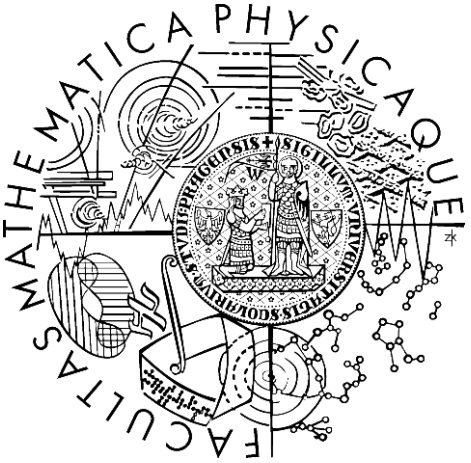
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Tomáš Herceg

Grafický editor 3D scén pro projekt AGE

Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: *Mgr. Pavel Ježek*

Studijní program: *Informatika, Programování*

2010

Poděkování

Panu Mgr. Pavlu Ježkovi za pomoc, cenné náměty a připomínky při počátečním návrhu aplikace a při psaní tohoto textu.

Prohlášení

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne: 24. 5. 2010 Tomáš Herceg

_____

Obsah

ÚVOD 5

Cíle práce 5

1. PROBLEMATIKA RENDEROVÁNÍ 3D SCÉN 6

1.1. Reprezentace 3D scény 6

1.2. Transformace bodů ze 3D prostoru na obrazovku 6

1.3. Aplikace textury a barev 6

1.4. Viditelnost ploch a bodů 7

1.5. Vertex a index buffer 7

1.6. Shadery 7

2. SPECIFIKACE APLIKACE AGE A EDITORU 3D SCÉN 8

2.1. Aplikace AGE 8

2.2. Editor 3D scén 8

3. ANALÝZA 9

3.1. Moduly aplikace AGE 9

3.2. Registrace modulů v aplikaci 10

3.3. Šablony pro nové soubory 10

3.4. Renderování 3D scény uvnitř WPF aplikace 11

3.5. Použitý DirectX wrapper 12

3.6. Více pohledů na scénu 13

3.7. Engine pro vyhodnocování výrazů 13

3.8. Objekty a reprezentace scény 15

3.9. Objekty a modifikátory 15

3.10. Nástroje editoru a dekorátory 16

4. IMPLEMENTACE APLIKACE AGE A EDITORU 3D SCÉN 17

4.1. Hlavní okno aplikace AGE 17

4.2. Dceřinné MDI okno modulu 18

4.3. Renderování 3D scény uvnitř aplikace 19

4.4. Reprezentace 3D scény 20

4.5. Engine pro vyhodnocování výrazů 21

4.6. Implementace konkrétních 3D objektů ve scéně 22

4.7. Generování vertexů a indexů 24

4.8. Používání funkcí a datových typů ve výrazech 25

4.9. Komponenta TreeObjectPropertyGrid 27

4.10. Nástroje editoru 29

4.11. Dekorátory 31

4.12. Výběr objektů 31

4.13. Formát uložení 3D scény (A3D) 31

Specifikace formátu A3D 32

Objekty a vlastnosti 33

4.14. Rozšiřitelnost 35

5. SROVNÁNÍ S EXISTUJÍCÍMI ŘEŠENÍMI NA TRHU 36

5.1. Autodesk 3ds Max 36

5.2. Google SketchUp 37

5.3. Blender 38

5.4. AGE 3D Editor 38

ZÁVĚR 40

Možná budoucí rozšíření 40

POUŽITÁ LITERATURA 41

Abstrakt

Název práce:	*Grafický editor 3D scén pro projekt AGE*
Autor:	*Tomáš Herceg*
Katedra (ústav):	*Katedra distribuovaných a spolehlivých systémů*
Vedoucí bakalářské práce:	*Mgr. Pavel Ježek, Katedra distribuovaných a spolehlivých systémů*
E-mail vedoucího:	*pavel.jezek@dsrg.mff.cuni.cz*

Abstrakt:

Cílem práce je vytvořit grafický editor, který umožní tvorbu a editaci grafických scén ve 3D prostoru. Editor by měl být realizován jako rozšiřující modul do projektu AGE. Jelikož je projekt AGE teprve ve fázi vývoje, měla by jako součást této bakalářské práce vzniknout i jeho první verze.

Samotný editor vytvořený v rámci práce by měl podporovat vytváření základních primitivních objektů, manipulaci s jednotlivými vrcholy, stěnami a částmi těchto objektů, a dále pak implementovat základní nástroje pro mapování textur.

Jako cílová platforma bude použito prostředí .NET Framework a pro zobrazování grafických výstupů je doporučeno použít technologie Windows Presentation Foundation a DirectX.

Klíčová slova: *Počítačová grafika, Windows Presentation Foundation, DirectX, 3D*

Abstract

Title:	*Graphical 3D Editor for the AGE Project*
Author:	*Tomáš Herceg*
Department:	*Department of Distributed and Dependable Systems*
Supervisor:	*Mgr. Pavel Ježek, Department of Distributed and Dependable Systems*
Supervisor's e-mail address:	*pavel.jezek@dsrg.mff.cuni.cz*

Abstract:

The goal of the thesis is to create an editor of graphical three-dimensional scenes. The editor is going to be an AGE project add-in. Because the AGE project is by the time of writing this thesis still in development its first version should be implemented as well as a part of this thesis.

The 3D editor add-in should support creation of simple 3D primitives, manipulation with vertices, faces and mesh parts and also should contain basic tools for texture mapping.

The Microsoft .NET Framework will be used as a target platform and all graphics content will be rendered using the Windows Presentation Foundation and DirectX technologies.

Keywords: *Computer graphics, Windows Presentation Foundation, DirectX, 3D*

Úvod

Tato bakalářská práce se zabývá implementací jednoduchého editoru 3D scén, který bude podporovat základní manipulace s objekty v trojrozměrném prostoru. Tento editor bude realizován jako modul do aplikace AGE, která je rovněž součástí této bakalářské práce. Aplikace AGE (*Advanced Graphics Editor*) je obecné hostitelské prostředí, které bude prostřednictvím rozšiřujících modulů podporovat tvorbu a úpravy 2D i 3D grafiky.

Hlavním cílem této práce je vytvořit modul, který bude obsahovat základní nástroje pro tvorbu a editaci 3D scén, a dále vytvořit první verzi aplikace AGE, jež bude sloužit jako hostitelské prostředí pro tento modul.

Modul 3D editoru by měl umožňovat základní operace s tělesy ve scéně, například jejich pozicování, otáčení, změnu měřítka, manipulaci s jejich jednotlivými vrcholy a stěnami. Dále bude obsahovat základní podporu materiálů a základní nástroje pro aplikování textury na těleso.

Druhým cílem této práce je zhodnotit a otestovat dva dílčí koncepty, které se při implementaci 3D editoru uplatnily.

Prvním konceptem ke zhodnocení je schopnost prostředí Windows Presentation Foundation[1] hostovat v aplikacích grafiku vykreslovanou pomocí rozhraní DirectX. To je technicky poměrně obtížný úkol, neboť samotná WPF používá DirectX interně, aby dosáhla vyššího výkonu při vykreslování uživatelského rozhraní a obsahu okna.

Druhým konceptem, který bude v této práci zhodnocen a vyzkoušen, je podpora výrazů uvnitř 3D scény pro definování vztahů mezi tělesy. Editor 3D scén, který bude jako součást této práce implementován, umožní při definici parametrů těles ve scéně používat matematické výrazy. Pomocí těchto výrazů bude možné například pozici, velikost či barvu objektu ve scéně vypočítat z pozice či velikosti ostatních těles.

Tato práce může sloužit jako základní seznámení s architekturou aplikace AGE pro vývojáře, kteří se chystají vyvíjet další moduly do této aplikace. 3D editor může sloužit jako vzorový příklad.

Tento text může též být užitečný pro vývojáře, kteří vyvíjí vlastní 3D editor, jelikož kromě popisu konkrétní implementace obsahuje i některé poznatky a rozhodnutí, která platí obecně.

Cíle práce

Jak již bylo řečeno v předchozím textu, cílem této práce je vytvořit hostitelskou aplikaci AGE s modulem jednoduchého 3D editoru, dále zhodnotit možnosti integrace DirectX scény uvnitř okna ve WPF a konečně zhodnotit nové možnosti, které přináší možnost definování vztahů mezi tělesy ve scéně pomocí výrazů v porovnání s ostatními a běžně používanými editory 3D grafiky.

[1] V následujícím textu se bude místo *Windows Presentation Foundation* používat též jeho oficiální zkratka *WPF*.

1. Problematika renderování 3D scén

V této kapitole je stručně popsán a nastíněn způsob, jakým se v prostředí knihovny DirectX 9 reprezentují a vybreslují 3D objekty.

1.1. Reprezentace 3D scény

Existuje několik různých způsobů, pomocí kterých můžeme v paměti reprezentovat prostorová geometrická tělesa. Nejčastěji se v grafických aplikacích používá tzv. *polygonální reprezentace* [1], v níž 3D objekt reprezentujeme jako sadu mnohoúhelníkových plošek. Samozřejmě mnoho těles (například kouli nebo jakékoliv těleso obsahující křivku) nelze rozložit na sadu mnohoúhelníků přesně, v takovém případě je nutné provést aproximaci a použít dostatečné množství mnohoúhelníků, aby se tato aproximace na výstupu neprojevila vůbec, či aby bylo zkreslení co možná nejmenší.

Metoda *polygonální reprezentace* je pro naše použití vhodná, neboť dnešní grafické karty podporují na hardwarové úrovni tzv. *trojúhelníkovou reprezentaci* [1], v níž se objekt reprezentuje soustavou trojúhelníků. Polygonální reprezentaci lze na trojúhelníkovou poměrně snadno převést.

1.2. Transformace bodů ze 3D prostoru na obrazovku

V okamžiku, kdy máme těleso reprezentované pomocí trojúhelníků ve 3D prostoru, je nutné je nyní vykreslit na obrazovku. V prostředí DirectX se běžně používá transformace pomocí tří matic *world*, *view* a *projection* [2].

World matrix (globální matice) je matice, která udává globální transformaci celé scény. Lze ji využít například k pozicování a otáčení objektů v rámci scény. *View matrix* (matice pohledu) definuje pozici a veškerá další nastavení kamery. *Projection matrix* (projekční matice) pak definuje použité promítání včetně všech jeho parametrů. Konstrukci těchto matic zajišťují funkce knihovny DirectX.

Transformace bodů z prostoru objektů scény (tzv. *object space* [2]) do prostoru obrazovky (tzv. *screen space* [2]) pak probíhá přinásobením vektoru souřadnic bodu ve scéně postupně maticemi *world*, *view* a *projection* zprava. Výsledkem je vektor udávající souřadnice promítnutého bodu na obrazovce a jeho hloubku vzhledem k průmětně. Výhodou této metody je její jednoduchost a podpora na úrovni hardwaru u grafických karet.

1.3. Aplikace textury a barev

V okamžiku, kdy jsou pro každý trojúhelník spočítány pozice bodů na obrazovce, je nutné trojúhelník vyplnit. Každý vrchol ve scéně může mít definovanou barvu a mapovací souřadnice pro texturu, které se obvykle značí U a V. Body uvnitř trojúhelníka se na grafickém výstupu obvykle vyplňují *řádkovým algoritmem* [3] a pro každý pixel se dopočítá barva a mapovací souřadnice jako lineární kombinace podle jeho vzdáleností od vrcholů aktuálního trojúhelníka.

V případě, že je na trojúhelník aplikována *rastrová textura* [4], pro každý renderovaný pixel se spočítají U a V souřadnice a podle nich se určí barva z pixelu dané textury. Mapovací souřadnice (0, 0) označují obvykle levý horní roh textury, souřadnice (1, 1) pak dolní pravý roh. Pokud jsou souřadnice mimo tento rozsah, použije se typicky jen jejich desetinná část. Tím lze dosáhnout opakování vzorku na textuře.

1.4. Viditelnost ploch a bodů

Pro řešení viditelnosti ploch, totiž aby vzdálenější objekty při vykreslování nezakryly objekty, které jsou blíže kameře, se v prostředí DirectX používá standardně technika *Z-Buffer* [5]. Její základní idea spočívá v tom, že pro každý pixel na výstupu se ve speciálním poli (tzv. hloubkový buffer) pamatuje vzdálenost nejbližšího již vykresleného bodu ve scéně od kamery, který se na tento pixel promítnul. Před vykreslením každého bodu ze scény na daný pixel se nejprve ověří, zda-li již není na stejném pixelu vykreslen bod scény, který je blíže. Pokud není, bod se vykreslí a jeho vzdálenost od průmětny se zapíše do bufferu.

1.5. Vertex a index buffer

Vzhledem k tomu, že dnešní grafické karty disponují svou vlastní pamětí, která bývá často rychlejší než běžná operační paměť počítače, vyplatí se reprezentace objektů scény a textury ukládat přímo do této paměti. V prostředí knihovny DirectX se jednotlivé trojúhelníky ukládají ve speciálních bufferech [6] v předem určeném formátu.

Prvním bufferem je tzv. *vertex buffer*[2], do nějž se ukládají samotné vrcholy trojúhelníků. Ty obvykle obsahují pozici ve scéně, barvu, mapovací souřadnice a normálu. Druhým bufferem je tzv. *index buffer*, který je nepovinný a používá se v případech, kdy je každý vrchol součástí většího množství trojúhelníků. Index buffer je pole čísel, které definuje pořadí, v jakém se vrcholy z vertex bufferu budou číslovat.

V případě, kdy index buffer není definován, je *i*-tý trojúhelník definován *i*-tou trojicí vrcholů ve vertex bufferu. Naproti tomu pokud index buffer existuje, je *i*-tý trojúhelník definován vrcholy z vertex bufferu, jejichž indexy jsou v *i*-tá trojice čísel v index bufferu.

Ve chvíli, kdy se mají trojúhelníky vykreslit, stačí zavolat příslušnou metodu knihovny DirectX.

1.6. Shadery

Vzhledem k tomu, že dnešní grafické karty obsahují velmi výkonné procesory, vyplatí se provádět některé operace s vrcholy a renderovanými objekty přímo na procesoru grafické karty. Programům, které se na grafické kartě provádí, se obecně říká *shadery*, přičemž rozlišujeme několik druhů těchto shaderů.

Nejběžněji se používají *vertex shadery* a *pixel shadery*. Vertex shader se spustí pro každý renderovaný vrchol trojúhelníka (vertex) a umožňuje s ním provést nějakou operaci. Vertex shader typicky souřadnice vrcholu vynásobí maticemi *world*, *view* a *projection* a dále na základě normály trojúhelníka, do nějž vrchol patří, může počítat parametry pro osvětlení. Pixel shader se naproti tomu spouští pro každý renderovaný pixel a obvykle podle spočítaných mapovacích souřadnic zjistí barvu pixelu z rastrové textury.

[2] Pro termíny *vertex buffer* a *index buffer* neexistuje obecně uznávaný a používaný český ekvivalent.

2. Specifikace aplikace AGE a editoru 3D scén

V následujících odstavcích jsou popsány funkce, které bude aplikace AGE a editor 3D scén implementovat.

2.1. Aplikace AGE

Aplikace AGE bude poskytovat hlavní infrastrukturu pro moduly, které se uvnitř ní budou spouštět. Při startu aplikace načte všechny nainstalované moduly a zobrazí hlavní okno. Toto hlavní okno umožní otevřít jeden nebo více souborů a pomocí modulů, které daný typ souboru podporují, je umožní upravovat.

Hlavní okno by pro tyto účely mělo obsahovat hlavní panel, do nějž bude mít každý modul možnost přidat vlastní záložky s ovládacími prvky. Tento hlavní panel bude též obsahovat základní aplikační tlačítka, jež umožňují vytvořit nový soubor, otevřít existující soubor z disku a uložit soubory, s nimiž se právě pracuje. Samotné ukládání a načítání souborů už budou implementovat jednotlivé moduly, hostitelská aplikace AGE by tyto funkce měla pouze zpřístupňovat pomocí tlačítek a standardních klávesových zkratek, přičemž by též měla zajišťovat zobrazení oken pro výběr umístění otevíraných či ukládaných souborů.

