faces() - vykreslí seznam polygonů.

render_faces_ID() - vykreslí seznam polygonů a pro každý nastaví ID, které je využito při kreslení do select bufferu.

render_grid() - vykreslí pomocnou síť.

render_mesh() - vykreslí zadanou polygonální síť.

render_selection_window() - vykreslí obdélník vizualizující okenní výběr.

render_vertex() - vykreslí vrchol jako malý čtvereček.

render_vertex_ID() - vykreslí vrchol a nastaví jeho ID pro selekci.

render_vertices() - vykreslí seznam vrcholů.

render_vertices_ID() - vykreslí seznam vrcholů a nastavuje jejich ID pro selekci.

Modul vykresluje na základě svého vnitřního stavu, ten se dá modifikovat pomocí několika metod:

set_color*() - několik metod pro nastavování barev s různým určením. Při víceprůchodovém vykreslování není nutné přepínat barvy, modul se o ně stará sám.

set_id() - nastaví ID ukládané do select bufferu.

set_render_mode() - změní způsob vykreslování (drátěný model, zvýrazněné hrany, atd.).

set_render_params() - slouží k předání nastavení pro vykreslování scény.

set_transform() - nastavení transformačních matic.

5.7 Modul SELECTION

Tento modul obstarává záležitosti týkající se aktuálně vybraného objektu scény. Poskytuje o něm informace, stará se o nastavení jeho vykreslení pomocí modulu GLRenderer, řídí interaktivní manipulaci s objektem[2].

5.8 Modul SCENE

Nezávislý modul reprezentující scénu. Poskytuje rozhraní pro vkládání, odebírání a hledání objektů. Stará se také, aby při renderování byly vykresleny jen ty objekty, které nejsou v selekci (ty ošetří modul Selection).

Pro zjednodušení manipulace každý objekt dostane přiděleno ID číslo.

5.9 Modul FILE WORKER

Tento modul poskytuje funkční rozhraní pro nahrávání scény, ukládání scény, import a export objektů ze scény.

Je rozšiřitelný pomocí dvou druhů pluginů - nahrávacích a ukládacích (více viz. 7.3), které rozšiřují množinu typů souborů, jež lze načítat, resp. ukládat.

Všechny pluginy přidané do modulu musí být unikátně pojmenované (stačí jen v množině dané typem pluginu) - podle názvů probíhá vyhledávání.

5.10 Modul UNDO

Implementuje rozhraní pro žurnálování editačních kroků. Záznamy tvoří funkční objekty (potomci třídy `JOURNAL_RECORD`) s metodami `undo()` a `redo()`. Metody stejného jména má i samotný modul. Při jejich volání je použita odpovídající metoda na objektu, který byl poslední přidán (resp. poslední, na kterém bylo zavoláno `undo()`). Daný záznam by pak měl navrátit editační změny do doby před jeho přidáním.

Modul sám však do činnosti probíhající při volání zmiňovaných metod nevidí. Tím je sice snížena robustnost řešení (musíme se spolehnout, že ten kdo vložil do žurnálu daný záznam věděl co dělá), získáme ale navíc vyšší rozšiřitelnost. Například nějaký specializovaný plugin může provést těžko vysledovatelné, nebo náročné změny a jen on ví, jak vše jednoduše vrátit zpět.

Pro zjednodušení, či pomoc při ladění, modul poskytuje metody pro vytvoření zálohy jednoho objektu, případně celé scény, jako žurnálovací záznam.

5.11 Modul SCRIPT ENGINE

Modul s jednoduchým rozhraním - dokáže přijímat příkazy psané v jazyce python a spustit je v interpretu tohoto jazyka. Navíc dokáže spustit pythonský skript zadaný cestou v souborovém systému. Takto je například provedena poslední fáze inicializace programu.

Při své inicializaci modul vytvoří vestavěné knihovny jazyka python pro práci s programem. Ty mají svůj vlastní namespace EPY, kde jsou wrappery na všechny vestavěné pythonské fce ze zmiňovaných knihoven (viz. uživatelská dokumentace) a pomocné fce pro pro použití v c/c++.

