Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

GLSL editor

Plzeň 2014 Bohumil Podlesák

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 15. května 2014

Bohumil Podlesák

Abstract

First part of this work is an overview of existing IDE tools that can be used to edit the code and visualize the scene of graphics applications that use GLSL shaders. The advantages and disadvantages of the different tools will be evaluated.

The second part of this work focuses on creating a GLSL editor. This new tool will allow the user to easily create and visualize a new effect. User will be able to create a scene using C# language and GLSL shaders in interactive editor.

Abstrakt

Úkolem první části této práce je seznámit se s existujícími vývojovými prostředími pro úpravu a vizualizaci GLSL shaderů. Budou popsány výhody a nevýhody jednotlivých nástrojů.

Úkolem druhé části této práce je navrhnout GLSL editor, který bude umožňovat uživateli jednoduše vytvořit a zobrazit nový efekt. Uživatel bude moci sestavit scénu v jazyce C# za použití GLSL shaderů v interaktivním editoru.

Obsah

1 Úvod

Cílem této diplomové práce je vyvinout software, který půjde jednoduše použít pro vývoj efektů pro grafické karty. Program musí být schopen jednoduše a pohodlně umožnit programátorovi inicializovat zdroje, sestavit scénu, specifikovat vykreslovací smyčku a napsat programový kód shaderů pro konečné vykreslení. Programátor bude mít k dispozici možnost upravovat program vizualizační smyčky v jazyce C#. Shadery bude programátor psát v jazyce GLSL.

Teoretická část bude popsána v kapitole vysvětlující základy grafického řetězce a používání OpenGL. Bude zde vysvětlena terminologie celé problematiky a získané znalosti o funkci OpenGL programů aplikujeme při návrhu editoru.

Součástí práce je dále analýza software, který se v současné době může použít pro účely editace, testování, ladění a vývoj shaderů. Bude zhodnocena kvalita jednotlivých programů z hlediska zaměření této práce. V závěru této kapitoly bude shrnutí vlastností daných programů a bude vysvětleno, co by bylo třeba udělat lépe nebo čím by se dalo inspirovat.

V části programové realizace budou zhodnoceny některé existující programové komponenty a technologie, které mohou být použity při implementaci. Zároveň budou popsány jednotlivé vlastnosti výsledné aplikace a jejich implementace.

V příloze bude vysvětlena práce s výsledným programem na připravených ukázkových úlohách.

2 Grafický řětězec

Při využívání fixní funkcionality grafického řetězce je těžké dosáhnout pokročilejších efektů. Buď je třeba často přepínat stavy, nebo je nutné vyvinout nový hardware, který dokáže daný efekt provést. S tím se však váží problémy s nekompatibilitou různých grafických karet. Navíc narůstá neúměrně komplexita celého řetězce. Velmi často není možné dosáhnout určité kvality požadovaných efektů nebo dokonce nemůžeme zobrazení daných efektů dosáhnout vůbec.

Trendem moderní doby při programování grafických aplikací je vyměnit fixní funkcionalitu grafického řetězce a nahradit ji za programovatelné shadery. Výhody vyplývají z toho, že u programovatelných grafických karet je možné popsat mnohem složitější a rozmanitější efekty, než v případě fixní funkcionality.

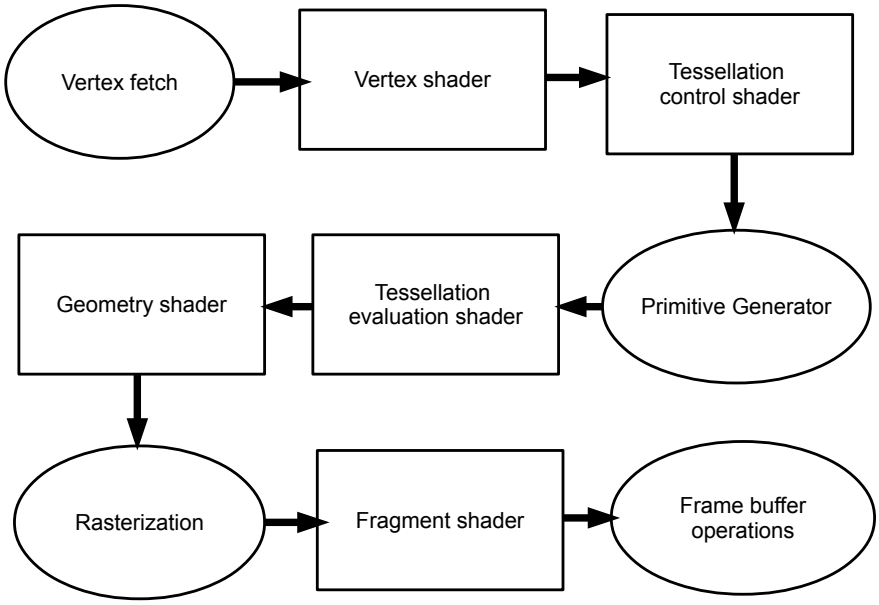
Nejčastější a nejjednodušší využití je při zpracování vrcholů přicházejících na vstupu grafického řetězce (transformace pozic jednotlivých bodů) a zpracování fragmentů, které jsou výstupem z rasterizéru. Aplikační programátoři mohou vyjádřit programovým kódem, jak se budou primitiva (vrcholy, fragmenty, nebo jiná) zpracovávat v programovatelných bodech grafického řetězce. Jednotlivé programy, které se píší pro programovatelné procesory zmíněných bodů, se nazývají shadery [1].

2.1 OpenGL

OpenGL je prostředí pro vytváření přenositelných a interaktivních 2D či 3D grafických aplikací. OpenGL 1.0 bylo poprvé vydáno v roce 1992. OpenGL se stalo jedním z nejrozšířenějších API, které podporuje 2D a 3D grafické aplikace. Ekvivalent k OpenGL, který je poměrně rozšířený, je DirectX, který se dá používat pouze na platformách se systémem od Microsoftu.

OpenGL API je standardem v průmyslu, neustále se vyvíjí a je stabilní. OpenGL aplikace mohou běžet na osobních PC, tak na přenosných zařízeních. Toto API je velmi dobře zdokumentováno: bylo publikováno mnoho knih o OpenGL a zároveň vydáno mnoho příkladů, kde se toto API využívá [3].

Na obrázku 2.1 je znázorněn grafický řetězec. Oválné rámečky ukazují



Obrázek 2.1: Grafický řetězec v OpenGL 4.3 [4]

části řetězce, které jsou fixní. Obdélníkové rámečky ukazují části řetězce, které jsou programovatelné. Tyto programovatelné části spouštějí shadery, které programátor poskytuje.

Shadery jsou nezávislé a dají se kompilovat zvlášť. Program je pak množina shaderů, které jsou kompilované a spojené dohromady (této fázi se říká anglicky linking). OpenGL používá entry pointy pro manipulaci a komunikaci s těmito programy.

2.2 GLSL jazyk

Jazyk GLSL byl navržen proto, aby dal vývojářům možnost psát přenositelné shadery, tedy bez specifických instrukcí hardwaru pro různé grafické karty. Existuje několik jazyků pro shadery OpenGL, ale GLSL je jediný, který je částí jádra OpenGL. Tento jazyk sdílí model zastarávání určitých funkcí (deprecation model).

GLSL je založen na ANSI C. Případy, kdy má tento jazyk jiné vlastnosti, nastanou pouze tehdy, když by stejná funkcionalita ve skutečnosti snižovala výkon takto napsané aplikace nebo jednoduchost implementace. Na rozdíl od implementace ANSI C byly přidány například vektorové či maticové typy (které jsou hardwarově podporovány). Byly použity i některé mechanismy z C++, jako například přetěžování funkcí na základě typu argumentu a schopnost deklarovat proměnné kdekoliv uvnitř funkčního bloku (ne pouze na začátku) [5].