2.2. Editor 3D scén

Editor 3D scén je modul, který se při spuštění aplikace AGE načte a připraví pro použití v této aplikaci. V okamžiku otevření souboru obsahujícího 3D scénu by se v hlavním okně měla vytvořit záložka s editorem této scény. Veškeré funkce od ovládacích prvků editoru až po renderování scény budou implementovány v tomto modulu.

3D editor bude definovat základní prostorové objekty – krychli, kolmý jehlan, n-boký hranol, kouli a n-úhelník. Tato základní sada bude rozšiřitelná o další tělesa.

Editor bude umožňovat vybírat objekty ve scéně, vybírat jednotlivé vrcholy a stěny objektů a na těchto výběrech provádět základní transformace – posunutí, otočení a změny měřítka. Na objekty nebo na jejich vrcholy či stěny bude dále možné aplikovat tzv. modifikátory, například pro změnu barvy či nastavení mapování textury.

Jednotlivým objektům ve scéně bude možné přidělit materiál, který může definovat jednu podkladovou barvu, jednu texturu a míru průhlednosti této textury vzhledem k podkladové barvě.

Pro každý objekt ve scéně bude možné definovat jeho libovolný parametr (např. výšku, šířku, hloubku, pozici, materiál atd.) nejen jako absolutní hodnotu, ale též jako výraz. Díky tomu bude možné definovat vztahy mezi objekty. Budou-li ve scéně například krychle s názvy A a B, bude možné nastavit, aby šířka krychle A byla vždy rovna šířce krychle B. V případě, že šířka krychle B později změní, automaticky bude přepočítána i šířka krychle A.

Do scény půjde též přidat speciální těleso – graf 2D funkce. Tomuto tělesu bude možné jako jeden z parametrů nastavit výraz definující hodnotu funkce v závislosti na hodnotách dvou parametrů X a Y, které jsou z uzavřeného intervalu od 0 do 1. Hodnota tohoto výrazu se pak použije při modelování grafu této funkce.

Modul 3D editoru bude umožňovat uložit scénu do souboru a opět tuto scénu načíst. Formát souboru scény je popsán v kapitole 4.13.

3. Analýza

Při návrhu a implementace aplikace AGE a 3D editoru bylo nutno provést několik důležitých rozhodnutí. V této kapitole je popsáno a diskutováno, jaké možnosti se u těchto rozhodnutí nabízely, které z nich byly nakonec vybrány a proč.

3.1. Moduly aplikace AGE

Aplikace AGE je, jak již bylo řečeno dříve, jen obecné prostředí, uvnitř něhož se hostují jednotlivé moduly. Při startu by tato aplikace měla načíst všechny instalované moduly a zpřístupnit jejich funkce. Objevila se tedy otázka, jaké služby bude modul aplikaci poskytovat a jak bude distribuován.

Od modulu se očekává jen poměrně malé množství funkcí. Předně musí být schopen aplikaci AGE dát vědět, které typy souborů umí zpracovávat. Další funkcí, která se od něj očekává, je vytvoření dceřinného MDI[3] okna editoru, jež soubor načte, zobrazí a umožní uživateli s ním pracovat.

Jednou ze zvažovaných možností bylo vymyslet speciální skriptovací jazyk, v němž by se moduly implementovaly, a vlastní formát souboru, v němž by tyto skripty byly uloženy a posléze distribuovány. Toto řešení by mohlo být výhodné, jelikož by bylo možné navržený skriptovací jazyk lépe přizpůsobit pro potřeby implementace modulů, například zkrátit některé syntaktické konstrukce, navrhnout jiný přístup pro ošetření událostí uživatelských vstupů z klávesnice a myši.

Druhou možností bylo použít standardní prostředky .NET Frameworku, totiž techniku Reflection, moduly implementovat v libovolném jazyce nad .NET Frameworkem a distribuovat je jako DLL knihovny. Toto řešení je implementačně relativně jednoduché a nevyžaduje, aby se vývojář učil nový programovací jazyk. Na straně hostitelské aplikace stačí deklarovat rozhraní, jehož funkcionalitu musí modul implementovat.

Po zvážení byla zvolena druhá možnost, jelikož naimplementovat vlastní DSL[4] a jeho běhové prostředí by bylo velmi pracné. Mohlo by to přinést sice různá zajímavá vylepšení a usnadnění práce v podobě různých rozšíření syntaxe, na druhou stranu za cenu přinucení vývojáře naučit se nový jednoúčelový programovací jazyk.

Naproti tomu dynamicky načíst .NET assembly za běhu aplikace a vytvořit instanci třídy v ní definované je za použití funkcí ze jmenného prostoru System.Reflection velmi jednoduché. Vzhledem k tomu, že assembly obsahující modul může být napsána v libovolném .NET programovacím jazyce, má vývojář možnost vybrat si jazyk dle vlastních preferencí a může při vývoji použít již existující vývojové prostředí, nástroje pro ladění kódu apod.

O tom, jaké rozhraní modul a vytvářené dceřinné MDI okno musí implementovat, pojednávají kapitoly 4.1 a 4.2.

---

[3] MDI je zkratka termínu *Multiple Document Interface* a označuje se s ním uživatelské rozhraní okna, které uvnitř umožňuje hostovat několik dceřinných oken.

[4] DSL je zkratka pro *Domain Specific Language*.

3.2. Registrace modulů v aplikaci

Bylo tedy rozhodnuto, že moduly aplikace AGE budou distribuovány ve formě standardní .NET assembly. Nastala otázka, jakým způsobem bude aplikace AGE moduly hledat a dle jakých pravidel je bude načítat.

První možností bylo nechat vývojáře modulů, aby DLL knihovny při instalaci nakopírovali do libovolného adresáře a tuto knihovnu nějakým způsobem zaregistrovali u aplikace AGE, například v registrech nebo v konfiguračním souboru. Toto řešení vyžaduje instalátor, neboť knihovnu nestačí nakopírovat do nějakého adresáře, ale je nutné ji i registrovat.

Druhá možnost, která přicházela do úvahy, bylo v adresáři aplikace AGE vytvořit speciální adresář, kam by jednotlivé moduly nakopírovaly své knihovny. Nebyla by třeba žádná registrace, neboť aplikace AGE by přesně věděla, kde má moduly hledat. Zkrátka by jen načetla všechny knihovny, které by v daném adresáři našla.

Při použití druhé možnosti by bylo možné instalovat rozšíření aplikace pouhým nakopírováním souborů bez nutnosti vytváření složitějšího instalátoru, který rozšíření zaregistruje. První řešení naopak umožňuje aplikaci některé moduly deaktivovat bez nutnosti jejich odinstalace (například v případě, že se je uživatel rozhodně zakázat kvůli jejich nestabilitě apod.), stačí do konfiguračního souboru přidat informaci, že tento modul se nemá načítat. To by bylo možné řešit i u první možnosti, například zapsáním zakázaných knihoven do konfigurace.

Další výhoda prvního řešení se může projevit v případě, kdy danou pracovní stanici používá více uživatelů, přičemž každý si zakoupil a nainstaloval jiné rozšiřující moduly. Vzhledem k tomu, že konfigurační soubor aplikace je pro každého uživatele separátní, nebude ve své konfiguraci mít uživatel zaregistrované moduly, pro něž například nemá licenci (pokud je tato licence vázána na jednoho uživatele a ne na jednu pracovní stanici), a tudíž tyto moduly nebude moci používat. Jedinou možností, jak by „cizí“ modul mohl načíst, by byla ruční úprava konfiguračního souboru. I toto by se dalo pomocí první možnosti řešit, například jedním sdíleným adresářem a jedním adresářem pro každého uživatele, ale tím už se ono řešení mírně komplikuje.

V tomto případě byla zvolena první možnost, jelikož nutnost registrace modulů významně nezkomplikuje proces jejich instalace (jde o připsání jednoho elementu do XML souboru) a umožní to do budoucna ke každému nainstalovanému modulu ukládat další informace, například jeho nastavení atd., pokud by se tato potřeba časem objevila.

Způsob, jakým se do aplikace AGE jednotlivé moduly registrují, je popsán v uživatelské dokumentaci v kapitole Konfigurační soubor aplikace.

3.3. Šablony pro nové soubory

Tlačítko pro vytvoření nového souboru by mělo uživateli umožnit vybrat, jaký typ souboru hodlá vytvořit. Vzhledem k tomu, že pro některé moduly bude jistě třeba mít možnost, aby uživatel nemusel vždy vytvářet soubor od začátku, ale aby mohl vyjít z již předpřipravené šablony, mělo by být každému modulu umožněno, aby definoval několik šablon pro nový soubor. Bylo nutné rozhodnout, jakým způsobem se jednotlivé šablony budou společně s moduly distribuovat a zda-li (a případně jakým způsobem) dát uživateli možnost, aby tyto výchozí šablony mohl upravit a nebo dokonce přidávat šablony vlastní.

Jednou z možností bylo šablony zabalit přímo do knihovny modulu. Díky tomu by byly šablony chráněny proti poškození, nechtěné úpravě či smazání, navíc by se mírně zjednodušila možnost jejich aktualizace s novými verzemi modulu. Výhodou by bylo též to, že všechny soubory vztahující se k modulu, by byly na jednom místě, v jedné DLL knihovně.

Druhou možností bylo vytvořit v adresáři aplikace speciální adresář, do nějž by se šablony pro nové soubory umístily. Šablona by byl obyčejný soubor, který umí daný modul otevírat a ukládat. Díky tomu by uživatel snadno mohl vytvořit vlastní soubor, uložit jej do adresáře šablon a začít jej jako šablonu používat, navíc by této šabloně bylo možné vytvořit i vlastní obrázek. Soubor šablony by se mohl lišit například příponou, aby bylo možné snadno poznat, zda-li se jedná o šablonu, či nikoliv, na druhou stranu to není nutné, pokud je formát šablony stejný jako formát obyčejného souboru.

Třetí možností by bylo zkombinovat dvě možnosti předchozí, totiž umožnit výrobcům modulů dodávat své vlastní chráněné šablony, a zároveň dát uživatelům možnost přidávat si šablony vlastní.

V tomto případě byla vybrána možnost druhá, jelikož pro první verzi aplikace postačuje a umožňuje do budoucna rozšíření na možnost třetí. Hlavní výhodou z uživatelského hlediska je možnost snadné úpravy a tvorby vlastních šablon.

O tom, do jakého umístění se šablony v aplikaci ukládají, pojednává kapitola Adresářová struktura aplikace v uživatelské dokumentaci.

3.4. Renderování 3D scény uvnitř WPF aplikace

Jak bylo již zmíněno v úvodu, jedním z vedlejších cílů této práce je zhodnotit možnosti začlenění scény renderované pomocí technologie DirectX do WPF aplikace.

V .NET Frameworku 3.5 SP1 se objevila ve WPF nová třída *D3DImage* [7], která je určena právě pro tento scénář. Tato třída se používá nejčastěji ve spojení s komponentou *Image*, dědí totiž ze třídy *ImageSource* a slouží tedy jako zdroj obrazových dat komponenty *Image*. První možností tedy je použít tuto třídu, podrobný příklad použití a popis fungování této komponenty je uveden například v článku [8]. Výhodou komponenty *D3DImage* oproti jiným možnostem integrace je možnost obejití tzv. *airspace restriction* [9], což v praxi znamená, že každý pixel může být renderován buď pomocí WPF, nebo pomocí DirectX, ale ne pomocí obou technologií najednou.

Druhá možnost, která byla jediná možná před uvedením .NET Frameworku 3.5 SP1, je použití třídy *HwndHost*, která uvnitř WPF aplikace vymezí prostor umožňující hostovat klasické Win32 okno. Tato komponenta se používá především pro integraci *Windows Forms* komponent ve *WPF* okně. Vzhledem k tomu, že region této komponenty má přidělené HWND, je možné do něj renderovat i DirectX obsah. Nevýhodou tohoto přístupu je již dříve zmíněné *airspace* omezení – přes oblast renderovanou pomocí DirectX již není možné nakreslit WPF komponentu. Toto řešení je popsáno například v článku [10].

Nakonec byla vybrána první možnost, především kvůli absenci omezení překrývání s ostatním obsahem renderovaným pomocí WPF. Použití druhého způsobu by znemožnilo přes 3D scénu vykreslovat ovládací prvky, což by mohlo být omezením do budoucna. Jak konkrétně je komponenta *D3DImage* ve 3D editoru použita, je popsáno v kapitole 4.3.

3.5. Použitý DirectX wrapper

Při integraci DirectX a WPF je třeba brát v úvahu fakt, že knihovna DirectX je psaná v unmanaged kódu. Naskytlo se několik možností, jak funkce této knihovny volat.

Jednou z možností bylo použít nějaký existující managed wrapper, který používání funkcí z DirectX značně zapouzdřuje a zpříjemňuje. Takových wrapperů je k dispozici několik, například SlimDX[5], Managed DirectX, XNA[6], Windows API Code Pack[7] nebo například Ogre[8].

Druhou možností by bylo implementovat wrapper vlastní. Tato možnost ale byla téměř okamžitě zamítnuta, neboť použitelné knihovny řešící tento problém již existují a vlastní wrapper by s největší pravděpodobností nepřinesl mnoho výhod navíc. Otázkou tedy bylo, který z existujících wrapperů použít.

Managed DirectX je oficiální DirectX wrapper vyvinutý společností Microsoft. Vývoj Managed DirectX je však v době vzniku této práce již delší dobu ukončen a Microsoft svoji pozornost zaměřil na framework XNA, který je určen především pro vývoj her. Managed DirectX je funkční a používá se v mnoha aplikacích, nicméně to, že se dále nevyvíjí, byl hlavní důvod, proč od něj bylo při použití v projektu AGE upuštěno.

Použití XNA frameworku by pro účely aplikace AGE bylo možné, nicméně i to bylo nakonec zamítnuto vzhledem k tomu, že u XNA není přímo podporován scénář renderování mimo vlastní okno. Navíc XNA samo spravuje tzv. herní smyčku [12], což se pro aplikaci AGE nehodí, jelikož není třeba scénu renderovat s frekvencí 60 snímků za sekundu, ale jen v případě, kdy se na ní něco změnilo. Obejít tuto vestavěnou herní smyčku je v XNA poněkud problematické a vyžaduje to reimplementovat některé třídy deklarované uvnitř XNA, což může být problém při upgradu na novou verzi XNA, jelikož vnitřní implementace těchto tříd se může změnit.

Dalším wrapperem, který by bylo možné použít, je Windows API Code Pack. Ten obsahuje třídy pro práci s DirectX 10 a 11. Tato altermativa byla ovšem též zamítnuta, neboť WPF ve své komponentě *D3DImage* počítá se zařízením Direct3D 9, které Windows API Code Pack neimplementuje, a navíc by aplikace AGE nefungovala na počítačích se staršími grafickými kartami, které DirectX 10 nepodporují.

Byla zvažována i možnost použít nějaký obecný grafický engine, který není cílen jen jako wrapper nad DirectX, ale implementuje grafické třídy a API nezávislé na konkrétní grafické knihovně, jež se stará o rendering. Takovým enginem je třeba Ogre, který umí vykreslovat jak pomocí DirectX, tak i pomocí OpenGL. Použití této knihovny bylo nakonec též zamítnuto, jelikož aplikace AGE by využila jen velmi malou část funkcionality této knihovny a bylo by nutné hledat vyhovující managed wrapper, neboť Ogre je též unmanaged knihovna.

Nakonec byl vybrán DirectX wrapper SlimDX, který implementuje managed třídy nad DirectX 9, ale zároveň podporuje i DirectX 10 a 11, což by mohlo být využito při budoucím vývoji 3D editoru

[5] Knihovnu SlimDX je možné stáhnout z webové adresy http://www.slimdx.org.
[6] Informace o Microsoft XNA Frameworku jsou na webové adrese http://creators.xna.com.
[7] Knihovna Windows API Code Pack je k dispozici na webové adrese http://code.msdn.microsoft.com/WindowsAPICodePack.
[8] Knihovnu Ogre je možné stáhnout z webové adresy http://www.ogre3d.org/.

aplikace AGE. Knihovna SlimDX implementuje jen tenkou vrstvu nad unmanaged rozhraními DirectX, což pro účely aplikace AGE postačuje.