[2] Kupříkladu podle informací zjištěných při ošetřování uživ. událostí mění jeho pozici. Při stornu navrátí vše do stavu před započetím editace.

5.12 Modul CONSOLE

Vytváří komunikační rozhraní mezi uživatelem a modulem `ScriptEngine`. Jde o funkční rozhraní, které je používáno GUI rozhraním pro konzoli.

Prvotním účelem tohoto modulu je posílání zadaných příkazů skriptovacímu jádru. Případně jakási cenzura zadaného textu - některé speciální příkazy mohou být odfiltrovány a použity jinak. V plánu je zatím neimplementovaný přepínač zadávání víceřádkových příkazů - ty se ukládají a jsou poslány k provedení po přijetí ukončovacího příkazu.

Z dalších možných rozšíření je nyní implementována pouze historie použitých příkazů tak, jak ji známe z jiných konzolí. Doplňování příkazů po stisku klávesy `Tab` není implementováno.

Kapitola 6

Objekty scény

Do scény je možné vložit pouze objekty, které jsou potomky třídy `BASE_SCENE_OBJECT`. Ta obsahuje informace o pozici a natočení ve scéně, ID objektu atd... Z ní jsou odděděny různé pomocné třídy (v současnosti pouze `SIMPLE_CAMERA`[1]).

Nejdůležitějšími objekty vkládanými do scény jsou instance třídy `GEOMETRY_CONTAINER`, obsahující *zdrojovou geometrii* a *zásobník modifikátorů*.

6.1 GEOMETRY_CONTAINER

Třída `GEOMETRY_CONTAINER` je potomkem třídy `BASE_SCENE_OBJECT`, oproti té navíc obsahuje *zásobník modifikátorů*, což je seznam *geometrických objektů*. První z nich nazvu *zdrojovou geometrií* a ostatní pojmenuji *modifikátory*. Rozdíl mezi nimi je v tom, že zdrojová geometrie nepotřebuje žádný vstup a svá data vytvoří čistě na základě algoritmu a svých parametrů (u koule například poloměr); modifikátor k vytvoření své geometrie potřebuje geometrii svého předchůdce jako vstup. Navíc pro zdrojovou geometrii stačí, aby byla potomkem třídy `GEOM_OBJ`, modifikikátory musí být potomky třídy `GEOM_MODIFIER`.

Ze spojitosti řetězce modifikatorů plyne, že nelze za sebe napojit v zásobníku modifikátorů dva objekty s rozdílným typem geometrie na výstupu u prvního a vstupu u druhého[2].

Zásobník modifikátorů je srdcem celého modelovacího systému v programu. Vytvořením *geometrického objektu* (potomka třídy `GEOM_OBJ`), který se uloží na dno zásobníku (více viz. 5.4),

6.2 Hierarchie geometrických objektů

Pro návrh hierarchie geometrických objektů jsem využil vícenásobné dědičnosti. Díky tomu jsem mohl oddělit rozhraní čistě pro geometrická data a rozhraní používané zásobníkem modifikátorů pokrývající procedurální stránku (konstrukce, aktualizace).

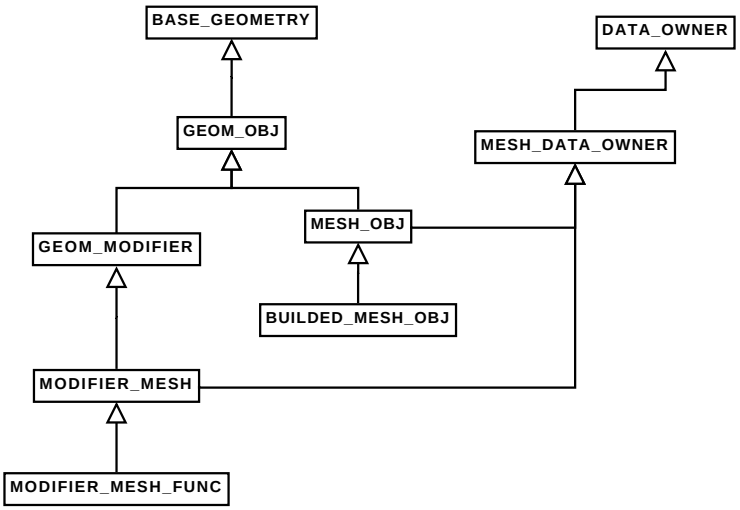
6.3 Plošková reprezentace

Pro reprezentaci polygonální sítě slouží tři seznamy pro uložení vrcholů, hran a plošek. Pro snazší, rychlejší vyhledávání a editaci jednotlivé subobjekty obsahují redundantní informace.