2.2.1 GLSL kompilační model

Kompilační model je podobný standardu paradigma C. Je to imperativní programovací jazyk. Kompilací GLSL kódu získáme GLSL objekty. GLSL objekt zapouzdřuje kompilované či nalinkované programy, které zpracovávají část řetězce.

GLSL má složitější kompilační model než jiné jazyky pro grafické karty. Ostatní jazyky mají jednofázový model, kdy pro danou fázi grafického řetězce (vertex shader, fragment shader) stačí pouze jeden řetězec (kód shaderu). Tento řetězec se zkompiluje a získá se objekt shaderu (v angličtině shader object), který se pak jen naváže na kontext.

V jazyce C se v jednoduchém případě z jednoho zdrojového souboru vytvoří jeden objektový soubor. Ve složitějším případě zdrojový soubor může obsahovat reference na některé externí hlavičkové soubory. Pak se zkompiluje několik zdrojových souborů do několika objektových souborů a objektové soubory jsou pak mezi sebou propojeny do jednoho programu.

Kód GLSL se kompiluje podobným způsobem jako kód jazyka C. Několik zdrojových souborů se zkompiluje do objektu shaderu, což je jako analogie k objektovému souboru. Pak se tyto objekty musí nalinkkovat. GLSL model dokáže několik takovýchto objektů spojit do jednoho programu. Pokud nemá shader ve svém kódu funkci main, pak není aktivní (program vykresluje tak, jakoby shader chyběl). Za podmínky, že je v programu použito `glEnable(GL_RASTERIZER_DISCARD)`, je možné použít grafický řetězec pouze s vertex shaderem [4].

2.2.2 Standardní knihovní funkce

Existuje mnoho standardních funkcí. Některé jsou specifické pouze pro danou fázi shaderů, ale většina z nich jsou použitelné ve všech fázích.

Některé funkce, jako například vzorkování textury, se nedaly používat v nižších verzích OpenGL ve vertex shaderu. Od verze OpenGL 2.0 je toto již umožněno [6]. Pokud překladač stále hlásí chybu, může to být omezením hardware nebo ovladačů.

Matematické funkce se většinou dají používat ve všech fázích výpočtu. Jde například o často používané funkce pro lineární interpolaci (`mix`), skokovou funkci (`step`) nebo matematické operace s vektory (skalární součin `dot`, vektorový součin `cross`).

2.2.3 Předdefinované proměnné

Existuje určitý počet speciálních proměnných, které jsou již definované jazykem. Jsou používány většinou pro komunikaci s vnějšími částmi grafického řetězce, na které navazuje programový kód shaderů. Díky konvenci začínají všechny předdefinované proměnné řetězcem `gl_`. Uživatelem definované proměnné tak začínat nesmí.

V programovém kódu shaderů bude programátor ve většině případů využívat alespoň proměnnou `gl_Position`. Pokud chceme, aby proběhlo vykreslování, musí se tato proměnná nastavit v kódu shaderu, který předchází fragment shaderu (geometry shader a tessellační shadery jsou nepovinné, takže to může být jakýkoliv z předešlých tří shaderů). Pokud se této proměnné nepřiřadí platná `vec4` hodnota, neuvidíme při rasterizaci žádný výstup.

2.2.4 Datové typy proměnných

GLSL definuje mnoho datových typů. Zároveň definuje způsob, který je možné vytvářet vlastní datové typy (struktury). Vektorové a maticové datové typy jsou složeny ze základních datových typů.

- základní typy
  - skalární
    - `bool` (true/false)
    - `int` (znaménkový 32-bitový int s dvojkovým doplňkem)
    - `uint` (neznaménkový 32-bitový int)
    - `float` (číslo v plovoucí řádové čárce podle IEEE-754)
  - vektorové (n je 2, 3 nebo 4)
    - `bvecn` (vektor bool hodnot)
    - `ivecn` (vektor znaménkových int)
    - `uvecn` (vektor neznaménkových int)
    - `vecn` (vektor float čísel)
    - `dvecn` (vektor double čísel)
  - matice
    - `matnxm`
    - `matn`
  - neprůhledné (anglicky opaque)
    - vzorkovače
    - obrázky
    - atomické čítače
- pole
- pole polí

Skutečný datový typ je dán až po překladu. Velikosti a formáty integer a float čísel jsou dány až od GLSL 1.3 a výše. Nižší verze mohou používat různé specifikace.

Pro pohodlné zacházení s proměnnými mají vektory velké množství konstruktorů. Čtyřsložkový vektor lze složit ze dvou dvousložkových vektorů nebo z třísložkového vektoru a skaláru apod. Lze také vytvořit třísložkový vektor z čtyřsložkového (ignoruje se poslední hodnota).

Při nastavování hodnot vektorovým proměnným nám pomáhají vlastnosti glsl jazyka nazvané swizzling a masking. Swizzling umožňuje přeházet složky vektoru (hodnoty na pravé straně výrazu) bez psaní složitých a dlouhých konstruktorů. Masking naopak umožňuje přeházet hodnoty na levé straně výrazu, čímž se nastaví, do kterých hodnot se má zapisovat.

```
vec4 vector4 = another_vector.xxyw;
vector4.xw = vector2;
```

Pro operace s vektory jsou operátory přetíženy tak, aby se operace prováděly po složkách. Oba vektory musí mít však stejný počet položek. Násobení vektorů skalárem pak má očekávatelný efekt, kdy se všechny složky vektoru vynásobí skalárním číslem.

U matic se dá přistupovat k jednotlivým sloupcům (`Matrx[col]`) nebo k jednotlivým hodnotám (`Matrx[col][row]`).

2.2.5 Kvalifikátory proměnných

GLSL jazyk specifikuje obsahuje mnoho klíčových slov, které mění nejrůznější vlastnosti datových typů. Modifikují lokálně a globálně definované proměnné. Kvalifikátory mohou ovlivňovat zdroj dat pro proměnné, zarovnání v paměti či sdílení.

Kvalifikátory ukládání

Lokální proměnné bez kvalifikátoru se chovají jako standardní lokální proměnné, které jdou inicializovat při deklaraci a měnit.

- `const` (proměnné se nemohou během jejich existence změnit, musí být inicializovány při deklaraci výrazem, který musí být opět konstanta)
- `in`, `out` (proměnné pro vstup a výstup mezi fázemi, nemůže být použito na lokální proměnné)
- `attribute` (použitelný pouze v kompatibility módu a pouze ve vertex shaderu, nahrazuje kvalifikátor in)
- `uniform` (globální proměnné nastavené uživatelem v OpenGL API, hodnota se nemění během vykreslování, společné pro všechny fáze)
- `varying` (použitelný pouze v kompatibility módu, nahrazuje out ve vertex shaderu a in ve fragment shaderu)

- `buffer` (pouze od GL 4.3, umožňuje sdílet informaci buffer objektů mezi GLSL kódem a OpenGL API)

Vstupní a výstupní proměnné mohou mít ještě specifikován navíc jeden ze tří interpolačních kvalifikátorů – `smooth`, `flat` nebo `noperspective`. Kvalifikátor `smooth` je přednastavený, kvalifikátor `flat` neprovádí žádnou interpolaci a kvalifikátor `noperspective` provádí lineární interpolaci bez ohledu na perspektivu.