3.6. Více pohledů na scénu

Jednou z možností 3D editoru by mělo být i zobrazení více pohledů na scénu najednou, například pohledy zepředu a z boku vedle sebe. Uživatel by měl mít též možnost měnit velikost těchto pohledů posunem lišty na jejich hranici.

První možností by bylo pro každý pohled vytvořit nové Direct3D zařízení a při posunu lišty pro změnu velikosti pohledů měnit velikost renderované plochy. Změna velikosti výstupu přináší poměrně velkou režii (je nutné uvolnit dosavadní výstupní buffer, tzv. surface, a naalokovat nový). Navíc každé Direct3D zařízení, které se inicializuje, zabere nezanedbatelnou část systémových prostředků. Na druhou stranu lišty pro oddělení a změnu velikosti pohledů lze vytvořit pomocí WPF a neporuší se tím *airspace* omezení – lišta bude mezi oblastmi renderovanými přes DirectX, nebude je překrývat.

Druhou možností je mít jen jedno Direct3D zařízení pro celou renderovanou plochu. DirectX umí nastavit v rámci svého výstupu tzv. viewport, což je obdélníková oblast na grafickém výstupu, na níž se bude vykreslovat. Stačí tedy stejnou scénu vykreslit vícekrát s nastavenou obdélníkovou oblastí, do níž se má vykreslovat, a pokaždé s použitím jiné kamery. Drobným problémem je vykreslování lišty pro oddělení a změnu velikosti pohledů. Pro vyhnutí se airspace omezení by bylo možné lišty vykreslovat přímo pomocí DirectX, díky čemuž by renderovaná oblast nemusela být překrytá WPF komponentami.

Vzhledem k tomu, že v předchozí kapitole bylo rozhodnuto použít komponentu *D3DImage*, která *airspace* omezení nemá, je ještě třetí možnost, jež se od předchozí druhé možnosti liší pouze tím, že pro lišty použijeme WPF komponentu (například GridSplitter), jelikož oblast renderovanou pomocí DirectX překrývat můžeme.

První možnost byla zavrhnuta kvůli vyšší režii s držením více Direct3D zařízení a protože airspace omezení bylo eliminováno použitím komponenty *D3DImage*, byla v tomto případě vybrána třetí možnost. Použít hotovou lištu z WPF je snazší než ji implementovat od začátku v DirectX.

3.7. Engine pro vyhodnocování výrazů

Druhým z vedlejších cílů této práce bylo vyzkoušet koncept výrazů pro definici vztahů mezi objekty. Každý objekt ve scéně může definovat různé vlastnosti – například pozici středu, šířku, výšku, materiál atd. Konkrétní vlastnosti závisí samozřejmě na typu daného objektu. Hodnota každé vlastnosti každého objektu by mohla být definovanána jako výraz, což znamená, že hodnoty vlastností by nebyly dané pevně, ale mohly by se měnit v závislosti na vlastnostech ostatních objektů či jiných parametrů.

Jednoduchým příkladem použití může být například požadavek uživatele, aby jeden objekt scény byl vždy 2 jednotky napravo od objektu druhého. To by se dalo řešit například parent-child vztahem, druhý objekt by zkrátka byl pozicován relativně vůči tomu prvnímu. Tento přístup však není tak silný, neboť s pomocí konceptu výrazů lze simulovat i složitější scénáře.

Zajímavějším příkladem může být například situace, kdy uživatel má ve scéně 2 krychle, mezi ně vloží například kouli a přeje si, aby střed této koule byl vždy přesně mezi těmito krychlemi a aby

koule procházela středem těchto krychlí. Stačí pozici koule a její poloměr zapsat jako vzorec, který příslušné hodnoty vypočítá podle pozic dvou krychlí. V případě změny pozice libovolné krychle se automaticky upraví i velikost a pozice koule.

Jak by tyto výrazy vypadaly? Nechť krychle se jmenují A a B. Poloměr koule může být definován výrazem **center($A.Position, $B.Position)**, poloměr **distance($A.Position, $B.Position) / 2**. Operátor **$** značí, že jde o objekt ve scéně, je to opatření kvůli rozšiřitelnosti do budoucna, které by mělo zamezit možné kolizi názvů identifikátorů, pokud by se výrazy například rozšířily o podporu proměnných apod. Funkce **center** a **distance** vrací střed úsečky resp. vzdálenost pro dané dva body.

Od výrazů se očekává podpora jednoduchých (řetězec, desetinné číslo) i složených datových typů (dvojrozměrný vektor, trojrozměrný vektor, ARGB barva atd.) a funkcí. Objekty ve scéně by měly být schopny odebírat notifikace o změně objektů, na nichž některé z jejich vlastností závisí.

Přístupů k řešení enginu a vyhodnocování výrazů bylo poměrně mnoho, byly z nich vybrány postupně následující tři. Prvním nápadem bylo uchovávat všechny vlastnosti objektů scény jako řetězce a pro každý výraz si pamatovat kolekci objektů, které na něm závisí, aby mohly být při změně tohoto výrazu tyto objekty upozorněny. Při každém dotazu na hodnotu vlastnosti by se výraz rozparsoval a vyhodnotil. Toto řešení má mnoho nevýhod, největším problémem je bezesporu jeho zbytečná náročnost – parsovat výraz při každém dotazu by velmi zdržovalo. Tuto nevýhodu je možné řešit cacheováním hodnoty výrazu.

Druhou možností bylo výraz reprezentovat jako strom objektů. Toto řešení se používá poměrně běžně, vyhodnocení je též mnohem rychlejší, jelikož se výraz nemusí parsovat pokaždé. Každý uzel ve stromě výrazu by obsahoval kolekci objektů, které na jeho hodnotě závisí, aby se daly snadno implementovat notifikace, a dále by zde bylo možné implementovat cacheování, čímž by se vyhodnocování mohlo opět mírně urychlit.

Třetí možností bylo celý strom výrazu ještě zapouzdřit do speciální třídy, čímž se umožní mechanismus pro cacheování a notifikace z každého uzlu stromu výrazu přesunout pouze do objektu na nejvyšší úrovni, který celý strom zapouzdřuje. To se nakonec ukázalo jako nejpraktičtější řešení, neboť pokud by si každý uzel měl držet kolekci objektů, které na něm závisí, a svou cacheovanou hodnotu, zabralo by to více paměti a úspora času při vyhodnocování (např. tím, že by se přepočítával jen podstrom, který se změnil) by ve většině případů pravděpodobně nebyla velká tak, aby se to vyplatilo i přes zvýšené paměťové nároky.

Při implementaci výrazů je třeba vzít v úvahu, že výrazy by se měly vyhodnocovat až v okamžiku, kdy je jejich hodnota požadována, a ne přímo při parsování. To by mohlo působit problémy v případě, kdy se scéna například načítá ze souboru a některému objektu nastavujeme do vlastnosti výraz závisející na objektu, který ještě nebyl načten a inicializován.

Další problém, na nějž je třeba dát pozor, jsou hromadné úpravy více vlastností objektů. Je vhodné mít možnost notifikace o změnách dočasně vypnout a následně opět zapnout. Během hromadných úprav vlastností objektů (například při přejmenování či smazání objektu ve scéně, což vyžaduje projít všechny výrazy a vhodně je opravit) je vhodné notifikace pozastavit a obnovit až ve chvíli, kdy je celá akce hotová.

Podrobný popis implementace enginu pro vyhodnocování výrazů je v kapitole 4.5.

3.8. Objekty a reprezentace scény

Dalším bodem analýzy byl návrh reprezentace scény a objektů, které scéna obsahuje. Je každopádně nutné definovat abstraktní třídu nebo rozhraní, z něhož bude muset být odvozen každý objekt ve scéně.

Prvním nápadem bylo definovat rozhraní, které by musely všechny objekty ve scéně implementovat. Toto rozhraní by definovalo kolekci potomků, jednoznačné ID objektu a referenci na rodičovský objekt. Velmi záhy se ale ukázalo, že lépe než rozhraní bude použít abstraktní třídu, jelikož už v samotném základním objektu bude třeba hlídat například unikátnost ID a zda-li neobsahuje nepovolené znaky (ID smí obsahovat jen čísla, podtržítko a písmena anglické abecedy, navíc smí začínat jen písmenem; identifikátory začínající podtržítkem jsou vyhrazeny pro interní potřeby aplikace a pro dočasné objekty).

Dále bylo třeba rozhodnout, zda-li se základní abstraktní třída a další fundamentální třídy z ní dědící (např. třída reprezentující scénu) budou definovat přímo v assembly 3D editoru, anebo se vyčlení do samostatné assembly, aby je případně bylo možné včetně enginu pro vyhodnocování výrazů používat i z jiných modulů. Tato volba nakonec byla uskutečněna, neboť strom objektů a výrazy se mohou velmi dobře využít například v modulu pro 2D grafiku.

O tom, jak je scéna v modulu 3D editoru implementována, pojednává kapitola 4.6.

3.9. Objekty a modifikátory

Základní objekty, které lze do scény přidat, je samozřejmě často třeba upravovat, například měnit pozice některých vrcholů či stěn apod. Existuje několik přístupů, jak toto řešit.

První zvažovaná možnost byla implementovat pouze jeden objekt scény, který by si pamatoval seznam vrcholů a stěn. V okamžiku, kdy by uživatel přidával do scény nějaký základní objekt, například hranol či kouli, by se pouze seznam vrcholů a stěn jednorázově naplnil. Uživatel by pak mohl volně hýbat s jednotlivými vrcholy či stěnami. Tento přístup je výhodný svou relativní snadností implementace vzhledem ke druhému řešení.

Druhá zvažovaná možnost, která byla nakonec vybrána, je koncept objektů s modifikátory. Je definováno několik základních objektů (krychle, polygon, n-boký hranol atd.), jež umí vygenerovat seznam vrcholů a stěn podle nastavených parametrů (například počet stěn hranolu apod.). Na každý objekt nebo jeho určitou část (množinu vrcholů či stěn) je pak možné aplikovat tzv. modifikátory. Modifikátor může například vybraný objekt nebo jeho část posunout, otočit kolem určité osy, nastavit vybraný materiál atd. U aplikovaných modifikátorů záleží samozřejmě na pořadí. Při renderování se pak vygeneruje kolekce vrcholů a stěn tělesa. Tato kolekce se postupně předá všem modifikátorům, které ji upraví dle svého nastavení. Výstup posledního modifikátoru se pak vyrenderuje.

Výhodou vybraného druhého přístupu je možnost zpětně upravovat původní objekt či jeho modifikátory. Mejme například krychli, na jejíž dva vrcholy je aplikován modifikátor otočení o 45 stupňů podle osy X, následně je na celou krychli aplikován modifikátor posunutí a dále ještě například otočení o 90 stupňů podle osy Z. Vzhledem k tomu, že ani jeden modifikátor nemění počet vrcholů ani stěn tělesa (jen některé vrcholy jsou vychýlené z původní polohy modifikátorem otočení), lze

měnit například osy či úhly, podle nichž se otáčí, anebo vzdálenost, o níž se posouvá. To přináší některé poměrně zajímavé možnosti.

Samozřejmě modifikátory nemusí vrcholy jen posouvat či otáčet, ale mohou například generovat nové stěny, mazat vrcholy, měnit materiál, barvu či mapovací souřadnice pro textury. Vzhledem k tomu, že modifikátory mohou mít, stejně jako objekty ve scéně, vlastnosti definované jako výrazy závisející na dalších objektech, lze tím opět získat nové možnosti.

Při implementaci je též vhodné počítat s tím, že modifikátory je možné akumulovat. Máme-li na jednom objektu za sebou například dva modifikátory posunutí, lze je sloučit do jednoho. Obdobně se dají skládat po sobě jdoucí otočení kolem stejné osy. Některé modifikátory, například i již výše zmíněné posunutí, lze na těleso aplikovat přímo přičtením vektoru posunutí k pozici tělesa.

Konkrétní implementace konceptu modifikátorů je popsána v kapitole 4.7.

3.10. Nástroje editoru a dekorátory

V okamžiku, kdy je již vyřešen rendering scény a deklarovány základní objekty scény, je třeba řešit například otázku, jak s objekty manipulovat v editoru pomocí myši či klávesnice. Je samozřejmě několik možností, jak toto řešit, lze je demonstrovat například na situaci, kdy přetažením objektu myší tento objekt přesouváme z místa na místo.

Asi nejtriviálnějším řešením by bylo přímo do obsluhy událostí *MouseDown*, *MouseMove* a *MouseUp* v okně editoru implementovat požadovanou logiku. Toto řešení ale není vhodné, jelikož objekty je třeba posouvat, otáčet, zvětšovat a vše se dělá pohybem myši. V zájmu zachování přehlednosti kódu a jeho logického členění je nutné jednotlivé úkony rozdělit do více tříd a vytvořit pro ně nějaká rozhraní. Bylo tedy navrženo řešení, které implementuje koncept nástrojů editoru.

Nástroj editoru je reprezentován třídou, která odebírá určité události okna. Ve chvíli, kdy nástroj zjistí, že uživatel začal provádět akci, která odpovídá danému nástroji (například drží klávesu Control a stisknul tlačítko myši), nástroj oznámí editoru, že je aktivní. Od této chvíle tento nástroj dál obsluhuje události z editoru, například při pohybu myši pohybuje vybraným objektem. V okamžiku, kdy nástroj detekuje, že uživatel akci ukončil (například při uvolnění tlačítka myši), dokončí prováděnou akci a oznámí editoru, že přestal být aktivní. Po dobu, kdy je některý nástroj aktivní, se nemohou další nástroje aktivovat, i když detekují začátek akce, kterou obsluhují.

Během akcí, které nástroje řídí, je často nutné do scény přikreslit nějaké ovládací prvky, například směrové šipky pro jednotlivé osy, pomocí nichž může uživatel myší objekty posouvat v daném směru. Každý nástroj tedy má metodu, která vrací seznam tzv. dekorátorů. Podle toho, v jakém je nástroj aktuálně stavu, vrací objekt, který se stará o vykreslení daného ovládacího prvku do scény.

Pomocí dekorátorů je například realizováno ohraničení označených objektů, směrové šipky při posouvání, otáčení či zvětšování objektů a podobně. Vzhledem k tomu, že koncept nástrojů opět může být užitečný i pro ostatní moduly, jeho základní třídy jsou definovány přímo v assembly aplikace AGE.

Podrobný popis implementace dekorátorů je obsažen v kapitolách 4.10 a 4.11.

4. Implementace aplikace AGE a editoru 3D scén

V následující části je podrobně popsána implementace modulu 3D editoru a aplikace AGE. Aplikace je rozdělena do několika assembiles, v následující tabulce je jejich stručný popis.

Assembly	Popis
Age.Application.Main	Hlavní assembly aplikace AGE, hostování modulů a uživatelské rozhraní hlavního okna
Age.Core.Utils	Pomocné třídy a funkce
Age.Core.Scene.Tree	Engine pro vyhodnocování výrazů
Age.Core.Scene.Scene3D	Třídy a objekty scény specifické pro 3D, např. scéna, materiály, modifikátory apod.
Age.Core.Render.Render3D	Třídy zajišťující renderování 3D grafiky a správu vertex a index bufferů.
Age.Core.Objects.Objects3D	Implementace primitivních 3D objektů ve scéně.
Age.Application.Dev.Editor3D	Modul a uživatelské rozhraní 3D editoru, nástroje editoru atd.

Tabulka 1: Assembly AGE a 3D editoru a jejich popis

4.1. Hlavní okno aplikace AGE

Aplikace AGE je celá implementována v assembly **Age.Application.Main**. Tato assembly definuje několik jmenných prostorů, jež jsou popsány v následující tabulce.