Základním prvkem je hrana. Zvolil jsem reprezentaci pomocí půlhran. Všechny hrany jsou rozděleny podélně na dvě půlhrany, kde každá z nich má polovinu informací (počáteční vrchol, sousední hrany, přilehlá ploška) a ukazatel na druhou polovinu. To má výhodu v čistě symetrickém přístupu ke všem informacím.

[1] Tu však zatím není možné využít a zobrazit ve scéně - každá její instance je svázána s nějakým konkrétním *viewportem*.

[2] V současnosti to problém není, jelikož je implementována pouze geometrie v ploškové reprezentaci. Kontrola typu při aplikaci modifikátoru implementována je.



Obrázek 6.1: Hierarchie geometrických objektů

Ve vrcholech je pak uložen seznam hran z něj vystupujících - půlhrana pro sousedni vrchol.

Základní editační funkce

Pro manipulaci s polygonální sítí je připravena sada základních editačních funkcí, z nichž by většina nutných úkonů měla být proveditelná bez větších problémů.

První skupinou jsou metody kontejneru polygonální geometrie (`MESH_DATA`), které slouží hlavně k vytváření (je více způsobů - metody jsou přetížené) a odebírání subobjektů (vrcholy, hrany, polygony) :

`add_vertex()` přidá nový vrchol.

`add_edge()` přidá novou hranu.

`add_face()` přidá novou plošku (polygon).

`remove_vertex()` odstraní vrchol a všechny incidenční hrany s ploškami.

`remove_edge()` odstraní hranu a incidenční plošky.

`remove_face()` odstraní plošku.

Druhou skupinou jsou pokročilejší funkce, jež berou, z důvodu provázanosti celé datové struktury, geometrii jako svůj parametr a editaci provádějí v ní :

`insert_vertex_in_edge()` vloží nový vrchol do hrany a rozdělí ji tak na dvě.

`split_face()` rozdělí polygon úsečkou spojující dva vrcholy polygonu.

`cut_face()` rozdělí plošku na dvě libovolnou úsečkou ležící v rovině polygonu.

`collapse_vertices()` několik vrcholů sjednotí do jednoho.

`contract_edge()` provede kontrakci hrany.

`isolate_faces()` zdvojí hrany a vrcholy ležící mezi zadanými polygony a zbytkem geometrie tak, aby vždy jedna z hran měla incidenční data z vybraných polygonů a druhá z ostatních - tím je přerušena veškerá spojitost se zbytkem geometrie.

`extrude_faces()` provede vysunutí vybraných polygonů ve směru průměru z normálových vektorů.

`triangulate_face()` daný polygon rozloží na trojúhelníky.

6.4 Subdivision surfaces

K zacházení se subdivision surfaces je vytvořeno vysokoúrovňové rozhraní, pomocí něhož jsem se snažil pokrýt co nejvíce společných rysů běžně užívaných schémat, aby se co nejvíce snížila nutnost je opakovaně implementovat pro každé nové schéma.

Zautomatizován je celý průběh dělení sítě i výpočet jednotlivých iterací. Jediné, co je nutné pro každé nové schéma vytvořit, jsou dělicí a repozicovací pravidla.

Pythonské rozhraní pro výpočet SubSurf je velmi těsně navázáno na to, které je psané v jazyce c++.

Implementována je zatím pouze výpočetní struktura pro primární schémata, jak je uvedeno v první části textu. Pro ostatní schémata (duální a nezařaditelná) jsem připravil knihovnu psanou v Pythonu. Výpočet probíhá voláním v cyklu funkce pro výpočet jedné iterace. Tuto funkci musí autor schématu napsat sám, případně modifikovat nějakou již hotovou.