Kvalifikátory rozvržení

Kvalifikátory pro rozvržení ovlivňují, kde je ukládací prostor pro proměnnou. Kvalifikátory rozvržení mohou mít i přiřazené hodnoty. Existuje mnoho forem jejich deklarace:

- Rozvržení rozhraní – specifikuje indexy hodnot, např.
  ```
  layout(location=0) in vec4 color;
  layout(location=1) in vec2 texCoord;
  ```
- binding points – bloky v paměti mají index, který může být navázaný na binding point, bloky shaderu mohou být navázány na stejný binding point – dojde k vytvoření sdíleného bloku mezi shadery
- Formáty textur – tři hlavní formáty
  - Floating point
  - Signed integer
  - Unsigned integer
- Atomický čítač – lze inkrementovat v shaderu a použít na sčítání pixelů, chyb nebo jiných charakteristik ve výsledném obrázku
- Uniform blok (rozvržení bloků v paměti)
  - `shared` (přednastavené, žádné optimalizace a rozložení v paměti je vždy stejné mezi různými programy, pokud mám stejné typy a pořadí proměnných a explicitně zadané délky polí)
  - `std140` (stejně jako shared se snaží o přenositelnost, nepoužité uniformy nejsou eliminovány)

  - `packed` (každá proměnná v bloku má byte offset, může mít optimalizace, nedá se sdílet mezi různými programy)

Kvalifikátory parametrů

Parametry funkcí mohou mít rovněž svoje kvalifikátory. Parametry funkcím můžeme předávat například

- `in` – hodnotou
- `inout` – odkazem
- `out` – odkazem bez inicializace
- `const` – hodnota parametru se ve funkci nemůže měnit

3 GLSL shadery a fixní část grafického řetězce

Jak již bylo dříve řečeno, shadery jsou specifické programy psané pro určité programovatelné části grafického řetězce. Zkompilované a propojené dohromady tvoří program, který převede vstupní data na počátku grafického řetězce na výsledné pixely na obrazovce. Shadery se spouštějí postupně a každý následující přijímá výstupní data předchozího. Není povinné specifikovat program pro všechny shadery.

3.1 Vertex fetch

Tato fáze proběhne před vertex shaderem, který je první programovatelnou částí grafického řetězce. Úkolem této fáze automaticky poskytuje vstupní data vertex shaderu. Kromě hodnot vrcholů musí posílat vertex fetch také další hodnoty vrcholů (normály, barvy, atd.). Těmto různým skupinám dat se říká atributy. Musíme nastavit, jaká data v paměti grafické karty jsou asociována s jakým výstupem této fáze, a tedy vstupem do vertex shaderu. Natavení vertex fetch ve skutečnosti probíhá buď použitím OpenGL příkazu `glBindAttribLocation` nebo přímo ve vertex shaderu použitím kvalifikátoru `layout(location=0)` [2].

3.2 Vertex shader

Vertex shader zpracovává jednotlivé vrcholy, které vstupují do grafického řetězce. Pořadí zpracovaných vrcholů nelze ovlivnit. Každý vrchol se zpracovává bez informace o dalších vrcholech nebo jeho sousedech. z toho vyplývá, že nelze napsat výpočet, který by využíval některý z atributů vrcholů jako uspořádanou nebo jen sdílenou proměnnou. Zároveň v téhle části programu ani shader neví, kterému primitivu vrchol patří (primitivem je myšlen trojúhelník, trojúhelníkový strip, atd.). Programátor tedy píše kód, který bude vykonán paralelně pro každý vrchol zvlášť. Pro každý vstupní vrchol program vyprodukuje výstupní vrchol. Vrcholy budou často obsahovat dodatečné informace ve formě atributů.

Nejčastějšími vstupními atributy jsou pozice, normála, texturovací souřadnice nebo barva. Popřípadě mohou mít vrcholy definovány další dodatečné hodnoty, které mohou být užitečné v dalších fázích grafického řetězce. Standardně budeme chtít ve vertex shaderu měnit pozici a normálu vrcholů na základě matic geometrických transformací. Hodnoty jako barva a texturovací souřadnice vrcholu budou většinou pouze poslány do další fáze beze změny.

Kromě vlastních uživatelsky definovaných proměnných jsou přednastavené následující atributy:

```
in int gl_VertexID
in int gl_InstanceID
```

Atribut `gl_VertexID` obsahuje index vrcholu. Při použití instancování atribut `gl_InstanceID` obsahuje index současné instance (bez použití instancování vrací vždy nulu).

Grafická karta vnitřně využívá indexy vrcholů pro urychlení výkonu. `Atribut gl_VertexID` obsahuje stejnou hodnotu jakou má vrchol přiřazenou v poli indexů. Předtím, než je vrchol zpracován se porovná jeho ID s dalšími indexy, které jsou uloženy v cache. Pokud je v cache uloženo, že vrchol na tomto indexu se již zpracoval, nespouští se znovu výpočet pro stejný vrchol a pouze se vrátí uložený výsledek z cache [2].

Dále jsou definovány vnitřní výstupní atributy vrcholů, do kterých může vertex shader zapisovat:

```
outgl_PerVertex
{
    vec4 gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[];
};
```

Zapisování do těchto atributů není povinné. Pokud však je další fáze fragment shader, pak se očekává, že bude zapsáno do `gl_Position`, aby fungovala vnitřní funkcionalita rasterizace. `Atribut gl_Position` ukládá výstupní souřadnici vrcholu v homogenních souřadnicích.

Uživatel může definovat vlastní vstupní a výstupní atributy a jejich typy pro každý vrchol.

```
in vec3 positionIn;
in vec3 normalIn;
in vec2 texCoordIn;
out vec2 texCoordOut;
out vec3 normalOut;
```

Uživatel dále může definovat pojmenovaný blok, podobně jako strukturu v jazyce C.

```
outVertexData
{
    vec2 texCoord;
    vec3 normal;
} VertexOut;
```

3.3 Tessellation shader

Napsat a nastavit tessellační část grafického řetězce je nepovinné. Tessellaci použijeme, pokud chceme jedno primitivum rozdělit na několik menších.

Tesselační shader přijímá záplaty (patches) a generuje na jejich základě nová primitiva (body, čáry, trojúhelníky). Na rozdíl od jiných primitiv mají záplaty uživatelem specifikovaný počet vrcholů. Uživatel použije funkci glPatchParameteri, aby nastavil, kolik vrcholů bude záplata mít. Tato hodnota je pak konstantní v průběhu volání funkce pro vykreslení.

```
glPatchParameteri(GL_PATCH_VERTICES, pocetVrcholuNaZaplatu );
```

Záplata je pole vrcholů (řídících bodů), jejichž atributy jsou vypočítány vertex shaderem. Tessellační shader pak většinou rozdělí záplatu na menší primitiva. Existují dva druhy záplat – trojúhelníkové záplaty a kvady. Počet vrcholů v záplatě však nezávisí na jejím druhu.

Tessellační část řetězce se dělí na tři další části:

- tessellationControl
- Primitive Generation
- tessellationEvaluation

Uživatel může napsat vlastní shader pro první a třetí část tessellace. Fáze Primitive Generation programovatelná není.