Jmenný prostor	Popis
Age.Application.Main.Config	Obsahuje deklarace konfiguračních sekcí a elementů pro konfigurační systém .NET Frameworku
Age.Application.Main.Modules	Obsahuje třídy infrastruktury rozšiřujících modulů aplikace
Age.Application.Main.Serialization	Obsahuje abstraktní třídu definující rozhraní tříd pro načítání a ukládání souborů a získávání metadat ze souborů
Age.Application.Main.UI	Obsahuje třídy implementující MDI funkcionalitu a základní třídu reprezentující nástroje editoru
Age.Application.Main.Utils	Obsahuje třídy s různými pomocnými funkcemi
Age.Application.Main.Windows	Obsahuje sdílená dialogová okna, která mohou jednotlivé moduly používat

Tabulka 2: Jmenné prostory v assembly hlavní aplikace a jejich význam

Hlavní okno aplikace reprezentuje třída **MainWindow**. Toto hlavní okno obsahuje komponentu Ribbon[9] s definovanými základními tlačítky pro vytvoření nového souboru, otevření souboru z disku, uložení souboru apod. Dále toto okno obsahuje komponentu *TabContainer*, která ve svých záložkách hostuje dceřinná MDI okna. Každé takové okno musí dědit ze třídy **MdiChild** deklarované ve jmenném prostoru **Age.Application.Main.UI**.

Při startu hlavní aplikace se načte konfigurační soubor. Formát konfiguračního souboru je popsán v kapitole Konfigurační soubor aplikace v uživatelské dokumentaci aplikace. Používá se standardní konfigurační systém implementovaný v .NET Frameworku [11], konfigurace je rozšířena s použitím vlastní konfigurační sekce **AgeConfigurationSection**. Z konfigurace se získají všechny instalované moduly a ty se dynamicky načtou pomocí volání *System.Reflection.Assembly.LoadFile*.

[9] V aplikaci se používá komponenta WPF Ribbon, která by měla být v budoucnu součástí .NET Frameworku. Její pre-release verze je ke stažení z webové adresy http://msdn.microsoft.com/cs-cz/office/aa973809(en-us).aspx.

Pak se pomocí volání *Activator.CreateInstance* vytvoří instance modulu (název třídy je stejně jako cesta k assembly uložen v konfiguračním souboru).

Každý modul aplikace AGE musí dědit z abstraktní třídy **AgeModule** deklarované ve jmenném prostoru **Age.Application.Main.Modules**. Tato třída deklaruje abstraktní metodu **CreateEditorWindow**, kterou je třeba při implementaci vlastního modulu přepsat, aby vracela novou instanci dceřinného MDI okna, které bude posléze umístěno do hlavního okna aplikace. Dále musí každý modul implementovat abstraktní vlastnost **SupportedFormats**, která vrací seznam podporovaných formátů, jak ukazuje následující ukázka kódu.

```
public override IEnumerable<SupportedFormat> SupportedFormats {
    get {
        yield return new SupportedFormat(this, "Scéna se 3D objekty",
            new[] { "a3d" }, typeof(A3dSerializer));
    }
}
```

Vlastnost **SupportedFormats** má za úkol vrátit seznam instancí třídy **SupportedFormat**, která reprezentuje informace o podporovaném formátu. Prvním parametrem jejího konstruktoru je instance modulu aplikace AGE, k němuž se formát vztahuje. Druhým parametrem je popis pro zobrazení v uživatelském rozhraní (například v dialogu pro otevření či uložení souboru). Třetím parametrem je pole přípon souborů, které tento typ souboru identifikují (může jich být více, například formát JPEG podporuje přípony *jpg* i *jpeg* apod.).

Posledním parametrem konstruktoru je typ třídy, která se stará o načítání, ukládání a získávání metadat z tohoto typu souboru. Tato třída musí dědit ze třídy **AgeSerializer** deklarované ve jmenném prostoru **Age.Application.Main.Serialization**. Podrobnější popis ukládání a načítání souborů je v kapitole 4.13.

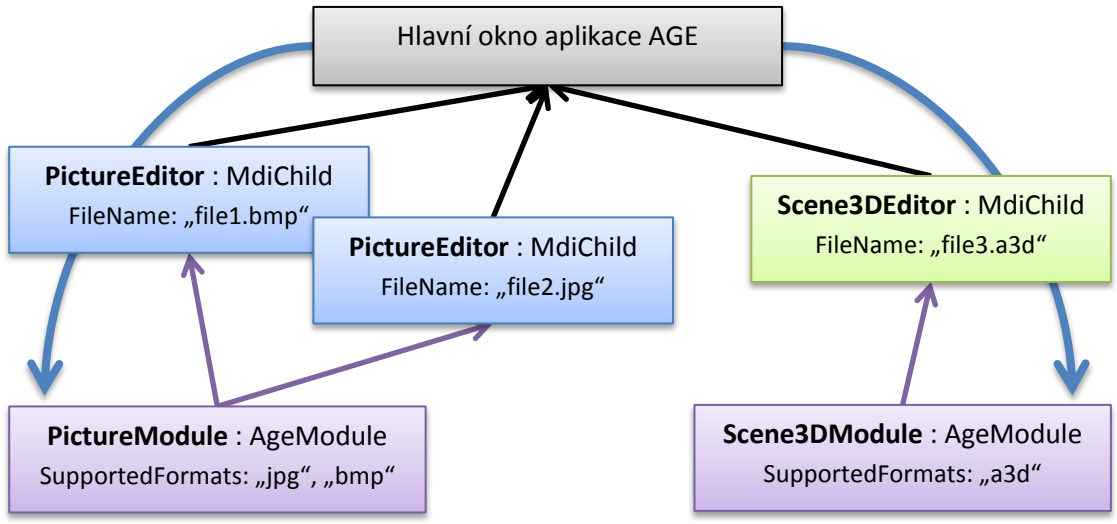
Modul 3D editoru je obsažen v assembly **Age.Application.Dev.Editor3D**.

4.2. Dceřinné MDI okno modulu

Jak bylo řečeno v předchozí kapitole, každé dceřinné MDI okno musí dědit ze třídy **MdiChild**. Tato třída deklaruje několik metod, které je třeba implementovat. Patří mezi ně například metody **ExecuteCutCommand**, **ExecuteCopyCommand** a **ExecutePasteCommand**, které jsou hlavním oknem zavolány v případě, kdy jsou použita tlačítka nebo klávesové zkratky pro akce schránky.

Dále tato třída definuje abstraktní metody **LoadFile** a **SaveFile**, které se starají o načítání a ukládání souborů. Tyto metody nemají za úkol zobrazovat dialog pro výběr souboru pro načtení či uložení, o to se stará hostitelská aplikace. Dostanou již rovnou cestu k souboru a starají se tedy jen o uložení či načtení.

Třída **MdiChild** též obsahuje kolekci **RibbonTabs**, která vrací záložky do Ribbonu, které se mají zobrazit v případě, že je MDI okno aktivní.



Obrázek 1: Proces vytvoření editoru pro otevíraný soubor

Na tomto obrázku je znázorněn proces otevření souboru v aplikaci AGE. V aplikaci jsou načteny dva fiktivní moduly – PictureModule a Scene3DModule. Po kliknutí na tlačítko pro otevření souboru a vybrání cesty hlavní okno aplikace AGE dle přípony souboru určí modul, který tento soubor podporuje. Pokud je modul nalezen, zavolá se na něm metoda **InstantiateEditorWindow**, která vytvoří instanci dceřinného MDI okna s editorem a toto okno umístí do panelu záložek v hlavním okně aplikace. Na tomto okně je následně zavolána metoda **LoadFile**, čímž se do editoru načte otevíraný soubor. Během umístění MDI okna do hlavního okna aplikace AGE se též do panelu nástrojů Ribbon přidají záložky, které definuje třída MDI okna.

4.3. Renderování 3D scény uvnitř aplikace

Pro vykreslování 3D grafiky pomocí knihovny DirectX je v modulu 3D editoru implementována komponenta **DirectXWPFHost**, která dědí z WPF třídy **Canvas** a vykresluje scénu pomocí knihovny **SlimDX**.

V konstruktoru třída **DirectXWPFHost** jako své potomky vygeneruje komponenty **Image** a **D3DImage**, přičemž se postará o jejich správnou inicializaci. Při zavolání metody **Initialize** této třídy je pomocí knihovny **SlimDX** vytvořeno zařízení **Direct3D** nebo **Direct3DEx** podle toho, na jakém operačním systému je aplikace provozována.

Na systémech *Windows Vista* a novějších je totiž použití **Direct3DEx** několikanásobně rychlejší, jelikož se využívá technologii WDDM [13]. Na *Windows XP* a starších systémech naproti tomu není **Direct3DEx** k dispozici, takže je nutné použít klasické **Direct3D** zařízení. Tyto rozdíly mezi platformami jsou ošetřeny uvnitř komponenty a navenek se s ní pracuje jednotně.

Komponenta **D3DImage**, která integraci DirectX a WPF zajišťuje, obsahuje velmi důležitou událost *IsFrontBufferAvailableChanged*. Ta je vyvolána v případě změny stavu front bufferu[10]. To může nastat například při ztrátě zařízení (například při přechodu počítače do režimu spánku

[10] Obvykle se používá renderování do dvou bufferů – jeden buffer je zobrazen na obrazovce, do druhého se vykresluje. Ve chvíli, kdy je vykreslování hotovo, se buffery prohodí (tzv. operace *flip*). Buffer, který se zobrazuje, je označován jako *front butter*; buffer, do nějž se právě vykresluje, se označuje jako *back buffer*.

a následném „probuzení“, nebo v případě spuštění jiné aplikace v režimu celé obrazovky). V takovém případě přestane být front buffer dostupný a je nutné na tuto situaci odpovídajícím způsobem reagovat. Typicky je třeba Direct3D zařízení resetovat a alokovat nový front buffer.

Další událost, kterou je třeba ošetřit, je událost *CompositionTarget.Rendering*, jež nastává před vyrenderováním grafického obsahu WPF aplikace. V této události je potřeba překreslit scénu v komponentě **D3DImage**, pokud to okolnosti vyžadují (v našem konkrétním případě pouze pokud se scéna změnila).

Obě výše uvedené činnosti jsou již implementovány v komponentě **DirectXWPFHost**. Pro použití této komponenty na ní stačí zavolat metodu **Initialize**. V případě, že této metodě předáme jako argument hodnotu **true**, bude se obsah komponenty překreslovat vždy při zavolání *CompositionTarget.Rendering*. V případě předání hodnoty **false** bude scéna vykreslována jen na vyžádání, a to po zavolání metody **Invalidate** na komponentě.

Při použití komponenty **DirectXWPFHost** stačí v obsluze události **RenderFrame** vykreslit obsah scény.

V modulu editoru 3D je dále definována komponenta **Scene3DView**. Tato komponenta dědí z třídy **DirectXWPFHost** a přepisuje mechanismus renderování scény tak, aby bylo možné vykreslit více pohledů na scénu vedle sebe podle nastavené kolekce **Viewports**, kterou tato komponenta obsahuje. Obsluha události **RenderFrame** třídy **Scene3DView** se při každém renderování scény zavolá jednou pro každý pohled na scénu, přičemž na DirectX zařízení je již přednastavena obdélníková oblast okna, do níž se má vykreslovat, a příslušné matice kamery a projekce, jež se mají v daném pohledu použít.

Ve dceřinném MDI okně 3D editoru je umístěna komponenta **Scene3DView** tak, aby vyplnila celou plochu okna. Přes ní jsou umístěny dvě komponenty **GridSplitter** (jedna vertikálně a jedna horizontálně), čímž vznikají 4 pohledy na scénu. Při posunu komponenty **GridSplitter** hlavní okno aktualizuje kolekci **Viewports** obsahující rozměry jednotlivých pohledů a jejich kamery.

4.4. Reprezentace 3D scény

Jak již bylo naznačeno v kapitole 3.8, třídy reprezentující strom objektů ve scéně a třídy reprezentující výrazy by měly být vyčleněny do samostatné assembly, aby je bylo možné použít i z dalších modulů. Proto jsou následující třídy rozděleny do dvou assemblies – **Age.Core.Scene.Tree** a **Age.Core.Scene.Scene3D**.

Assembly **Age.Core.Scene.Tree** obsahuje základní třídy reprezentující výrazy a obecné třídy stromu scény. Assembly **Age.Core.Scene.Scene3D** pak obsahuje implementace a třídy specifické pro modul 3D editoru.

Scéna je navržena jako stromová struktura objektů typu **TreeObject**. Třída **TreeObject** je definována ve jmenném prostoru **Age.Core.Scene.Tree** a obsahuje mimo jiné vlastnosti **Id**, **Parent** a kolekci **Children**. První vlastnost uchovává unikátní ID objektu v rámci scény, druhá vlastnost obsahuje referenci na rodičovský objekt. Kolekce **Children** je typu **TreeObjectCollection** a obsahuje potomky daného objektu.

Kořenovým objektem stromu je vždy instance třídy **TreeRoot** nebo třídy, která z ní dědí. Třída **TreeRoot** dědí ze třídy **TreeObject** a obsahuje funkce pro přejmenování objektů ve scéně, rekurzívní procházení či vyhledávání objektů dle jejich identifikátoru. Mimo jiné umožňuje reagovat na události přidávání, změny či odebírání prvků uvnitř celého stromu, a to pomocí událostí **Inserted**, **Updated** a **Deleted**. Události přidávání a odebírání jsou vyvolávány kolekcí **TreeObjectCollection**, která byla zmíněna již dříve. Událost změny objektu je vyvolávána v případě, že se změní některá z vlastností tohoto objektu, o čemž bude řeč v následující kapitole.

V modulu 3D editoru se tyto tři události využívají pro aktualizaci vertex bufferu držícího seznam trojúhelníků objektů ve scéně. Při změně objektu je třeba trojúhelníky vygenerovat znovu, jelikož se mohl změnit tvar tohoto objektu. Při přidání objektu do scény je třeba jeho trojúhelníky do vertex bufferu přidat, obdobně při odebrání objektu ze scény je třeba jeho trojúhelníky z vertex bufferu odstranit.

Vzhledem k tomu, že objekty ve 3D scéně a i 3D scéna samotná obsahují některé vlastnosti navíc oproti obecné implementaci v assembly **Age.Core.Scene.Tree**, jsou v assembly **Age.Core.Scene.Scene3D** definovány třídy **TreeObject3D** (dědí z **TreeObject**) a **Scene3D** (dědí z **TreeRoot**). Třída **Scene3D** definuje navíc oproti třídě **TreeRoot** například knihovnu materiálů a textur, které se na objektech scény používají. To jsou již záležitosti specifické pro 3D editor. Třída **TreeObject3D** navíc deklaruje metody pro generování trojúhelníků k vyrenderování a dále například seznam modifikátorů.

4.5. Engine pro vyhodnocování výrazů

Jak již bylo dříve několikrát řečeno, každý objekt ve scéně může definovat vlastnosti, jejichž hodnotou je výraz. V kapitole 3.7 bylo popsáno, že výraz by měl být implementován jako strom zapouzdřený do nějaké třídy, která bude obstarávat cacheování a notifikace změny hodnoty daného výrazu. Veškeré třídy týkající se výrazů jsou obecné a nezávislé na 3D editoru, tudíž jsou deklarovány v assembly **Age.Core.Scene.Tree**.

Jednotlivé třídy reprezentující uzly stromu výrazu jsou popsány v následující tabulce.