Kapitola 7

Možnosti rozšiřování

7.1 Idea rozšiřitelnosti

Ve všech modulech, u nichž je možné přidat nové funkce, jsem se snažil, aby tato rozšíření byla umožněna pomocí co nejuniverzálnějšího rozhraní.

V případě pluginů u konkrétních modulů jsou tyto vždy odvozené od nějaké abstraktní třídy, která definuje povinné rozhraní pluginu. Jde o transparentní řešení. Modul se nemusí zajímat, jak a kde je daný plugin implementován. Kupříkladu rozšiřovaný modul vůbec neví o použití interpretu jazyka python - pro pythonské funkce a objekty se použije obalovací c++ objekt děděný od zmiňovaného pluginového rozhraní.

7.2 Geometrické pluginy

Jak již bylo uvedeno množinu lze množinu objektů, které mohou být vloženy do scény rozšířit pomocí pluginů. Pro přidávání za běhu jsou však připravena rozhraní pouze pro geometrické objekty (zdrojové geometrie i modifikátory). Ta mají totiž připravena obalující funkce a objekty pro Python (ta naleznete v uživatelské dokumentaci).

Když chceme vytvořit nový geometrický plugin stačí nadefinovat objekt s funkčním rozhraním `GEOMETRY_PLUGIN_IFC`, které sjednocuje několik specializovaných funkčních rozhraní. Jsou to `GEOMETRY_CREATION_IFC`, `GUI_CREATION_IFC` a `EVENT_HANDLER_IFC`.

Tvorba geometrie

`GEOMETRY_CREATION_IFC` definuje metody, které jsou vyžadovány objekty spravujícími geometrická data (např. `GEOMETRY_STACK`). Tou hlavní je `rebuild()`. Ta by měla spouštět přestavbu celé geometrie, což je nutné například při změně nějakého parametru, nebo u modifikátorů při změně geometrie, na kterou je aplikován.

Druhou metodou je `recalc()`. Ta je určená jako urychlující prvek pro případy, kdy není měněná celá geometrie, ale pouze její část. V současnosti není tento systém dotažen do konce. Proto je v těle této metody často pouze přesměrování na metodu `rebuild()`.

Každý z objektů by měl také poskytovat možnost uložení jejich dat do souboru a jejich opětovné načtení. Na to by měly sloužit metody `get_save_data()` a `restore_from_save_data()`.

Vytvoření GUI

Metody z rozhraní `GUI_CREATION_IFC` slouží k interakci s uživatelským rozhraním. Pokud má objekt možnost přepnout se do nějakého módu editace, musí jejich názvy a idetifikační čísla vratít metodou `get_mod_names()`.

Uživatelské rozhraní pro komunikaci s objektem je vytvořeno metodou `create_gui()`. Ta potřebuje jako parametr objekt, který dokáže vytvářet komponenty uživatelského rozhraní, a aktuální

mód editace. Voláním metod předaného objektu vytvoří své prostředí pro komunikaci s uživatelem, které může být závislé na módu editace.

Módy editace

Některé z geometrických objektů si pro vytvoření svých geometrických dat nevystačí s množinou parametrů, ale potřebují složitěji interagovat s uživatelem. Hlavním zástupcem je například modifikátor *Edit mesh*. Ten nemá žádné parametry, takže při jeho aplikaci nenastane žádná změna. Ty proběhnou až po přepnutí do některého ze tří editačních módů a editaci sítě uživatelem na úrovni vrcholů, hran, nebo polygonů.

Při přepnutí se do nějakého módu editace musí objekt ošetřovat uživatelské události. Z toho důvodu musí nějakým způsobem implementovat metodu `handle_event()` zděděnou z `EVENT_HANDLER_IFC`.

7.3 Pluginy pro načítání a ukládání do souboru

Načítání ze souboru

Vytvoření načítacího pluginu pro modul FileWorker spočívá v naprogramování a předání objektu s rozhraním zděděným od `FILE_LOADING_FTOR`.

To znamená vytvořit objekt, jenž dokáže vytvořit soubor a uložit do něj seznam předaných objektů scény. Stejně jako u geometrických pluginů je i zde předpřipraven obalovací objekt, který dokáže volat fce pythonu jemu zadané v konstruktoru spolu s unikátním názvem pluginu.