3.3.1 Tessellation control shader

Tessellation control shader (dále jen TCS) přijímá vstupní záplatu, kterou tvoří pole řídících bodů. Tyto body nemusí nutně tvořit body na výsledném povrchu vykreslovaného primitiva. TCs vypočítá atributy pro každý vrchol výstupní záplaty. Jeho výstupem je tedy pole řídících bodů (s veškerými jejich atributy, které chceme případně předat). Počet řídících bodů v této nové záplatě, které vystupují z TCS, se může lišit od počtu řídících bodů, které do něj původně vstoupily. Počet řídících bodů `N`, které TCs generuje, se nastaví v jeho kódu kvalifikátorem layout:

```
layout (vertices = N) out;
```

TCS se spouští se znalostí všech řídících bodů ve vstupní záplatě. Jeho výstupem je však pouze jeden řídící bod. Číslo `N` udané v kvalifikátoru layout tedy udává počet volání (invokací) TCS. Pro indexování výstupních řídících bodů, které budeme zapisovat do výstupní záplaty, použijeme proměnnou `gl_InvocationID`. Jak její název napovídá, tato hodnota bude mít různou hodnotu indexu v rámci jedné záplaty pro každou invokaci TCS. Různá volání TCS dokonce mohou sdílet výstupní hodnoty – mohou číst výstupní hodnoty volání právě probíhající fáze, která zapisuje do stejné záplaty. Aby se ale mohlo toto sdílení uskutečnit, je nutné použít synchronizační mechanismy.

Zároveň tento shader zapíše pro každou záplatu (nikoliv vrchol) atributy, které definují úroveň dělenítéto záplaty. Tyto atributy se nazývají úrovně tessellace (anglicky tessellation levels), dále jen TL. Zapisují se do předdefinovaných výstupních proměnných `gl_TESsLevelInner` a `gl_TESsLevelOuter`.

```
#version 430 core
layout (vertices = 3) out;
void main(void)
{
    if(gl_InvocationID == 0)
    {
        gl_TESsLevelInner[0] = 5.0;
        gl_TESsLevelOuter[0] = 5.0;
        gl_TESsLevelOuter[1] = 5.0;
        gl_TESsLevelOuter[2] = 5.0;
    }
    gl_out[gl_InvocationID].gl_Position =
     gl_in[gl_InvocationID].gl_Position;
}
```

Tabulka 3.1: Ukázka tessellation control shaderu

Protože `gl_TESsLevelInner` a `gl_TESsLevelOuter` jsou hodnoty společné pro celou záplatu, stačí do nich zapsat pouze jednou, například při první invokaci (invokace s indexem 0). Jako vstupní a výstupní pole TCS můžeme použít předdefinované `gl_in` a `gl_out`.

TCS tedy vygeneruje pole transformovaných kontrolních bodů (vrcholů záplaty) a dále pro každou záplatu vygeneruje množinu čísel zvanou TL (znázorněno na ukázce 3.1).

3.3.2 Generátor primitiv

Generátor primitiv (dále jen PG) je část tessellačního shaderu, která není programovatelná (lze pouze konfigurovat výstupní topologii). Tato část grafického řetězce přijme množinu čísel TL, která je výsledkem TCS a na jejím základě a zvolené topologii vygeneruje souřadnice u, v v parametrickém prostoru. Zároveň určí konektivitu vrcholů.

Souřadnice u, v jsou normalizované na intervalu $\langle 0; 1\rangle$ a parametrický prostor tvoří buď čtverec nebo trojúhelník v případě barycentrických souřadnic.

Generátor primitiv je část, která se dá nastavit tak, aby vracela body nebo trojúhelníky. Pokud má vracet body, pouze vezme hodnoty z TL a vygeneruje

```
layout (triangles, equal_spacing, cw) in;

#version 430 core
layout (triangles, equal_spacing, cw) in;
void main(void)
{
    gl_Position = (gl_TESsCoord.x * gl_in[0].gl_Position +
    gl_TESsCoord.y * gl_in[1].gl_Position +
    gl_TESsCoord.z * gl_in[2].gl_Position);
}
```

Tabulka 3.2: Ukázka tessellation evalutation shaderu

uv souřadnice pro body a pošle je dále. Pokud má vracet trojúhelníky, posílá dále body po třech jako trojúhelníky.

3.3.3 Tessellation evaluation shader

Tessellation evaluation shader (dále jen TES) má přístup k transformovaným vrcholům z TCS a k *uv* souřadnicím z PG. Jedna instance TES se spustí pro každý bod vygenerovaný pomocí PG. Úkolem TES je pak z vygenerovaných údajů sestavit hodnoty výstupního vrcholu. Jelikož má TES přístup k vrcholům z TCS, může udělat více než jen vypočítat absolutní pozici lineární interpolací pozice vrcholů. Do výsledného výpočtu může TES použít například normály, nebo jakékoliv jiné informace uložené ve vrcholech.

TES musí vygenerovat právě jeden vrchol za invokaci. Nemůže se rozhodnout vrchol vypustit. Provede tedy invokaci pro každý vrchol, který má vzniknout ve výstupní záplatě. Je nutné dát pozor na vysoké hodnoty TL v kombinaci se složitými polygonálními modely nebo se složitým TES, protože v konečném důsledku může díky velkému množství vrcholů znatelně trpět výkon.

Nastavení v ukázce 3.2 na počátku TES říká, že jako mód zvolíme trojúhelníky. Nové vrcholy by měli být generovány rovnoměrně mezi vrcholy polygonu, který rozbíjíme na menší primitiva a uspořádání vrcholů v primitivu je po směru hodinových ručiček [2].

3.4 Geometry Shader

Geometry Shader (dále jen GS) je stejně jako tessellační shadery nepovinný. Je to poslední fáze před rasterizérem. Proběhne vždy jedna invokace pro jedno primitivum (například trojúhelník).

Na vstupu GS jsou jednotlivá primitiva z předchozí fáze. Pokud předcházející fáze generovala trojúhelníkové stripy, vějíře nebo jiné smyčky, GS přijme trojúhelníky. Na rozdíl od Vs má GS navíc i informaci o všech vrcholech náležících k jednomu primitivu a navíc informace o sousednosti, pokud jsou specifikovány. Dokáže produkovat jiné typy primitiv, než jaká přijímá. Například dokáže vyprodukovat samostatné body z trojúhelníků nebo naopak trojúhelníky z bodů.

Možná vstupní primitiva pro GS:

- body (1 vrchol)
- čáry (2 vrcholy)
- čáry se sousedností (4 vrcholy)
- trojúhelníky (3 vrcholy)
- trojúhelníky se sousedností (6 vrcholů)

Možná výstupní primitiva z GS:

- body
- lomené čáry
- trojúhelníkové pásy

GS je jediná fáze v grafickém řetězci, která dokáže programově vygenerovat primitiva navíc nebo nevygenerovat žádná primitiva. Tessellační shadery to provádějí také, ale pouze automaticky na základě nastavení PG. GS může použít funkci `EmitVertex()`, která vezme všechny nastavené vústupní proměnné a prohlásí, že byly výchozí hodnoty jednoho vrcholy nastaveny a

```
#version 430 core
layout (triangles) in;
layout (points, max_vertices = 3) out;
void main(void)
{
    int i;
    for (i = 0; i < gl_in.length(); i++)
    {
        gl_Position = gl_in[i].gl_Position;
        EmitVertex();
    }
}
```

Tabulka 3.3: Ukázka geometry shaderu

funkci `EndPrimitive()`, která vezme všechny takto vyprodukované vrcholy a vytvoří z nich primitivum. Toto primitivum je pak posláno do rasterizéru [2].

Kód v ukázce 3.3 ukazuje program GS, který přijme trojúhelníky a rozdělí je na jednotlivé body. První kvalifikátor `layout` říká, že na vstupu jsou trojúhelníky. Druhý kvalifikátor `layout` říká, že na výstupu jsou body a jejich maximální počet. Pokud bychom chtěli vykreslovat trojúhelníky, musíme u kvalifikátoru `layout` specifikovat, že chceme vykreslovat trojúhelníky místo bodů. Navíc musíme po nastavení tří vrcholů (a následném zavolání funkce `EmitVertex()`) zavolat funkci `EndPrimitive()` pro ukončení trojúhelníku. Tato funkce se volá explicitně na konci shaderu, ale pokud generujeme více trojúhelníků, musíme jí použít. Maximální počet stále udává maximální počet vrcholů, které GS generuje.