Název třídy či rozhraní	Význam
ITreeExpression	Obecný předek všech uzlů výrazu
TreeExpression<T>	Silně typovaný obecný předek všech výrazů
ConstantTreeExpression<T>	Výraz obsahující konstantní hodnotu
UnaryOperatorTreeExpression<T>	Obecný předek uzlů reprezentujících unární operátor
NumericNegativeTreeExpression	Unární mínus pro číselné datové typy
BinaryOperatorTreeExpression<T>	Obecná předek uzlů reprezentujících binární operátor
NumericAddTreeExpression	Sčítání pro číselné datové typy
NumericSubtractTreeExpression	Odčítání pro číselné datové typy
NumericMultiplyTreeExpression	Násobení pro číselné datové typy
NumericDivideTreeExpression	Dělení pro číselné datové typy
ObjectAccessTreeExpression	Vrací objekt ze scény s daným ID
PropertyAccessTreeExpression<T>	Vrací hodnotu dané vlastnosti daného objektu
FunctionCallTreeExpression<T>	Vrací výsledek volání dané funkce s danými argumenty

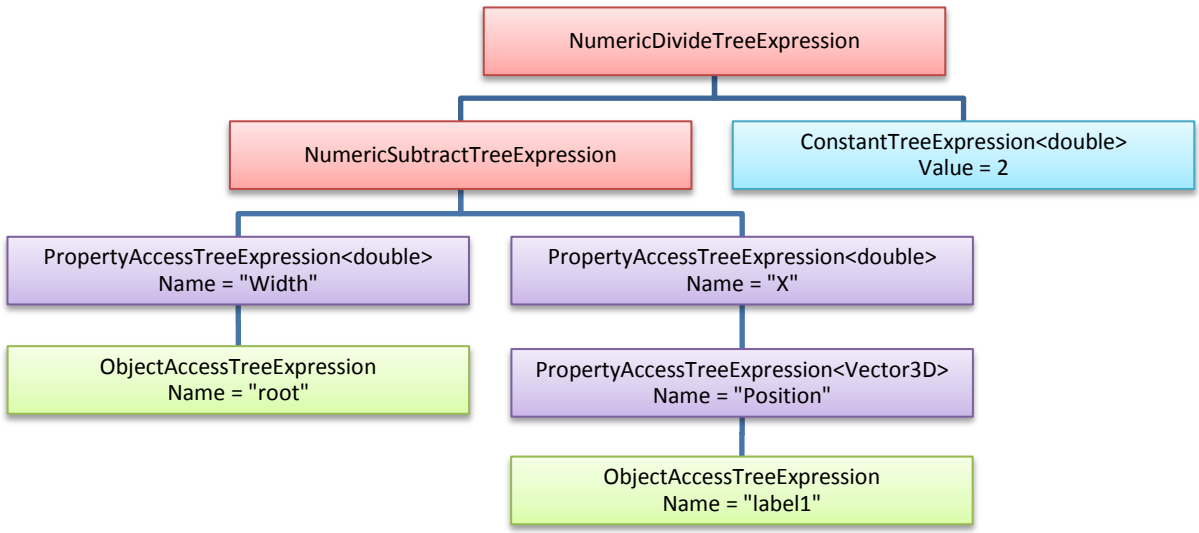
Tabulka 3: Třídy reprezentující jednotlivé uzly výrazů

Jednou z nejdůležitějších tříd je třída **ExpressionParser**, která obsahuje statickou metodu **Parse**, jež z textové reprezentace strom výrazu zkonstruuje. Syntaxe výrazů je popsána v kapitole Provázání objektů pomocí výrazů v uživatelské dokumentaci.

Kromě typů uvedených v předchozí tabulce existuje též třída **UserTreeExpression<T>**, která celý výraz zapouzdřuje a obsahuje obě reprezentace výrazu – stromovou i textovou. Tato třída obstarává též cacheování vypočítané hodnoty výrazu, a dále obsahuje kolekci **NotifiedExpressions**, jež obsahuje seznam výrazů objektů scény, které je třeba upozornit v případě, že se hodnota tohoto výrazu změní. Samotnou notifikaci zajišťuje metoda **SetDirty**, která zneplatní cacheovanou hodnotu výrazu, projde kolekci notifikovaných výrazů a zavolá **SetDirty** na nich. Tím se projde de facto celý strom závislostí a výrazy, jejichž hodnota se změnila, jsou aktualizovány.

Třída **UserTreeExpression<T>** dovede při změně hodnoty též notifikovat vlastníka tohoto výrazu, totiž objekt scény (**TreeObject**), který daný výraz obsahuje, a tím ve scéně vyvolat událost **Updated** pro daný objekt. Díky tomu se při změně vlastnosti objektu ve 3D editoru okamžitě přegeneruje obsah vertex bufferu pro daný objekt, tato akce je totiž na událost **Updated** vázána.

Následující obrázek znázorňuje strom výrazu **($root.Width - $label1.Position.X) / 2**.



Obrázek 2: Příklad stromu výrazu

4.6. Implementace konkrétních 3D objektů ve scéně

Na tomto místě by bylo vhodné ukázat, jakým způsobem jsou reprezentovány vestavěné 3D objekty ve scéně. 3D editor v základu podporuje několik základních objektů, například krychli, kouli, jehlan, hranol, mnohoúhelník nebo graf 2D funkce. Tato tělesa jsou deklarována v assembly **Age.Core.Objects.Objects3D**.

Základem pro tvorbu vlastního objektu je vytvoření třídy, jež dědí ze třídy **TreeObject3D**. Vlastnosti objektu obsahující výraz jsou deklarovány následující formou.

```
[AgeProperty("Šířka", "PropertyGridSimpleRangeEditor", 0.0, 10000.0)]
public UserTreeExpression<double> Width { get; set; }
```

Samotná vlastnost je typu **UserTreeExpression<double>**, což reprezentuje výraz, jež vrací hodnotu typu **System.Double**. Kromě toho je vlastnost označena atributem **AgePropertyAttribute**, který specifikuje další informace o této vlastnosti. První parametr je text, který se bude zobrazovat v uživatelském rozhraní. Druhý parametr je název datového typu editoru, který se použije pro editaci této vlastnosti (je možné použít i přetížení, kde se název typu nepředává jako řetězec, ale jako hodnota **System.Type**). Další parametry jsou volitelné a specifikují další nastavení editoru, v tomto konkrétním případě minimální a maximální číselnou hodnotu výrazu.

Komponenta **TreeObjectPropertyGrid**, jíž 3D editor implementuje a která je určena pro nastavování vlastností objektů v uživatelském rozhraní, je blíže popsána v kapitole 4.9.

Všechny vlastnosti typu **UserTreeExpression** musí být inicializovány v konstruktoru třídy, například podle následující ukázky konstruktoru třídy **Cube**.

```
public Cube(Scene3D scene) : base(scene)
{
    Width = new UserTreeExpression<double>(this, "1", scene.EvaluationContext);
}
```

Třída **TreeObject3D** nemá bezparametrický konstruktor, takže výše uvedený konstruktor je povinný i v případě, pokud by objekt žádné vlastnosti typu **UserTreeExpression** neobsahoval. Vlastnost se inicializuje vytvořením nové instance třídy **UserTreeExpression**.

První parametr je vlastník výrazu, tj. objekt, který výraz obsahuje. Tento vlastník je v případě změny hodnoty notifikován (je na něm zavolána metoda **SetDirty**) a scéna je informována o tom, že se tento objekt změnil. Tyto notifikace scény lze před nastavením hodnoty výrazu pozastavit zavoláním metody **BeginUpdate** na dotyčném objektu scény a následně tyto notifikace opět obnovit zavoláním metody **EndUpdate**. To je užitečné například v případě, že hodláme měnit několik výrazů tohoto objektu po sobě a není vhodné po každé změně přegenerovávat vertex a index buffery.

Druhý parametr konstruktoru třídy **UserTreeExpression<T>** je výchozí hodnota výrazu, v tomto případě výchozí šířka krychle je 1. Hodnotu lze určit přímo (v případě ukázky by místo řetězce byla jednička předána jako hodnota typu **double**), nebo jako řetězec s textem daného výrazu.

Poslední parametr konstruktoru je tzv. vyhodnocovací kontext, což je instance třídy **EvaluationContext**, která obsahuje informace o stavu prostředí, například funkce, které je možné z výrazů volat. O vyhodnocovacím kontextu, volání funkcí z výrazů a povolených datových typech pojednává kapitola 4.8.

Pro získání hodnoty výrazu na tento výraz stačí zavolat metodu **Evaluate** anebo jej přetypovat na příslušný datový typ (např. **UserTreeExpression<double>** na typ **double**), což provede prakticky totéž. Nastavení nové hodnoty výrazu lze provést dvěma způsoby – buď nastavením

vlastnosti **Text**, anebo zavoláním metody **SetValue**, které lze předat přímo cílovou hodnotu výrazu. V tomto druhém případě bude výraz nastaven jako **ConstantTreeExpression**.

4.7. Generování vertexů a indexů

Modul 3D editoru mimo jiné deklaruje třídu **VertexIndexBufferManager**, která se stará o vytváření vertex a index bufferů na DirectX zařízení a přidělování místa v těchto bufferech jednotlivým objektům scény.

Třída **TreeObject3D** implementuje rozhraní **IVertexIndexGenerator**, které definuje metody **GetMeshPartCountCore**, **GetVerticesCountCore**, **GetIndicesCountCore**, **GetVerticesCore** a **GetIndicesCore**. V následující tabulce jsou uvedeny významy těchto metod.

Název metody	Popis
GetMeshPartCountCore()	Jeden objekt může obsahovat několik částí. Tato metoda vrací počet částí tohoto objektu.
GetVerticesCountCore(int meshPart)	Vrací počet vertexů dané části modelu.
GetIndicesCountCore(int meshPart)	Vrací počet indexů dané části modelu. Pokud tato metoda vrací hodnotu 0, objekt nepoužívá index buffer.
GetVerticesCore(int meshPart)	Vrací kolekci vertexů dané části objektu.
GetIndicesCore(int meshPart)	Vrací kolekci indexů dané části objektu.

Tabulka 4: Popis metod deklarovaných v rozhraní IVertexIndexGenerator

Jak již bylo řečeno v kapitole 3.9, na každý objekt nebo jeho části můžeme navíc aplikovat v přesně definovaném pořadí různé modifikátory. Modifikátor je třída odvozená od třídy **Age.Core.Scene.Scene3D.Modifiers.Modifier**, která taktéž implementuje **IVertexIndexGenerator**.

Třída **TreeObject3D** obsahuje tyto modifikátory udržuje ve vlastnosti **Modifiers**, které se na objekt aplikují v pořadí, jaké tento seznam udává. Pokud se tato kolekce nebo její libovolný prvek změní, je je objekt scény o této situaci notifikován stejným způsobem, jako kdyby se změnil některý z jeho výrazů. Opět dochází i k vyvolání události **Updated** ve scéně, pokud tyto notifikace nejsou pozastaveny.

Třída **TreeObject3D** kromě výše uvedených metod deklaruje též metody **GetMeshPartCount**, **GetVerticesCount**, **GetIndicesCount**, **GetVertices** a **GetIndices**. Tyto metody se starají o cacheování, pokud tedy vertexy a indexy mají již cacheované, jen je vrátí. V případě, že cache je prázdná, jsou dvě možnosti.

V případě, že objekt ve scéně nemá žádné modifikátory, je pro naplnění cache z každé výše uvedené metody zavolána stejnojmenná metoda s příponou **Core**. V případě, že objekt modifikátory má, je stejnojmenná metoda s příponou **Core** zavolána na posledním modifikátoru.

Každý modifikátor má mimo jiné vlastnost **PreviousVertexIndexGenerator**, která vrací předchozí modifikátor anebo samotný objekt scény v případě, že ji voláme na prvním modifikátoru tohoto objektu. V metodách **GetMeshPartCountCore**, **GetVerticesCountCore**, **GetIndicesCountCore**, **GetVerticesCore** a **GetIndicesCore** modifikátoru se tedy vždy zavolá tatáž metoda na vlastnosti **PreviousVertexIndexGenerator**, její výsledek pak modifikátor upraví (například modifikátor posunutí změní pozice vrcholů) a tyto upravené hodnoty vrátí.

Editor 3D scén obsahuje v základu 5 modifikátorů – posunutí, otočení kolem osy, změnu měřítka, aplikaci textury pomocí kolmého promítání a aplikaci vlastního mapování textury a barvy vrcholů.

Mimo to každý modifikátor obsahuje kolekci **SelectedVertices** a **SelectedFaces**, protože může být aplikován jen na určité vrcholy či plochy vybraného objektu. Modifikátory se při přidávání do kolekce **Modifiers** pokusí pomocí metody **Accumulate** sloučit s modifikátorem předchozím anebo, pokud je to možné, se rovnou aplikují na daný objekt. Pokud je příkladně na celý objekt aplikován modifikátor posunutí a tento objekt ještě žádné modifikátory nemá, stačí vektor posunutí přičíst k vlastnosti **Position**, která je definovaná rovnou ve třídě **TreeObject3D**.

Je možné samozřejmě dodefinovat vlastní modifikátory, stačí vytvořit třídu odvozenou od třídy **Modifier** deklarované v assembly **Age.Core.Scene.Scene3D**. Aby se modifikátor zobrazil v aplikaci v okně vlastností objektů a bylo jej možné přidat, stačí třídu odekorovat atributem **AgeModifier3D**, který bere jeden parametr, a to textový název do uživatelského rozhraní.

I modifikátory mohou mít vlastnosti typu **UserTreeExpression**, při jejich inicializaci v konstruktoru je třeba jako první parametr (vlastník výrazu) uvést objekt, na nějž je modifikátor aplikován. Vzhledem k tomu, že třída **Modifier** nemá bezparametrický konstruktor, je třeba deklarovat konstruktor se dvěma parametry, z nichž jeden parametr je právě rodičovský objekt scény. Inicializace vlastnosti modifikátoru je znázorněna na následující ukázce kódu.

```
public RotateModifier(Scene3D scene, TreeObject parent) : base(scene, parent)
{
    Origin = new UserTreeExpression<Vector3D>(parent, "vector3(0,0,0)",
        scene.EvaluationContext);
}
```

Uvnitř modifikátoru stačí přepsat metody deklarované rozhraním **IVertexIndexGenerator**, anebo v případě, že modifikátor nemění počet vertexů ani indexů, ale jen drobně vybrané vertexy upravuje, stačí přepsat metodu **TransformVertex**, která se zavolá pouze pro vertexy, na něž je modifikátor aplikován. Je třeba dát pozor na fakt, že metoda **TransformVertex** je volána z metody **GetVerticesCore**, v případě přepsání této metody a nezavolání **base.GetVerticesCore** se metoda **TransformVertex** nevyvolá.

4.8. Používání funkcí a datových typů ve výrazech

Aby byly možnosti enginu pro výrazy širší, ukázalo se jako vhodné mít možnost používat v zápisu výrazů funkce. Každý výraz typu **UserTreeExpression** dostává v konstruktoru instanci třídy **EvaluationContext**, o níž již padla zmínka v předchozím textu. Vyhodnocovací kontext obsahuje kromě jiných členů kolekci **Functions** typu **AgeFunctionCollection**, která obsahuje seznam funkcí, jež se dají z výrazů volat. Funkce je identifikována názvem, počtem a datovými typy argumentů. Podporováno je přetěžování, v aktuální verzi nejsou implementovány automatické konverze typů, takže datové typy předaných argumentů musí přesně vyhovovat signatuře této funkce.

Kolekce **Functions** obsahuje objekty implementující rozhraní **IAgeFunction**, jež definuje vlastnosti pro název funkce, pole datových typů argumentů a datový typ návratové hodnoty. Kolekce **AgeFunctionCollection** se plní metodou **LoadFunctionsFromAssembly**, které je předána assembly

k prohledání. Tato funkce předanou assembly projde a najde v ní všechny statické metody, jež mají přiřazen atribut **AgeFunction**. Taková funkce může vypadat například takto.

```
[AgeFunction()]
public static Vector3D Vector3(double x, double y, double z)
{
    return new Vector3D(x, y, z);
}
```

Pochopitelně je povoleno používat pouze datové typy podporované enginem výrazů (viz dále), a to jak pro návratové hodnoty, tak i pro typy vstupních argumentů.

Ve výrazech jsou podporovány datové typy popsané v následující tabulce.