Ukládání do souboru

Pro ukládací plugin platí obdobné jako pro načítací, jen musí implementovat rozhraní `FILE_SAVING_FTOR`.

7.4 Obecné skripty

Není problém napsat v Pythonu jakýkoliv skript, který se má během používání programu spouštět. Můžeme používat libovolné další pythonské knihovny, musí být ovšem přítomny v systému a interpret k nim potřebuje znát cestu (v případě potřeby ji doplníme do seznamu prohledávaných adresářů).

Vytvořit můžeme, jak skript, který se vždy provede při spuštění programu, tak knihovnu funkcí, kterou lze importovat do dalších skriptů.

Více se lze dozvědět v uživatelské dokumentaci v části o skriptování.

Možnosti začlenění do programu

Pokud se nejedná o skript implementující nějaký plugin, musíme ho začlenit do programu nepřímo, buď jeho uvedením v inicializačním skriptu, spuštěním z konzole, případně z jiného skriptu, nebo lze vytvořit položku v hlavním menu programu, která může iniciovat jeho spuštění.

Kapitola 8

Další možnosti rozšiřování

8.1 Pluginy jako dynamicky linkované knihovny

Veškerá funkční rozhraní pro pluginy (jak pro modul Modeler, tak pro FileWorker) jsou navržena tak, aby o pluginech nepředpokládala žádné dodatečné informace a stačilo jim pouze implementování daných rozhraní pro interakci s moduly. Díky tomu je naprosto jedno odkud dané pluginy jsou a jak vnitřně fungují. V současnosti lze jako pluginy využít jen speciální pythonské skripty, ale kvůli transparentnosti rozhraní žádný z rozšiřovaných modulů nemá o nějakém pythonu vůbec ponětí.

Celý návrh je tedy připraven pro přijímání pluginů i odjinud než jen ze skriptů. Ty jsou pro určité časově náročné úkoly, z důvodu interpretování kódu, pomalé, a jsou vhodné hlavně k vývoji a experimentům.

Vhodnou alternativou pro implementování finálních verzí pluginů jsou *kompilované* dynamicky linkované knihovny. Stačí, aby při svém zavádění přidaly objekt (případně objekty) s požadovaným pluginovým rozhraním.

Je třeba doprogramovat správce dynamicky linkovaných knihoven, jenž by se staral o jejich nahrávání, uvolňování a zjišťování nově přidaných. Vhodné by bylo navrhnout ho jako nový programový modul, aby zapadl do navržené filozofie. Jednalo by se také zřejmě o nejčistší řešení - žádné další moduly by nebyly zasaženy jeho přítomností.

8.2 Přidání nového programového modulu

Vytvoření nového programového modulu je snadné řešení rozšíření programu o funkce a nástroje, které pokrývají určitou oblast, zatím nezasaženou již realizovanými moduly.

O programové moduly se v modulu Kernel stará pomocná datová struktura `MODULE_LIST`. Přidání nového modulu tedy obnáší zaregistrování v této struktuře.

Registrace se musí provést na úrovni zdrojových kódů[1] přidáním celočíselné konstanty do anonymního výčtového typu, zvýšením konstanty `MOD_COUNT` udávající celkový počet přítomných programových modulů a na závěr zajistit vytvoření instance modulu v metodě `create_modules()` - tady pozor na komentáře k dalším modulům, jelikož některé vyžadují konstrukci až po ostatních modulech a neopatrné narušení může mít závažné následky (například modul GUI se musí konstruovat po všech modulech, ke kterým poskytuje přístup).

[1] později by se mohlo rozšířit o zavádění za běhu

Literatura

[1] Foley J., van Dam A., Feiner S., Hughes J.: *Computer Graphics, Principles and Practice*, Addison-Wesley, 1997.

[2] Zorin D., Schröder P.: *Subdivision for Modeling and Animation*, SIGGRAPH 2000 Course Notes.

[3] *www.PYTHON.org*, stránka věnovaná jazyku Python.

[4] *trolltech.com*, stránka firmy Trolltech, která vyvýjí knihovnu Qt.

[5] *nehe.gamedev.net*, návody pro knihovnu OpenGL.