3.5 Primitive assembly, ořezávání a rasterizér

Tato část řetězce vezme výstupní vrcholy z předešlé fáze a informaci o jejich konektivitě vrcholů. Primitive assembly není programovatelná část grafického řetězce. Její chování se však dá určit nastavením parametru funkce `glDraw`. Po tom, co bylo primitivum složeno z jednotlivých vrcholů, je ořezáno na

základě jeho pozice a zobrazitelné části. Velikost zobrazitelné části udává takzvaný viewport, jehož pozice a velikost se dá nastavit příkazem OpenGL `glViewport`. Dále proběhne převod vrcholů z homogenního souřadného systému do normalizovaného euklidovského prostoru. Jakýkoliv vrchol, který se má vykreslit, má hodnoty všech zbylých tří souřadnic v intervalu od nuly do jedné. Jakýkoliv bod vně se má vyloučit. Nejprve ale proběhne ořezání, které se v této práci nebude probírat do hloubky. K ořezání geometrie se použije 6 poloprostorů, které korespondují se stěnami krychle, která tvoří normalizovaný prostor.

U všech zbylých částí geometrie se určí viditelnost na základě souřadnice *z*. Části, které jsou viditelné, se pošlou rasterizéru. Ten určí, které pixely pokrývají pozice primitiv a pošle fragment shaderu jejich seznam (spolu s ostatními interpolovanými informacemi získanými z vrcholů) [2].

3.6 Fragment shader

```
uniform sampler2D mySampler;
in vec2 texture_coordinate;
out vec4 outputColor;

void main()
{
    outputColor = texture2D(mySampler, texture_coordinate);
}
```

Tabulka 3.4: Ukázka fragment shaderu

Fragment shader je poslední programovatelná fáze grafického řetězce. Tato fáze má pouze nastavit barvu fragmentů před tím, než jsou zaslány do bufferu, jehož obsah se nakonec zobrazí na obrazovce.

Můžeme zde používat přednastavené proměnné jako `gl_FragCoord`, které obsahují pozici fragmentu v okně. V praxi však většinou budeme počítat barvu pixelů z osvětlení a z textur, které jsou aplikovány na vykreslované povrchy. Pro osvětlení se využijí normály, které jsme transformovali zároveň s pozicemi vrcholů. Fragment shader nám nyní poskytuje interpolované normály pro každý fragment. Stejně tak když potřebujeme aplikovat textury, tak

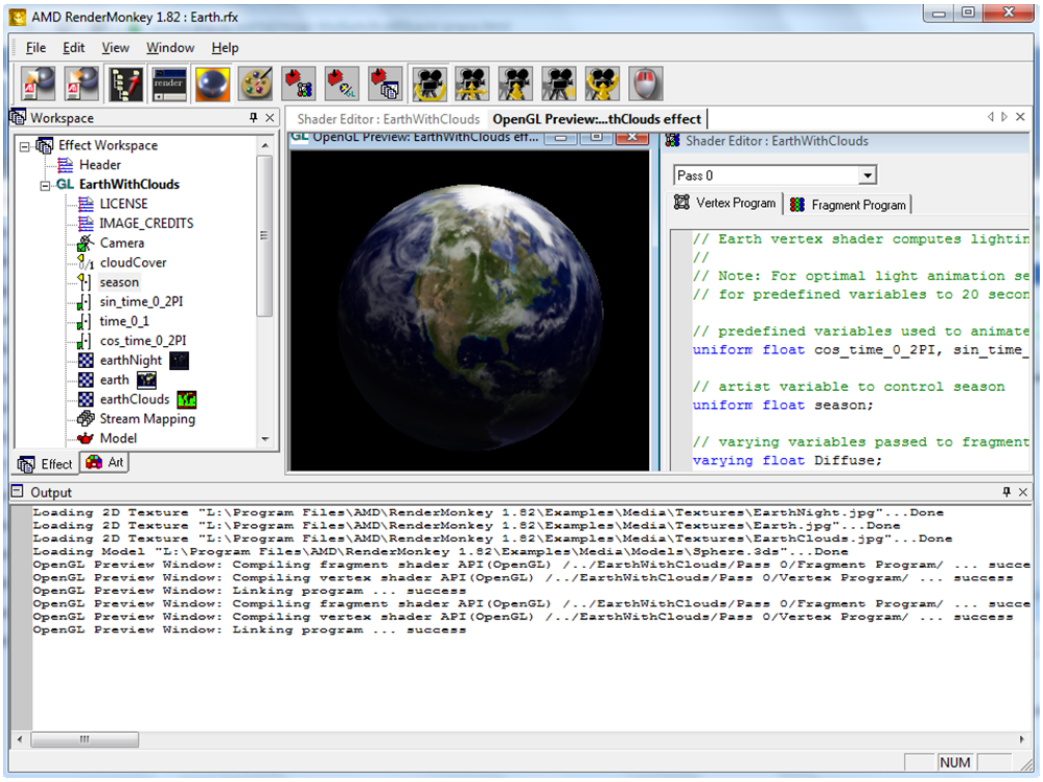
budeme používat texturovací souřadnice, které byly uloženy ve vrcholech a nyní jsou interpolovány ve všech fragmentech, které korespondují s viditelnými povrchy [2].

Kód v ukázce 3.4 ukazuje, jak nastavit výstupní barvu fragmentu. Vstupní proměnná `texture_coordinate` obsahuje texturovací souřadnici, kterou jsme museli propagovat z předešlých fází grafického řetězce. Obvykle je uložena ve vrcholu, ale můžeme s ní během průchodu různými shadery různě manipulovat. Uniform proměnná `mySampler` obsahuje vzorkovač, který je provázaný s texturou.

4 Nástroje pro editaci shaderů

V současné době existuje několik různých nástrojů pro editaci a ladění shaderů. Liší se výčtem svých funkcí a někdy i podporou programovacích jazyků pro grafické karty. Některé z těchto nástrojů jsou zaměřené více na vývoj jednoduchých aplikací, jiné jsou zaměřené na ladění programů.

4.1 AMD RenderMonkey (verze 1.82) [7]



Obrázek 4.1: Rozhraní nástroje RenderMonkey

Program přestal být vyvíjen. Poslední verze je z data 18. 12. 2008. Jediný z Testovaných programů, u kterého neexistuje verze pro Linux. Obrázek 4.1 ukazuje pohled na rozhraní. AMD RenderMonkey byl vytvořen tak, aby umožňoval či obsahoval následující funkce:

- vývojové prostředí pro vytváření shaderů
- mechanismus pro ulehčení sdílení shaderů mezi vývojáři
- flexibilní a rozšiřitelná platforma, která podporuje integraci vlastních komponent a poskytuje základ pro budoucí vývoj
- prostředí, kde mohou na vývoji efektů pracovat programátoři, umělci i herní designéři
- nástroj, který se dá jednoduše nastavit a integrovat do pracovního postupu vývojáře

Na stránkách AMD je oznámeno, že vývoj RenderMonkey byl ukončen. v poslední verzi podporuje všechny ShaderModely poskytované do DirectX 9.0c (Shader Model 1.0 až Shader Model 3.0). Lze použít programovací jazyky:

- HLSL (do verze DirectX 9.1)
- GLSL(do verze 2.0)
- Es (2.0)