Datový typ	Příklad	Význam
double	15.67	desetinné číslo
Vector3D	vector3(15.4, 17.8, 2)	souřadnice ve 3D
Vector2D	vector2(1, 5)	souřadnice ve 2D
string	“hodnota“	textová hodnota
Color	color(255, 164, 0, 78)	barva (A, R, G, B)

Tabulka 5: Vestavěné datové typy a způsoby jejich vytvoření pomocí vestavěných funkcí

V assembly **Age.Core.Scene.Scene3D** je dále definována řada funkcí označených atributem **AgeFunction**, tyto funkce lze používat z výrazů. Kompletní seznam těchto funkcí je uveden v kapitole Provázání objektů pomocí výrazů v uživatelské dokumentaci.

Engine pro vyhodnocování výrazů je rozšiřitelný o vlastní složené datové typy. Pro implementaci takového datového typu stačí zajistit, aby tento datový typ implementoval rozhraní **ITreeObject**. Toto rozhraní umožňuje zjistit datový typ a hodnoty vlastností složeného typu, například u hodnoty typu **Vector3D** se lze dotazovat na vlastnosti **X**, **Y** a **Z**.

Následující kód ukazuje implementaci vlastního datového typu.

```
public struct Vector3D : ITreeObject
{
    public double X, Y, Z;
    ...

    public ITreeExpression GetPropertyByName(string targetName)
    {
        switch (targetName.ToLower())
        {
            case "x":
                return new ConstantTreeExpression<double>() { Value = X };
            case "y":
                return new ConstantTreeExpression<double>() { Value = Y };
            case "z":
                return new ConstantTreeExpression<double>() { Value = Z };
            default: return null;
        }
    }
```

```
    public Type GetPropertyType(string targetName)
    {
        switch (targetName.ToLower())
        {
            case "x":
            case "y":
            case "z":
                return typeof(double);
            default: return null;
        }
    }

    public override string ToString()
    {
        return string.Format(EvaluationContext.FormatProvider,
            "vector3({0},{1},{2})",
            ConstantTreeExpression<double>.SerializeExpression(X),
            ConstantTreeExpression<double>.SerializeExpression(Y),
            ConstantTreeExpression<double>.SerializeExpression(Z)
        );
    }
}
```

Jak je vidět na této ukázce, je třeba implementovat metody **GetPropertyByName** a **GetPropertyType**. Dále je přepsána metoda **ToString** tak, aby vracela co nejjednodušší výraz, který danou hodnotu reprezentuje. Pro převedení hodnoty typu **double** na řetězec se používá metoda **SerializeExpression**, jelikož typ **double** se na řetězec převádí různě v závislosti na použitém národním prostředí v systému. Výše uvedená metoda zaručuje, že se hodnota převede do správného formátu, totiž na číslice bez oddělovače řádů a s desetinnou tečkou jako oddělovačem desetinných míst.

Je třeba dodat, že třída **TreeObject** implementuje rozhraní **ITreeObject** a uvnitř těchto metod pomocí Reflection najde příslušné vlastnosti a jejich datové typy. U rozhraní **ITreeObject** by bylo možné tento mechanismus použít také, ale metoda **GetPropertyByName** musí vracet výraz (**ITreeExpression**) a nepředpokládá se, že by jednoduché datové typy měly deklarovány skutečné vlastnosti obsahující výrazy. Proto jsou tyto metody ve strukturách **Vector3D**, **Vector2D** a **Color** implementovány bez použití Reflection.

4.9. Komponenta TreeObjectPropertyGrid

V uživatelském rozhraní editoru je třeba mít možnost uživatelsky zadávat a upravovat výrazy jednotlivých vlastností objektů. Proto byla implementována WPF komponenta **TreeObjectPropertyGrid**, která tuto funkcionalitu implementuje. Jediné, co je třeba pro její použití provést, je nastavit do její vlastnosti **Object** objekt, který chceme upravovat.

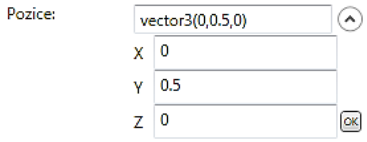
Komponenta projde pomocí Reflection všechny vlastnosti typu **UserTreeExpression<T>** označené atributem **AgeProperty** a pro každou tuto vlastnost vytvoří tzv. *editor*, který je specifikovaný pomocí tohoto atributu. Kromě toho se vytvoří i editor pro vlastnost **Id**, která se atributem **AgeProperty** neoznačuje a navíc je typu **string**.

Každý editor vlastnosti je opět WPF komponenta, která dědí ze třídy **PropertyGridEditor**. V následující tabulce jsou rozebrány implementované editory a jejich parametry.

Editor	Parametry	Význam
PropertyGridEditor		Základní třída, z níž všechny editory dědí.
PropertyGridIdEditor		Editor pro nastavení Id objektu scény.
PropertyGridExpressionEditor		Základní třída, z níž dědí všechny editory, které spravují vlastnost obsahující výraz.
PropertyGridColorEditor		Editor pro výrazy typu Color s možností výběru barvy z paletky.
PropertyGridEnumEditor	Type ... typ enumu	Editor pro výrazy, jejichž hodnotu lze vybrat z výčtového typu.
PropertyGridSimpleRangeEditor	double ... minimum double ... maximum	Editor pro výrazy typu double, jejichž hodnota musí být v daném intervalu.
PropertyGridRangeEditor	double ... minimum double ... maximum	Editor pro výrazy typu double, jejichž hodnota musí být v daném intervalu. Tento editor navíc zobrazí posuvník, na němž je možné hodnotu nastavit pomocí myši.
PropertyGridVector3DEditor		Editor pro výrazy typu Vector3D.
PropertyGridTextureIdEditor		Editor, jež umožňuje vybrat jednu z textur použitých ve scéně.

Tabulka 6: Typy editorů komponenty TreeObjectPropertyGrid a jejich popis

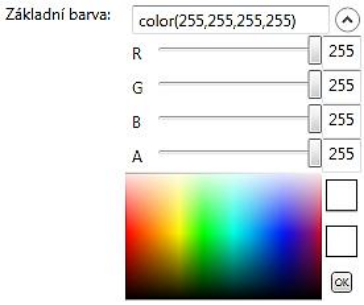
Každý editor zobrazí textové pole, do nějž je možné zadat výraz. Některé editory, například **PropertyGridEnumEditor** či **PropertyGridTextureIdEditor** zobrazí vedle tohoto textového pole ještě tlačítko, které zobrazí okno či kontextové menu pro výběr jedné z hodnot. Jiné editory, například **PropertyGridRangeEditor**, **PropertyGridVector3DEditor** či **PropertyGridColorEditor** zobrazí vedle textového pole tlačítko, které rozbalí pokročilé rozhraní pro výběr hodnoty výrazu, například posuvník, paletku s barvami, či tři textová pole pro zadání jednotlivých složek vektoru. Ostatní editory, například **PropertyGridSimpleRangeEditor** či **PropertyGridIdEditor** jsou obyčejná textová pole, která jen kontrolují, zda-li zadaná hodnota splňuje dané podmínky.



Obrázek 3: PropertyGridVector3DEditor

Obrázek 4: PropertyGridRangeEditor

Obrázek 5: PropertyGridTextureIdEditor



Obrázek 6: PropertyGridColorEditor

Na následující ukázce kódu jsou postupně uvedeny zápisy atributů pro výše uvedené obrázky.

```
[AgeProperty("Pozice", "PropertyGridVector3DEditor")]

[AgeProperty("Alpha textury", "PropertyGridRangeEditor", 0.0, 1.0)]

[AgeProperty("Textura", "PropertyGridTextureIdEditor")]

[AgeProperty("Základní barva", "PropertyGridColorEditor")]
```

4.10. Nástroje editoru

V kapitole 3.10 bylo popsáno a zdůvodněno, proč byl vybrán koncept nástrojů editoru, tato kapitola se věnuje jeho implementaci. Vzhledem k tomu, že koncept nástrojů editoru by měl být obecný, je základní třída nástroje **EditorTool** deklarována v assembly aplikace **Age.Application.Main**. Dále je zde též třída **EditorToolCollection**, což je třída reprezentující kolekci výrazů, která zajišťuje, že v každém okamžiku může být aktivní pouze jeden nástroj.

Třída **MdiChild** reprezentující dceřinné MDI okno editoru, o níž již byla řeč dříve, má mimo jiné vlastnost **EditorTools**, kterou lze naplnit libovolným množstvím nástrojů. Nástroje se pak zpřístupní v daném editoru.

Na následující ukázce je vzorová implementace nástroje, je z ní patrný způsob, kterými nástroje fungují. Tento nástroj zaregistruje odběr událostí **MouseDown**, **MouseMove** a **MouseUp** a zajišťuje například posun kamery ve scéně pomocí tažení myši se stisknutým pravým tlačítkem.

```
public class SceneMoveTool : Editor3DTool
{

    protected override void AddHandlers(Age.Application.Main.UI.MdiChild editor)
    {
        base.AddHandlers(editor);

        Editor.ViewPortGrid.MouseDown += new MouseButtonEventHandler(OnMouseDown);
        Editor.ViewPortGrid.MouseMove += new MouseEventHandler(OnMouseMove);
        Editor.ViewPortGrid.MouseUp += new MouseButtonEventHandler(OnMouseUp);
    }

    protected override void RemoveHandlers()
    {
        base.RemoveHandlers();

        Editor.ViewPortGrid.MouseDown -= new MouseButtonEventHandler(OnMouseDown);
        Editor.ViewPortGrid.MouseMove -= new MouseEventHandler(OnMouseMove);
        Editor.ViewPortGrid.MouseUp -= new MouseButtonEventHandler(OnMouseUp);
    }

    private void OnMouseDown(object sender, MouseButtonEventArgs e)
    {
        if (!CanStartAction) return;

        if (e.RightButton == MouseButtonState.Pressed)
        {
            NotifyBeginAction();

            ...
        }
    }

    private void OnMouseMove(object sender, MouseEventArgs e)
    {
        if (!IsActiveAction) return;

        ...
    }
```

```
    private void OnMouseUp(object sender, MouseEventArgs e)
    {
        if (IsActiveAction)
            NotifyEndAction();
    }
}
```

Metody **AddHandlers** a **RemoveHandlers** zajišťují registraci obsluhy událostí dceřinného MDI okna. Jsou automaticky zavolány při přidání resp. odebrání z kolekce nástrojů **EditorToolCollection**.

Nastane-li událost **MouseDown**, nástroj nejprve zkontroluje, zda-li může zahájit akci. Vlastnost **CanStartAction** vrací **false** v případě, že je aktivní jiný nástroj z kolekce, pokud se tak stane, nástroj nebude provádět žádnou akci.

Pokud žádný nástroj aktivní není, přijde na řadu podmínka, která testuje, zda-li je stlačeno pravé tlačítko myši. Pokud ano, zavolá se metoda **NotifyBeginAction**, která upozorní kolekci nástrojů, že aktuální nástroj zahájil nějakou akci a od této chvíle je aktivní. V tomto okamžiku již nástroj smí začít provádět vlastní akce, například může zaznamenat původní polohu kamery apod.

V události **MouseMove** se hned na začátku kontroluje, zda-li vlastnost **IsActiveAction** vrátí **true**. Pokud ne, žádná akce se nebude provádět, jelikož uživatel sice pohybuje s myší, ale aktuální nástroj není aktivní, neměl by tedy nic dělat. Pokud nástroj aktivní je, přijde na řadu opět logika specifická pro daný nástroj, například zjištění, o kolik se pozice myši změnila, a dále úprava kamery.

V okamžiku, kdy uživatel tlačítko myši uvolní, je nutné akci ukončit. V události **MouseUp** je opět kontrola, zda-li je nástroj aktivní. Pokud ano, zavolá se metoda **NotifyEndAction**, která kolekci nástrojů oznámí, že nástroj dokončil svou činnost a přestal být aktivní.

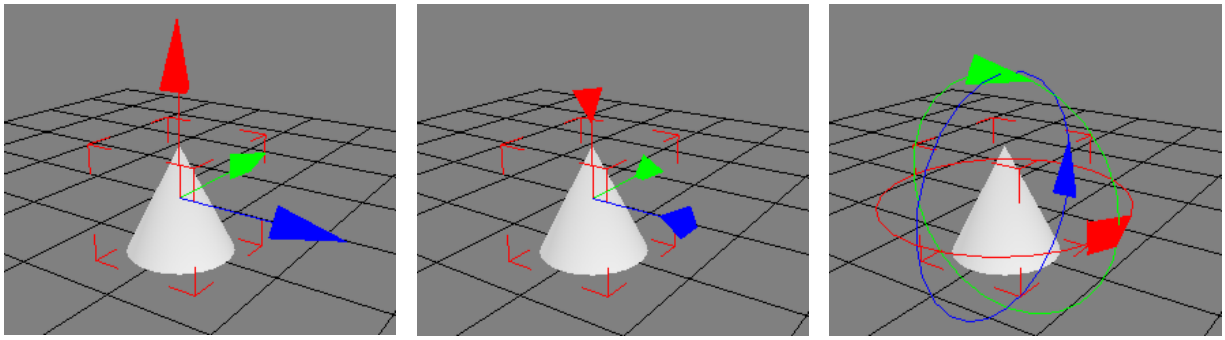
Tento mechanismus umožňuje oddělit logiku a implementaci jednotlivých akcí editoru do separátních tříd a zajišťuje, že se dvě akce nebudou provádět zároveň. Následující tabulka obsahuje seznam nástrojů a popis akcí, které obstarávají.

Nástroj	Popis
SceneMoveTool	Posun kamery ve scéně pomocí stisknutého pravého tlačítka myši.
SceneRotateTool	Rotace kamery ve scéně pomocí stisknutého pravého tlačítka myši v kombinaci s klávesou Ctrl nebo Alt.
SceneZoomTool	Přiblížení či oddálení kamery pomocí kolečka myši.
SceneSelectObjectTool	Výběr objektu ve scéně kliknutím, přidávání či odebírání objektů do výběru pomocí přidržení klávesy Shift.
SceneSelectVertexTool	Výběr vrcholů vybraného objektu kliknutím, přidávání či odebírání vrcholů do výběru pomocí přidržení klávesy Shift.
SceneSelectFaceTool	Výběr stěny vybraného objektu kliknutím, přidávání či odebírání stěn do výběru pomocí přidržení klávesy Shift.
SceneSelectionMoveTool	Posun vybraných objektů ve scéně.
SceneSelectionRotateTool	Rotace vybraných objektů ve scéně.
SceneSelectionScaleTool	Změna měřítka vybraných objektů ve scéně.

Tabulka 7: Implementované nástroje editoru a jejich funkce

4.11. Dekorátory

Některé nástroje, jako třeba výběr objektů, nebo například nástroj pro posunutí objektů, mohou potřebovat do scény dokreslit různé dekorativní a ovládací prvky, například vyznačit vybrané objekty, nebo zobrazit táhla, pomocí nichž je možné objekty posouvat. Každý nástroj editoru má proto metodu **GetDecorators**, jež se zavolá při každém renderováním scény a na základě stavu scény a nástroje by měla vrátit kolekci objektů dědících ze třídy **EditorToolDecorator**. Každý takový dekorátor má metodu **DrawDecoration**, jež má za úkol vykreslit potřebné ovládací prvky do scény.



Obrázek 7: Dekorátory pro posun, změnu měřítka a rotaci vybraných objektů ve scéně

4.12. Výběr objektů

Pro manipulaci se scénou je třeba též mít možnost vybírat objekty a jejich skupiny. Výběr je reprezentován abstraktní třídou **Age.Core.Scene.Scene3D.Selection**, z níž dědí tři různé třídy – **TreeObjectSelection**, **VertexSelection** a **EdgeSelection**.

První třída reprezentuje kolekci objektů ve scéně. Objekty, které jsou vybrané, jsou uchovávány v kolekci **SelectedItems**. Druhá a třetí třída fugurují pouze v rámci jednoho objektu scény, který je specifikován v jejich vlastnosti **Owner**. Kromě toho definují kolekci **SelectedItems** obsahující struktury typu **VertexPointer** resp. **FacePointer**. Třída **VertexSelection** je výběr množiny vrcholů v rámci jednoho objektu scény, třída **FaceSelection** je výběr množiny trojúhelníků v rámci objektu ve scéně.