4.1.1 Funkce a charakteristiky software

- schopnost exportovat GLSL efekty jako:
  - collada soubory (formát pro interaktivní 3D model definovaný otevřeným XML formátem),collada importér podporuje pouze trojúhelníkové sítě
  - soubory fx (Microsoft DirectX 9.0)
- podpora HW tessellace pro verzi ATI Radeon 2000 a výše
- možnost připojit či naprogramovat plug-in (vygeneruje se projekt Microsoft Visual Studio 2005)
- podpora pro shader model 3 (vzorkování textur ve vertex shaderu)
- knihovna příkladů pro HLSL, GLSL, Es a assemblylanguage

- vykreslování na celou obrazovku
- zobrazování textur 2D, 3D a kubických map
- generátor srsti / šupin
- hlášení run-time chyb
- editory pro vývoj:
  - vertex shaderu
  - fragment shaderu

4.1.2 Rozhraní

- workplace – abstraktní strom pro uložení projektu
  - modely
  - textury
  - efekty
    - parametry efektů
    - vertex shader
    - fragmentshader (pixel shader)
- okno náhledu na zobrazení modelu a efektu
- editory pro shadery a GUI editory pro parametry shaderů
- okno zpráv
  - kompilace
  - chyby
  - textové zprávy z aplikace
- artist editor – editor parametrů efektů

4.1.3 Editor

Doplňování kódu a našeptávač je přítomný pouze v nejvíce primitivní formě, kdy našeptávají pouze složky vektorů x, y, z a w po napsání tečky. Nikdy nenašeptává rezervovaná slova, funkce nebo proměnné. Syntaktické chyby nejsou podtrhávány. Chyby při kompilaci se nedají dohledávat dost jednoduše. Editor postrádá schopnost označit klíčové slovo nebo řádku, ve které nastala chyba. V logu se pouze vypíše popis chyby, který je smísený s dalšími zprávami o kompilaci. Chybová hlášení nejsou od dalšího textu nijak zvýrazněna. V chybovém hlášení se zobrazí popis chyby a odkaz na řádku. Bohužel v okně s programovým kódem není číslování řádek a ani nikde není zvýrazněno číslo řádky, na které se nachází kurzor.

4.1.4 Ladění shaderů

Pro vykreslení scény musíme mít minimálně model (popř. textury a další zdroje), který použijeme na vstupu, vertex shader a fragment shader. Na model se aplikují textury a shadery a následná vizualizace probíhá ve smyčce. V nastavení (Edit/Preferences/General) lze nastavit časový interval pro vykreslovací smyčku, popřípadě interval pro obnovování textur a modelu. Tato funkcionalita napomáhá uživateli provádět změny za běhu vizualizace. Kód shaderů lze také měnit za běhu. Řídící kód vizualizační smyčky se provádí interně a nelze jej upravovat.

Scéna lze animovat pouze pomocí použití proměnné pro čas. Do projektu se vloží nová proměnná (float) se sémantikou čas (`Time0_X`), a tato proměnná se použije v kódu shaderu. Každý snímek se provede inkrementace této hodnoty. Protože se po každé sekundě hodnota zvýší o 1, není tento způsob závislý na rychlosti snímkování.

Složitější efekty vyžadují několik průchodů. V takovémto případě se musí nastavit počet průchodů a pro každý průchod nastavit nový vertex shader, fragment shader a popřípadě další parametry grafické pipeline (musí se přidat takzvaný „render state block“, který nastavuje stav OpenGL či DirectX). Všechny shadery musí být zadány staticky a musí se zkopírovat. Nelze použít jeden objekt shaderu vícekrát, a to ani přesto, že chceme v různých průchodech například použít identický vertex shader.

Takovéto chování je jednoduché pro použití, ale není dostatečně flexi-

bilní. Nelze provádět žádné pokročilejší operace, jako například programově zadaný počet průchodů. Zobrazování více objektů se dá emulovat pouze tím způsobem, že se využije systém průchodů – každý další objekt se vykreslí v novém průchodu. Samozřejmě je tím znemožněno například zobrazování procedurálně generovaných objektů, protože každý objekt musí být nastaven zvlášť v okně Workplace.

4.1.5 Export

RenderMonkey umožňuje export zdrojů do souborů fx nebo collada, které následně mohou použít další aplikace. Exportem se ale nevytvoří aplikace se zobrazovací smyčkou, získáme jím pouze tyto soubory. Pokud bychom chtěli napodobit programově vizualizaci programu Render Monkey, museli bychom sami naprogramovat zobrazovací smyčku a v ní použít tyto shadery.

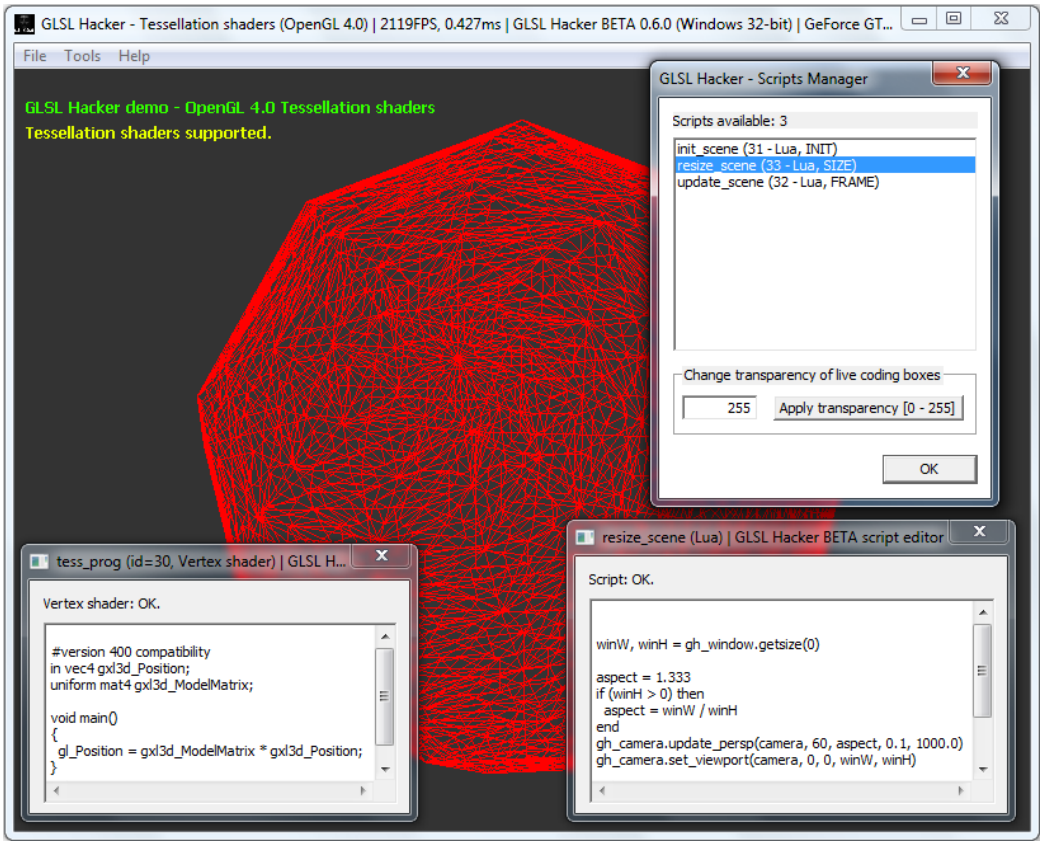
4.2 Glsl Hacker (verze 0.5.0) [8]

Byla vyzkoušena verze z 9. 5. 2013. Nástroj navržen pro rychlé 3D prototypování shaderů. Glsl Hacker má svoje API na nižší úrovni abstrakce. Je zapotřebí více kódu, ale tím poskytuje více možností, co se týče vykreslování scény. Bohužel, tyto možnosti je možno využít jen pokud chceme sestavit scénu ve skriptovacích jazycích Python nebo Lua. Tato verze existuje pouze na 64bitové systémy. Poslední vyvíjená verze 0.6.0 bude spustitelná i na 32bitových systémech. Obrázek 4.2 ukazuje pohled na rozhraní.

Tato verze neobsahuje skoro žádné grafické rozhraní. Má stavovou řádku, která umožňuje napsat příkaz ve skriptovacím jazyce a ihned jej vykonat. V jazycích Lua nebo Python se napíše jeden či několik inicializačních skriptů a dále se napíše jeden či několik skriptů pro vykreslování snímků.

4.2.1 Funkce a charakteristiky software

- umožňuje měnit za běhu aplikace:
  - GLSL programový kód
    - vertex shader



Obrázek 4.2: Rozhraní nástroje GLSL Hacker

    * fragment shader
    * geometry shader
    * tessellation control shader
    * tessellation evaluation shader
    * computation shader
  - vykreslovací smyčka (Lua skript nebo Python skript)
    * init
    * resize
    * update

- Síťové připojení pro ladění programu na jiném stroji

4.2.2 Rozhraní

Rozhraní je velice minimalistické. Obsahuje pouze hlavní nabídku a plochu, kde se bude zobrazovat vykreslená scéna. V hlavní nabídce je volba načítání scény a na zobrazení nápovědy či dokumentace. Hlavní funkcionalita tohoto software je dostupná v nástrojích. Zde se dá editovat skript inicializace, zobrazovací smyčky i GLSL kód. Editace kódu se může provádět buď na příkladu, který je spuštěný přímo v aplikaci, nebo je možné se připojit přes síť na jiný server, na kterém je tento program spuštěný.

4.2.3 Editor

Editor je velmi minimalistický. Nemá žádné zvýrazňování kódu, žádné našeptávání kódu ani žádné funkce na rozbalování funkčních bloků. Jedná se pouze o jednoduchý textový box. Dokonce užívá neproporcionální font, který se normálně pro programový kód nepoužívá.

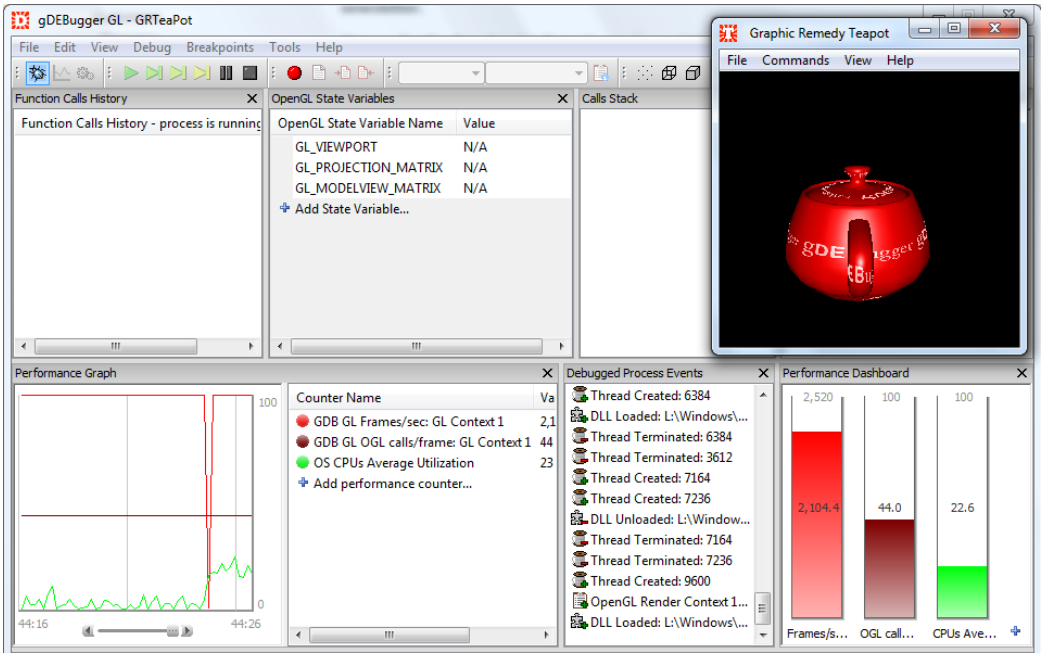
Další vlastností tohoto editoru je, že neobsahuje vůbec žádné ovládací prvky. Volby Zpět a Vpřed obsahuje pouze po pravém kliknutí do textového boxu, nemá žádné vyhledávání řetězců, nemá číslování řádek a nemá dokonce ani volbu uložení změn. Místo toho se změny aplikují vždy okamžitě. Pokud se v kódu objeví chyba, vizualizace jednoduše ukazuje poslední funkční verzi a v jednoduchém editoru se objeví pouze indikace, jestli je kód syntakticky v pořádku či nikoliv. V editoru, který je vytvořen pro síťovou verzi, je místo této řádky log událostí, ale chyby jsou rovněž bez jakéhokoliv popisu.

4.2.4 Ladění shaderů

Při ladění shaderů, jež využívají funkce, které nejsou hardwarově podporovány na příslušném stroji, se nedaří měnit kód shaderů. V takovém případě se dokonce ani neukládají změny.

V případech, kdy je vše podporováno je možno měnit shadery za běhu vizualizace. Vlastnost, která způsobuje, že se vizualizace aktualizuje po změně každého znaku v kódu shaderů, může být někdy velmi nepříjemná.

4.3 gDEBugger (verze5.8.1) [9]



Obrázek 4.3: Rozhraní nástroje gDEBugger

Profesionální aplikace pro ladění programů využívajících OpenGL a OpenCL. Testovaná verze vyšla 11. 12. 2012. Sleduje aktivitu nad OpenGL nebo OpenCL programem a poskytuje informace potřebné pro nalezení chyb v aplikaci nebo pro její úpravu za účelem optimalizace. Aplikace gDEBugger GL umožňuje sledovat aktivitu částí kódu v uživatelem spuštěné OpenGL aplikaci. Ukázka rozhraní je na obrázku 4.3.

4.3.1 Funkce a charakteristiky software

- profesionální nástroj na ladění grafických OpenGL aplikací
- schopnost krokování aplikací z pohledu volání OpenGL funkcí
- možnost upravovat GLSL kód shaderů
- tři základní módy – ladění, profilování a analyzátor paměti

4.3.2 Rozhraní

- historie volaných funkcí
- výkonnostní graf
- sledování OpenGL proměnných (projekční matice, modelview matice,. . .)
- zásobník volaných funkci
- log událostí
- vlastnosti (položek označených ve výkonnostním grafu)
- dashboard na zobrazování statistik (využití procesoru, snímky za sekundu, OpenGL volání za sekundu)

4.3.3 Editor

Editor programu

Editor zdrojového kódu má zvýrazňování syntaxe, postrádá ale našeptávání kódu. Zvýrazňování syntaxe se dá zvolit pro jazyky: Ada, Assembler, C++, CS, CSS, Fortran, GLSL, CL, HTML, Java, Javascript, Objective-C ,Pascal, Pearl, PHP, Python, Visual Basic a VB skript.

Má schopnost rozvinutí a kolapse kódu funkcí v editoru. Nedokáže podtrhávat syntaktické chyby během editace kódu. Není propojen s kompilátorem. Není tedy možné editovat kód aplikace za běhu. Je tomu tak vzhledem k tomu, že program je určen k ladění výsledných binárních aplikací.