Struktura **VertexPointer** definuje index části objektu (viz kapitola 4.7) a index vrcholu v dané části. Struktura **FacePointer** definuje index části objektu a index trojúhelníku v rámci tohoto objektu.

Zda-li se ve scéně aktuálně vybírají objekty, vrcholy nebo trojúhelníky, je dáno vlastností **SelectionMode** komponenty **Scene3DView**, hodnota této vlastnosti se přepíná tlačítky v panelu nástrojů či klávesovými zkratkami.

4.13. Formát uložení 3D scény (A3D)

Aplikace AGE umí 3D scénu uložit jako samostatný soubor ve formátu A3D, který je založen na formátu XML. Vzhledem k tomu, že aplikace AGE potřebuje ze souborů libovolného modulu získávat metadata, např. název či popis, je v assembly **Age.Application.Main** deklarována abstraktní třída **AgeSerializer**. Každý modul pak definuje kolekci svých serializérů pro všechny podporované formáty. Každý serializér musí implementovat minimálně metodu **LoadMetadata**, která vrací kolekci dvojic názvu a hodnoty.

Povinné položky metadat jsou **Title** a **Description**, každý formát si může samozřejmě definovat libovolná další metadata.

Modul 3D editoru implementuje třídu **A3dSerializer**, jež se stará o načítání a ukládání scén do souboru. Prakticky jde o velmi jednoduchou XML serializaci objektového modelu scény.

Specifikace formátu A3D

Soubor A3D je XML dokument s kořenovým elementem **Age3d**. Tento element musí obsahovat elementy **Metadata** a **Content** (každý právě jednou) a může obsahovat jeden nepovinný element **Resources**.

Uvnitř sekce **Metadata** je povolen libovolný počet elementů **Item** s povinným atributem **Key** definujícím název položky metadat. Hodnota příslušné položky metadat je textový obsah každého elementu **Item**.

Uvnitř elementu **Content** je serializovaný strom scény. Každou položku kolekce **Children** scény reprezentuje jeden element s názvem stejným jako je název třídy daného objektu (viz Tabulka 8: Povolené vlastnosti elementů reprezentující objekty ve scéněTabulka 8). Každý element reprezentující objekt ve scéně má nepovinné atributy stejných názvů jako jsou vlastnosti uvnitř třídy typu **UserTreeExpression<T>** označené atributem **AgeProperty**. Hodnoty těchto atributů jsou textové reprezentace výrazů v příslušných vlastnostech.

Každý element reprezentující objekt ve scéně může obsahovat právě jeden element **xxx.Modifiers**, kde **xxx** je název elementu. Uvnitř je pro každý prvek kolekce **Modifiers** objektu scény jeden element, pro nějž platí stejné pravidlo – název elementu je stejný jako název třídy modifikátoru a povolené atributy jsou vlastnosti typu **UserTreeExpression<T>** označené atributem **AgeProperty**.

Pokud element objektu scény obsahuje jiné vnitřní elementy, pak tyto elementy reprezentují obsah položky kolekce **Children** a platí pro ně stejná pravidla jako pro jejich rodičovské elementy.

Každý element reprezentující objekt ve scéně nebo modifikátor může navíc obsahovat atributy tvaru **ISerializationHandler.xxx**, kde xxx je název dodatečného atributu. Tyto atributy lze použít u elementů reprezentujících objekty, které implementují rozhraní **ISerializationHandler**. Toto rozhraní má dvě metody – **GetSerializationProperties** vrací kolekci název atributu a hodnota, které se mají danému objektu při serializaci do výstupu přidat, a metoda **SetSerializationProperties** dostává při deserializaci kolekci načtených atributů začínajících **ISerializationHandler**, aby je nastavila danému objekt.

Toto rozhraní implementují například modifikátory pro ukládání a načítání vlastností **SelectedVertices** a **SelectedFaces**, jelikož tyto vlastnosti nejsou reprezentovány pomocí výrazů. Engine pro výrazy totiž zatím nepodporuje práci s poli.

Poslední element **Resources** kořenového elementu **Age3d** je nepovinný a pokud je uveden, může obsahovat libovolný počet elementů **ResourceItem**. Každý element reprezentuje jednu položku kolekce **Scene.ResourceManager.Items** obsahující například textury atd. Povinné atributy jsou **Id** a **Extension**, které určují unikátní ID v rámci elementu **Resources** a příponu souboru. Obsah souboru je jako textový obsah uvnitř elementu **ResourceItem** zapsaný kódováním Base64.

Na následující ukázce je vidět vzorový soubor A3D.

```
<?xml version="1.0" encoding="utf-8"?>
<Age3d>
  <Metadata>
    <Item Key="Version">1</Item>
    <Item Key="Title">Vzorová scéna</Item>
    <Item Key="Category">Test</Item>
    <Item Key="Description">Popis vzorové scény</Item>
  </Metadata>
  <Content>
    <Cube id="Cube1" Width="1" Height="1" Depth="1" Position="vector3(0,0.5,0)"
      MaterialId="&amp;quot;&amp;quot;">
      <Cube.Modifiers>
        <VertexColorTextureModifier
          ColorFunction="color(255,($$.X+0.5)*255,($$.Y+0.5)*255,($$.Z+0.5)*255)"
          TextureFunction="Vector2(0,0)" />
      </Cube.Modifiers>
    </Cube>
  </Content>
  <Resources>
    <ResourceItem Id="Resource1" Extension="txt">aGVsbG8gd29ybGQ=</ResourceItem>
  </Resources>
</Age3d>
```

Objekty a vlastnosti

V následující tabulce je seznam povolených elementů reprezentujících objekt a jejich vlastností. Tyto objekty jsou implementovány ve 3D editoru.

V případě rozšíření aplikace o vlastní typy 3D objektů je třeba třídy těchto objektů označit atributem **AgeObject3D**. Serializér při hledání mapování mezi názvem elementu a datovým typem prohledá všechny načtené assemblies a umožní použít všechny třídy označené tímto atributem. Jediný požadavek je, aby názvy tříd byly unikátní.

Element	Vlastnost	Typ	Význam
Cube	Width	double	Šířka krychle
	Height	double	Výška krychle
	Depth	double	Hloubka krychle
	Position	Vector3D	Pozice středu tělesa
	MaterialId	string	ID materiálu
Sphere	Radius	double	Poloměr koule
	Position	Vector3D	Pozice středu tělesa
	MaterialId	string	ID materiálu
Prism	Sides	double	Počet stran hranolu
	Radius	double	Poloměr opsané kružnice podstav
	Height	double	Výška hranolu
	Position	Vector3D	Pozice středu tělesa
	MaterialId	string	ID materiálu
Prism	Sides	double	Počet stran jehlanu
	Radius	double	Poloměr opsané kružnice podstavy
	Height	double	Výška jehlanu
	Position	Vector3D	Pozice středu tělesa
	MaterialId	string	ID materiálu

Polygon	Sides	double	Počet stran polygonu
	Radius	double	Poloměr opsané kružnice
	Position	Vector3D	Pozice středu tělesa
	MaterialId	string	ID materiálu
Plot2D	Width	double	Šířka grafu 2D funkce
	Height	double	Výška grafu 2D funkce
	StepsX	double	Počet vzorků na ose X
	StepsY	double	Počet vzorků na ose Y
	ValueFunction	function<double>	Funkce udávající hodnotu v daném bodě
	ColorFunction	function<Color>	Funkce udávající barvu v daném bodě
	Position	Vector3D	Pozice středu tělesa
	MaterialId	string	ID materiálu
Material	DiffuseColor	Color	Barva povrchu tělesa
	TextureOpacity	double	Průhlednost textury
	TextureId	string	ID textury

Tabulka 8: Povolené vlastnosti elementů reprezentující objekty ve scéně

Každý objekt má kromě vlastností z tabulky též vlastnost **Id**, která udává jeho jednoznačné Id v rámci scény.

V následující tabulce je seznam povolených elementů reprezentujících modifikátor a jejich vlastností. Tyto modifikátory jsou ve 3D editoru aplikace AGE implementovány. I tento seznam lze rozšířit o vlastní modifikátory, je třeba třídu modifikátoru označit atributem **AgeModifier3D**. V takovém případě jej serializér opět rozpozná a zpřístupní.

Element	Vlastnost	Typ	Význam
TranslateModifier	PositionDelta	Vector3D	Relativní posunutí
RotateModifier	Angle	double	Úhel otočení
	Axis	double	Osa (1 = X, 2 = Y, 4 = Z)
	Origin	Vector3D	Počátek transformace otočení
ScaleModifier	AxisRatio	Vector3D	Vektor udávající změnu měřítka na jednotlivých osách
	Origin	Vector3D	Počátek transformace změny měřítka
PlanarTexture-MapModifier	TopLeft	Vector3D	Bod průmětny, kde je bod (0,0) textury
	BottomRight	Vector3D	Bod průmětny, kde je bod (1,1) textury
	Normal	Vector3D	Směr promítacích paprsků
VertexColor-TextureModifier	TextureFunction	function<Vector2D>	Funkce udávající texturovací souřadnice v daném bodě
	ColorFunction	function<Color>	Funkce udávající barvu v daném bodě

Tabulka 9: Povolené vlastnosti elementů reprezentující modifikátory objektů scény

Každý modifikátor navíc má rozšířené vlastnosti **ISerializationHandler.SelectedVertices** a **ISerializationHandler.SelectedFaces**, které obsahují prázdný řetězec, pokud se modifikátor aplikuje na objekt, anebo řetězec dvojic formátu **(x;y)** oddělených čárkou, kde prvním členem dvojice je vždy index části modelu a druhým členem je index vrcholu resp. trojúhelníka, na nějž se modifikátor aplikuje.

4.14. Rozšiřitelnost

V této kapitole jsou stručně shrnuty možnosti rozšiřitelnosti 3D editoru a aplikace AGE.

- Implementace vlastních rozšiřujících modulů, viz kapitola 4.1.
- Implementace vlastních typů objektů ve scéně, viz kapitola 4.6.
- Implementace vlastních modifikátorů objektů ve scéně, viz kapitola 4.7.
- Implementace vlastních datových typů, které lze použít ve výrazech, viz kapitola 4.5.
- Implementace vlastních funkcí, které lze použít ve výrazech, viz kapitola 4.5.

5. Srovnání s existujícími řešeními na trhu

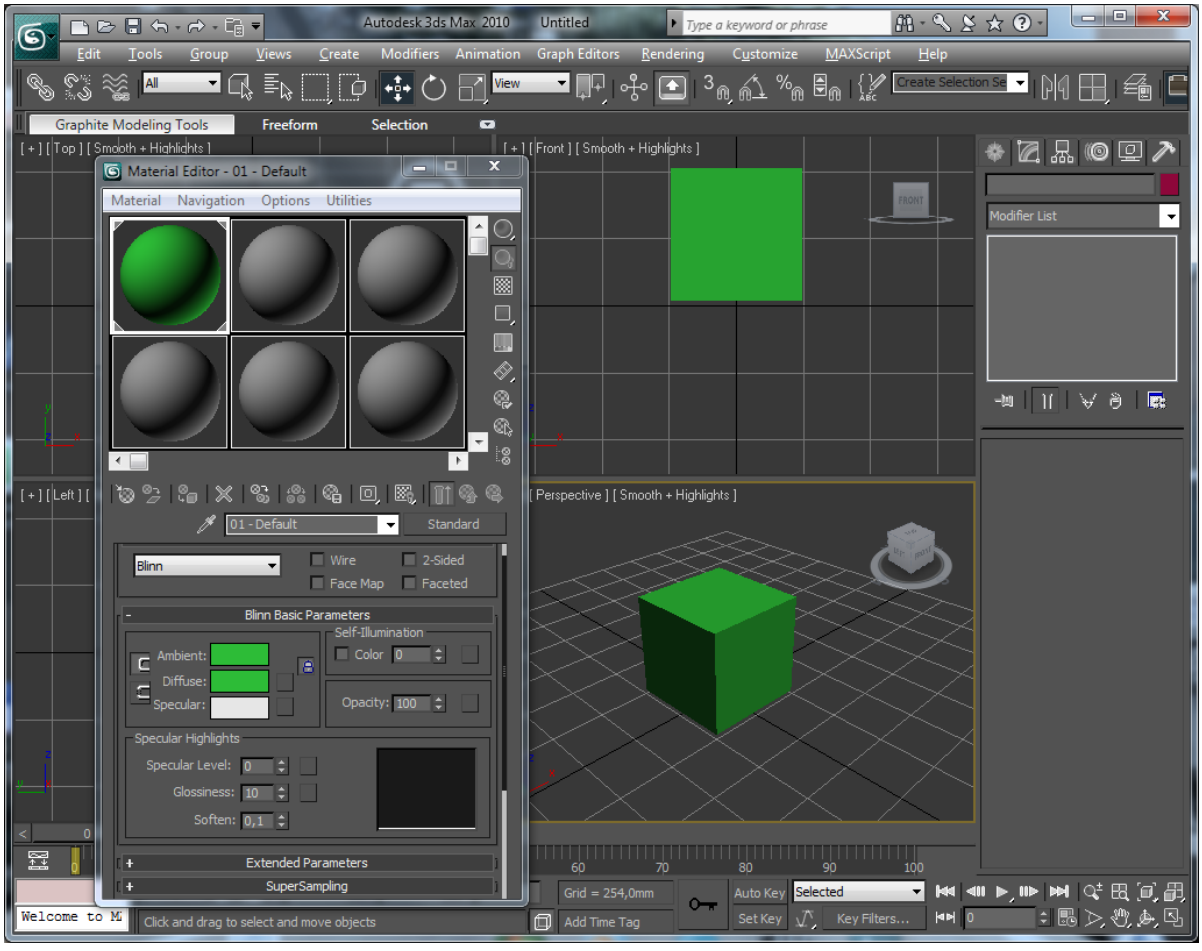
Vzhledem k tomu, že aplikací pro modelování 3D scén je na trhu celá řada a liší se funkcemi i cenou, vybral jsem pro srovnání jednu aplikaci, která je placená, a dvě, které jsou k dispozici zdarma. Jako příklad komerčního 3D editoru jsem vybral aplikaci **Autodesk 3ds Max**, jako příklad volně dostupné aplikace jsem vybral **Google SketchUp** a open source editor **Blender**.

5.1. Autodesk 3ds Max

Aplikace **3ds Max** vyvíjená společností **Autodesk** je kompletní řešení pro tvorbu a úpravy 3D scén a modelů, na trhu má již velmi dlouhou historii a těší se značné oblibě. Používá se například při tvorbě reklam a ve filmovém průmyslu, dále například pro modelování architektonických konstrukcí a v neposlední řadě nachází často uplatnění i při tvorbě grafiky do počítačových her.

Tato aplikace implementuje nepřebernou škálu funkcí pro modelování pokročilých tvarů a těles, podporuje tvorbu animací, obsahuje speciální nástroje pro modelování postav apod. Má též velmi pokročilé a detailní možnosti pro aplikování materiálů na tělesa, od jednoduchého texturování až po rozsáhlé možnosti nastavení odrazivosti, lomu světla a zvrásnění povrchu.

Podporováno je poměrně velké množství formátů pro import a export scén, díky čemuž je usnadněno předávání modelů z a do jiných 3D aplikací. Dnes je na trhu celá řada rozšíření této aplikace, 3ds Max obsahuje skriptovací jazyk MAXScript, který umožňuje skriptovat často opakované úkoly i vyvíjet nové nástroje a rozšíření celé aplikace.



Obrázek 8: Uživatelské rozhraní Autodesk 3ds Max 2010

Ve srovnání s 3D editorem aplikace AGE nabízí aplikace 3ds Max mnohem širší možnosti a větší počet použitelných nástrojů, což je samozřejmě dáno mnohem delší dobou vývoje 3ds Max a také faktem, že na jeho vývoji se podílejí stovky lidí.

AGE oproti 3ds Max nabízí podporu výrazů pro definování vztahů mezi tělesy ve scéně. Použití výrazů v 3D editoru AGE je jednoduché a velmi snadné na naučení. MAXScript má širší možnosti, protože jde o plnohodnotný skriptovací jazyk a ne jen o jednoduché výrazy, na druhou stranu naučit se a používat jej je znatelně složitější.

5.2. Google SketchUp

Google SketchUp je poměrně jednoduchý editor 3D scén a nákresů, který se vyznačuje zejména snadností ovládání. Původním záměrem jeho vzniku bylo vytvořit jednoduchý editor umožňující kterémukoliv uživateli vymodelovat budovu a umístit ji do online map Google Earth[11]. Kromě aplikace **Google SketchUp**, která je k dispozici zdarma, existuje i rozšířená placená verze **Google SketchUp Pro**, která obsahuje pokročilejší funkce a nástroje.

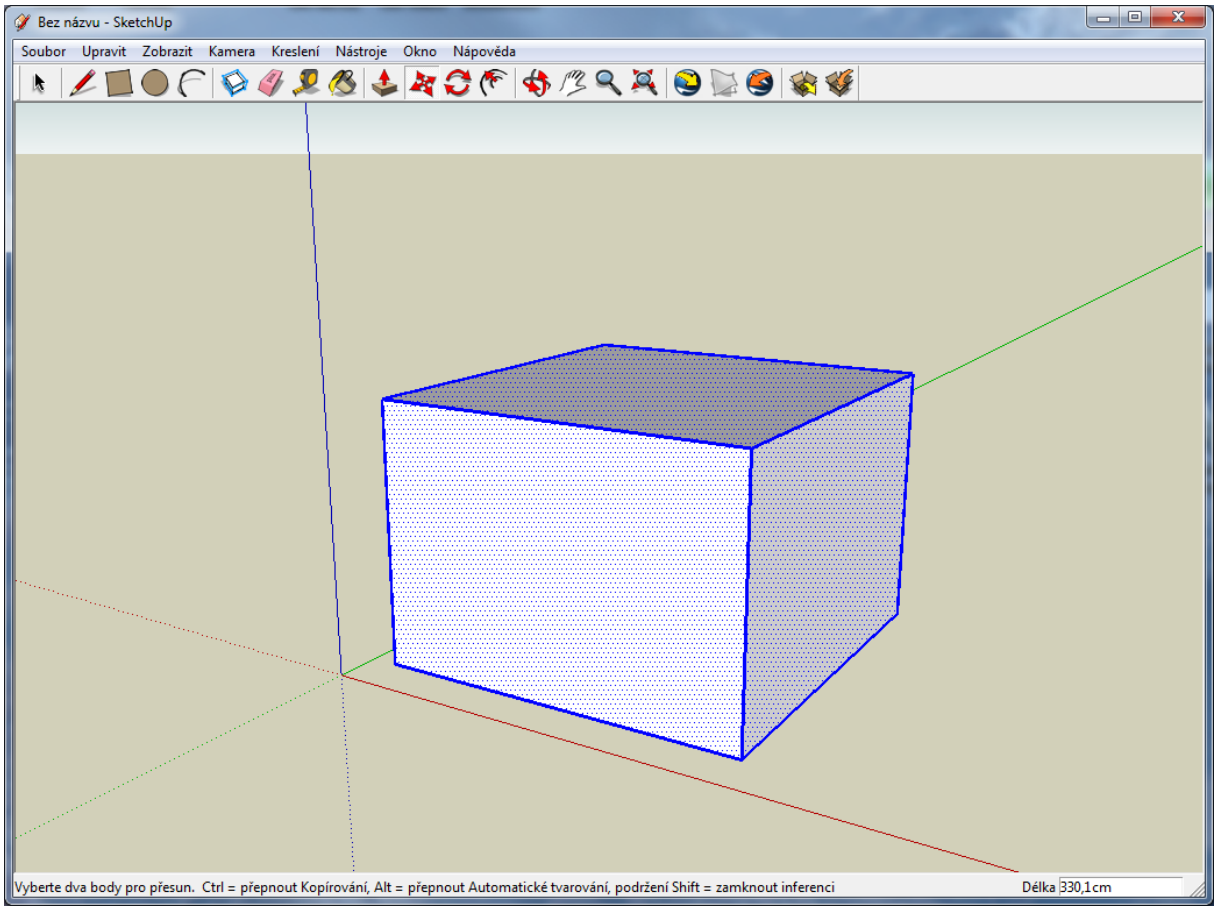
Uživatelské rozhraní této aplikace (viz Obrázek 9) je oproti předchozímu příkladu značně jednodušší. Tato aplikace je dostupná zdarma a nemá tak dlouhou historii. Rozhodně neobsahuje takové množství nástrojů a funkcí jako 3ds Max, na druhou stranu vzhledem k jejímu cílení na běžné uživatele a začátečníky je to pochopitelné.

Pokud bychom měli srovnat tuto aplikaci s 3D editorem AGE, je zde mnoho rozdílů. Jedním z rozdílů je například styl modelování. V AGE se typicky začíná tak, že do scény vložíme některý základní objekt, například krychli, a s ním pak pracujeme. Typicky jej upravujeme aplikací modifikátorů na jeho různé části. V aplikaci Google SketchUp se naproti tomu obvykle začne nakreslením 2D útvaru, například obdélníku či elipsy a následně se tento útvar rozšíří do 3D tělesa vytažením v určitém směru.

Celkem diskutovaným nedostatkem této aplikace je možnost specifikace přesných rozměrů u objektů. I v případě, že známe přesné cílové rozměry, nemáme možnost je zadat jako číselnou hodnotu, musíme se do příslušného rozměru „trefit“ tažením myši. Rozhodně příjemnější by bylo, kdyby aplikace umožňovala oba dva způsoby – určení velikosti tělesa myší či klávesnicí.

Aplikace Google SketchUp neumožňuje na rozdíl od 3D editoru AGE definovat vztahy mezi objekty ve scéně, například vzájemné pozicování či velikost relativní vůči určitému tělesu atd. Pro účely, na něž je aplikace Google SketchUp cílena, to nevadí, jelikož jednoduché budovy pro online mapy lze modelovat velmi pohodlně i bez těchto funkcí.

[11] Mapovou aplikaci Google Earth lze stáhnout a nainstalovat z webové adresy http://earth.google.com.



Obrázek 9: Uživatelské rozhraní Google SketchUp

5.3. Blender

Blender je multiplatformní open-source aplikace umožňující vytváření 3D modelů a animací. Obsahuje poměrně širokou škálu funkcí a nástrojů, obsahuje například engine pro tvorbu 3D her a aplikací, který je možno skriptovat v jazyce Python. Dále obsahuje například nástroje pro tvorbu částicových systémů. Podporuje též velké množství formátů pro import a export scén.

Opět vzhledem ke své dlouhé historii obsahuje Blender mnohem více nástrojů a funkcí než AGE 3D editor. Pro skriptovací účely se používá jazyk Python, možnosti skriptovacího prostředí jsou mnohem silnější než výrazy u 3D editor. Na druhou stranu psaní skriptů v Pythonu je opět znatelně složitější na naučení než použití jednoduchých výrazů v AGE 3D editoru, které na základní úkoly stačí.

5.4. AGE 3D Editor

Vzhledem ke své krátké historii editor AGE neobsahuje tak bohatou škálu funkcí jako předchozí dva produkty (a mnoho dalších, které jsou též na trhu), na druhou stranu přináší několik nových konceptů.

Již mnohokrát byl v předchozím textu zmíněn koncept výrazů, je tedy na čase prezentovat jeho výhody a přínosy při modelování 3D scén.

Nastavení velikosti objektu

Jednoduchým výrazem lze nastavit, aby šířka krychle **Cube2** byla stejná jako X souřadnice krychle **Cube1**. V okamžiku, kdy první krychli posuneme, druhá bude měnit svoji velikost. Krychli **Cube2** stačí nastavit jako šířku tento následující výraz.

```
$Cube1.Position.X
```

Vzájemné pozicování objektů pomocí výrazů

Mějme ve scéně dvě krychle s identifikátory **Cube1** a **Cube2**. Chceme, aby první krychle byla vždy o 3 jednotky vlevo od druhé krychle. V okamžiku, kdy posuneme krychli **Cube2**, automaticky se posune i krychle **Cube1**. Stačí pozici krychle **Cube1** nastavit na následující výraz.

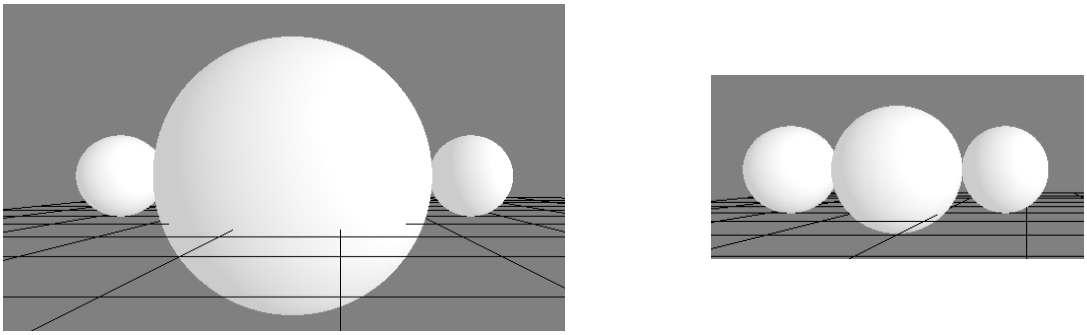
```
vector3($Cube2.Position.X - 3, $Cube2.Position.Y, $Cube2.Position.Z)
```

Vyplnění prostoru mezi objekty

Mějme ve scéně 3 koule **Sphere1**, **Sphere2** a **Sphere3**. Rádi bychom, aby koule **Sphere3** vyplnila prostor mezi koulemi **Sphere1** a **Sphere2**, ať už tyto koule přesuneme kamkoliv. Stačí pozici a poloměr koule **Sphere3** nastavit jako následující výrazy.

```
center($Sphere1.Position, $Sphere2.Position)

abs(distance($Sphere1.Position, $Sphere2.Position)
       - ($Sphere1.Radius + $Sphere2.Radius)) / 2
```



Obrázek 10: Koule vyplňující prostor mezi dvěma jinými koulemi

Další příklady použití výrazů jsou uvedeny v uživatelské dokumentaci. Zajímavé je například použití výrazů při obarvování objektů či použití v komponentě graf 2D funkce, kterou Age 3D editor implementuje.

Závěr

Hlavní cíl této práce byl splněn – vzniknul jednoduchý editor 3D grafiky, který obsahuje základní funkce pro manipulaci s objekty i jejich částmi a který obsahuje i některé mírně pokročilejší funkce, jako například funkce pro obarvení tělesa. Vznikla i první verze hostitelské aplikace, která v sobě umí spouštět moduly umožňující editovat různé typy souborů.

Integrace DirectX scény do aplikace vytvořené ve Windows Presentation Foundation je možná a funkční. Samotná třída *D3DImage*, která tuto interoperabilitu zajišťuje, je dosti nízkoúrovňová, takže je vhodné ji zapouzdřit do nějaké komponenty, aby se s ní snáze pracovalo. Není nutné psát vlastní wrapper nad DirectX rozhraním, lze bez větších obtíží použít nějaký existující, například poměrně bohatá knihovna SlimDX. Konkrétní způsob použití a implementace zapouzdřující komponenty byly podrobně diskutovány v kapitole 4.3.

Koncept výrazů pro specifikaci vlastnosti objektů ve scéně je ve 3D editorech poměrně neobvyklý. Existuje mnoho situací, kde se výrazy dají využít, zvláště pokud by se editor rozšířil tak, aby výrazy bylo možné místo psaní nastavit pomocí klikání v uživatelském rozhraní, mohlo by se jednat o poměrně použitelnou a oblíbenou funkci, která se dá použít například k zarovnávání objektů vůči sobě atd. Rozhodně výrazy umožnily velmi zajímavé funkce, jako například objekt grafu 2D funkce nebo parametrizovatelné obarvování objektů. Stávající aplikace na trhu místo výrazů používají většinou pokročilejší skriptovací jazyky, které mají sice větší vyjadřovací sílu, na druhou stranu nejsou tak jednoduché na naučení. Implementace enginu pro vyhodnocování výrazů byla rozebrána v kapitolách 4.5 a 4.8.

Možná budoucí rozšíření

Aplikaci AGE i modul 3D editoru lze samozřejmě dále rozšiřovat. 3D editoru chybí ještě mnoho funkcí, aby se vyrovnal existujícím komerčním produktům. Bylo by jistě vhodné přidat více podporovaných formátů pro ukládání a načítání scén, dále pak pokročilejší nástroje a modifikátory, širší možnosti materiálů a mapování textur, práci se světly, stíny, modelování křivek apod.

Velmi zajímavé možnosti by ve 3D editoru přineslo rozšíření enginu výrazů na plnohodnotný skriptovací jazyk a velmi zajímavé uplatnění by výrazy mohly najít při rozšíření možností 3D editoru o podporu animací. Tím by se značně zvýšil potenciál celého konceptu výrazů ve scéně.

Další obrovskou oblastí pro rozšiřování je vývoj dalších modulů pro aplikaci AGE, například editor 2D vektorové a rastrové grafiky. Aplikace AGE je sice původně cílená pro práci s grafikou, ale koncept modulů je natolik obecný, že by nebyl problém udělat například modul pro editaci textových dokumentů, biologické či fyzikální simulace a nejrůznější další oblasti.

Pro účely této bakalářské práce tedy byla implementována základní funkcionalita aplikace AGE a jednoduchý editor 3D scén, který poslouží jako ukázková implementace rozšiřujícího modulu, podle níž je možné implementovat moduly další.

Použitá literatura

[1] Sanchez J., Canton M. P. (2000): DirectX 3D Graphics Programming Bible. *Hungry Minds*, 36-40.

[2] Microsoft Developer Network (2010): Transforms in Direct3D 9.
URL: http://msdn.microsoft.com/en-us/library/bb206269(VS.85).aspx

[3] Pelikán J. (2009): Počítačová grafika I, materiály z přednášky (Vyplňování n-úhelníka).
URL: http://cgg.mff.cuni.cz/~pepca/lectures/pdf/fillpolygon.pdf

[4] Pelikán J. (2009): Počítačová grafika II, materiály z přednášky (Textury a šumové funkce).
URL: http://cgg.mff.cuni.cz/~pepca/lectures/pdf/textures.pdf

[5] Žára J., Beneš B., Sochor J., Felkel P. (2005): Moderní počítačová grafika, *Computer Press*, Praha.

[6] Microsoft Developer Network (2010): Rendering from Vertex and Index Buffers in Direct3D 9.
URL: http://msdn.microsoft.com/en-us/library/bb147325(VS.85).aspx

[7] Microsoft Developer Network (2010): D3DImage Class (System.Windows.Interop).
URL: http://msdn.microsoft.com/en-us/library/system.windows.interop.d3dimage.aspx

[8] Dr. WPF, článek na serveru CodeProject.com (2008): Introduction to D3DImage.
URL: http://www.codeproject.com/KB/WPF/D3DImage.aspx

[9] Microsoft Developer Network (2010): „Airspace“ and Windows Regions Overview.
URL: http://msdn.microsoft.com/en-us/library/aa970688.aspx

[10] Šturala A., článek na serveru Vývojář.cz (2006): Direct3D ve WPF?
URL: http://www.vyvojar.cz/Articles/438-direct3d-ve-wpf-jo.aspx

[11] Microsoft Developer Network (2010): Configuring Applications.
URL: http://msdn.microsoft.com/en-us/library/kza1yk3a.aspx

[12] Cox Ch., Klutcher M., článek v časopise MSDN Magazine (květen 2007):
Unleash Your Imagination With XNA Game Studio Express.
URL: http://msdn.microsoft.com/en-us/magazine/cc163420.aspx#S3

[13] Microsoft Developer Network (2010): Windows Vista Display Driver Model.
URL: http://msdn.microsoft.com/en-us/library/aa480220.aspx