Editor shaderů

Editování kódu za běhu je možné pouze u zdrojového kódu shaderů. V okně historie volaných funkcí lze označit funkci, která má shader jako svůj asociovaný parametr. Přes okno vlastností (properties) se můžeme dostat na editaci kódu shaderu.

Tento editor má zabudováno zvýrazňování syntaxe GLSL. Nemá žádné našeptávání kódu ani podtrhávání neplatné syntaxe. Pouze se po kompilaci

odkáže na číslo řádky, ve které nastala chyba. Výstup kompilace je vidět v přehledné tabulce. Na chybné řádky se dá přejít hypertextovým odkazem. Navíc se dá zapnout číselné označení řádek pomocí volby View. Editor dále dokáže zobrazovat mezery a odřádkování. Stejně jako druhá verze editoru pro zdrojový kód, dokáže provést rozvinutí a kolaps podprogramů.

Jakmile je shader zkompilován a sestaven, můžeme opět rozběhnout program a změny se okamžitě projeví na výstupu bez nutnosti restartovat program. Všechny změny, které uživatel učiní pomocí programu gDEBugger budou ale zapomenuty po restartování aplikace.

4.3.4 Ladění shaderů

Podporuje vertex shader, fragment shader a geometry shader.

Tato aplikace byla vytvořena zcela za účelem lazení. Většina informací se ale objevuje pouze, když je aplikace pozastavena. Pokud se aplikace pozastaví na break pointu, zastavení nenastane ihned, ale pouze na dalším OpenGL volání (při spuštěném módu pro ladění nebo pro analýzu) nebo na konci snímku (při spuštěném módu pro profilování).

Je možné přidávat break pointy na začátek OpenGL. To se vykoná přidáním v okně vyvolaném zkratkou Ctrl+B. Pokud si dáme breakpoint například jako glBegin, můžeme sledovat postup algoritmu buď po jednotlivých krocích (Single Step), přeskočit další funkce dokud nenarazíme na další vykreslení (Draw Step) nebo můžeme přeskočit na stejný breakpoint v dalším snímku (Frame Step).

Použitím tohoto prostředí se dá snadno provést oprava aplikace, protože můžeme najít a opravit problémy typu:

- chyby OpenGL (volba break on OpenGLErrors) – vrací OpenGL implementace; způsobeny špatnými výčty či voláním funkcí když nejsou relevantní
- detekované chyby – například pokud se používá pixel formát, který není hardwarově akcelerovaný
- nadbytečné změny OpenGL stavů
- používání nedoporučených OpenGL funkcí

- software fallback – nastává, pokud kombinace OpenGL stavů není podporována nebo akcelerována na grafickém hardware; v takovém případě se přepíná do softwarového módu a tím je značně snížen výkon

Když je aplikace vyčištěna od všech problémů způsobených voláním OpenGL funkcí, dá se nástroj použít pro detekování a opravení případů neuvolněných grafických objektů z paměti.

Nástroj gDEBugger dokáže navíc zobrazovat informace o texturách a bufferech. Tyto informace jsou sbírány za běhu a dají se zobrazit v okně vyvolaném Ctrl+T. Zde lze vybrat libovolnou texturu či buffer (DepthBuffer, Front Buffer, BackBuffer apod.), a pak v pravé části okna vidíme obsah textury či bufferu s přesnými informacemi o barvě v každém pixelu. Lze zapnout či vypnout kanály RGB, popřípadě kanál alfa. V druhém listu nazvaném Data View můžeme vidět obrázek pouze v numerických hodnotách. U 3D textur lze navigovat mezi vrstvami textury, podobě u pole textur. Lze zobrazit i Vertex BufferObject, ale pouze v režimuData View.

4.3.5 Statistiky

Program dokáže vypsat statistiky Ctrl+Shift+S. V těchto statistikách můžeme vidět procentuální poměr zastoupení volaných OpenGL funkcí dle jejich typu (texturovací, maticové, resterizační). Poměr zastoupení jednotlivých funkcí znázorněný ve sloupcovém grafu lze vidět v dolní části okna.

Toto okno má speciální záložku na zobrazení zastaralých (tzv. deprecated) funkcí, kde jsou všechny použité vypsány včetně detailů, kdy byla funkce takto označena, ve které verzi byla případně vymazána, počet volání funkce do aktuálního breakpointu či procentuální zastoupení vzhledem k ostatním funkcím.

Ostatní volané funkce můžeme vidět v záložce FunctionCalls, kde jsou uvedeny ke každé funkci další podrobnosti. Zejména jsou pak zvýrazněna doporučení případně nepoužívat nevhodnou a nahradit ji za jinou.

Další zajímavou statistikou je počet skupin vrcholů, které se během jednoho snímku posílají na zpracování. Je zvykem snažit se psát program tak, aby se najednou posílalo vždy co největší množství vertexů, čímž se zvýší výkon aplikace.

4.3.6 Profilování

V hlavním okně můžeme vidět časovou osu, na které je znázorněn výkonnostní graf. Existují tři skupiny čítačů:

- čítače specifické pro operační systém – sledují výkon z pohledu správy systémových zdrojů (správa paměti, využití procesoru apod.).
- čítače specifické pro grafický hardware – pouze použitelné pro některý hardware (NVIDIA, ATI/AMD, S3 Graphicsa 3DLabs), za podmínky že jsou instalovány příslušné ovladače.
- gDEBugger čítače – zde lze sledovat například počet snímků za sekundu, počet volání OpenGL funkcí a čítačů, počet trojúhelníků či vrcholů na snímek atd.

4.3.7 Analýza

Mód analýzy umožňuje sledování podrobných informací o funkcích na změnu stavu a vyhledávání redundantních použití takovýchto funkcí. Získávání informací pro použití tohoto sledování však má velký dopad na výkon aplikace.

Tento mód funguje podobně jako mód na získávání statistik. Zpomalení je způsobeno tím, že se musí vyhodnocovat a ukládat větší množství ukazatelů, aby se redundantní stavy odhalily. Ty se na rozdíl od obyčejných statistik, kdy se sčítá počet výskytů volání, musí obohatit o porovnávání hodnot ve všech předchozích stavech. Díky těmto porovnáním se může například zjistit, že dané konkrétní volání vždy nastavuje stav na stejnou hodnotu, jaká byla před tímto voláním.

Okno na statistiky získané v módu analýzy se vyvolává pomocí zkratky Ctrl+Shift+S. Jedná se stále o stejné okno. Tentokrát nás zajímá hlavně záložka StateChange, která zde může mít několik záznamů o volání, jež ve 100% případů nastavila stejný stav, který byl před zavoláním daného nastavení.

4.3.8 Následníci projektu gDEBugger

gDEBugger AMD (verze 6.2) [14]

Verze gDEBuggeru pro AMD grafické karty, která vyšla 20. 4. 2012. Rozhraní této verze se liší od původní. Nemá například rozdělení na tři módy (ladění, profilování a analýza paměti). Není možné spustit profilování a sledovat tak zvolené proměnné v grafu při běhu aplikace.

Z pohledu vlastností, které je vhodné uvážit pro zpracování této diplomové práce, nepřináší aplikace nic nového oproti její druhé verzi. Zdá se, že spíše některé funkce ztrácí (kompilace shaderů a profilační mód). Na stránce projektu je uvedeno, že již nebudou žádné další verze gDEBuggeru, kromě možných oprav chyb. Projekt, který je následníkem gDEBuggeru se nazývá CodeXL.

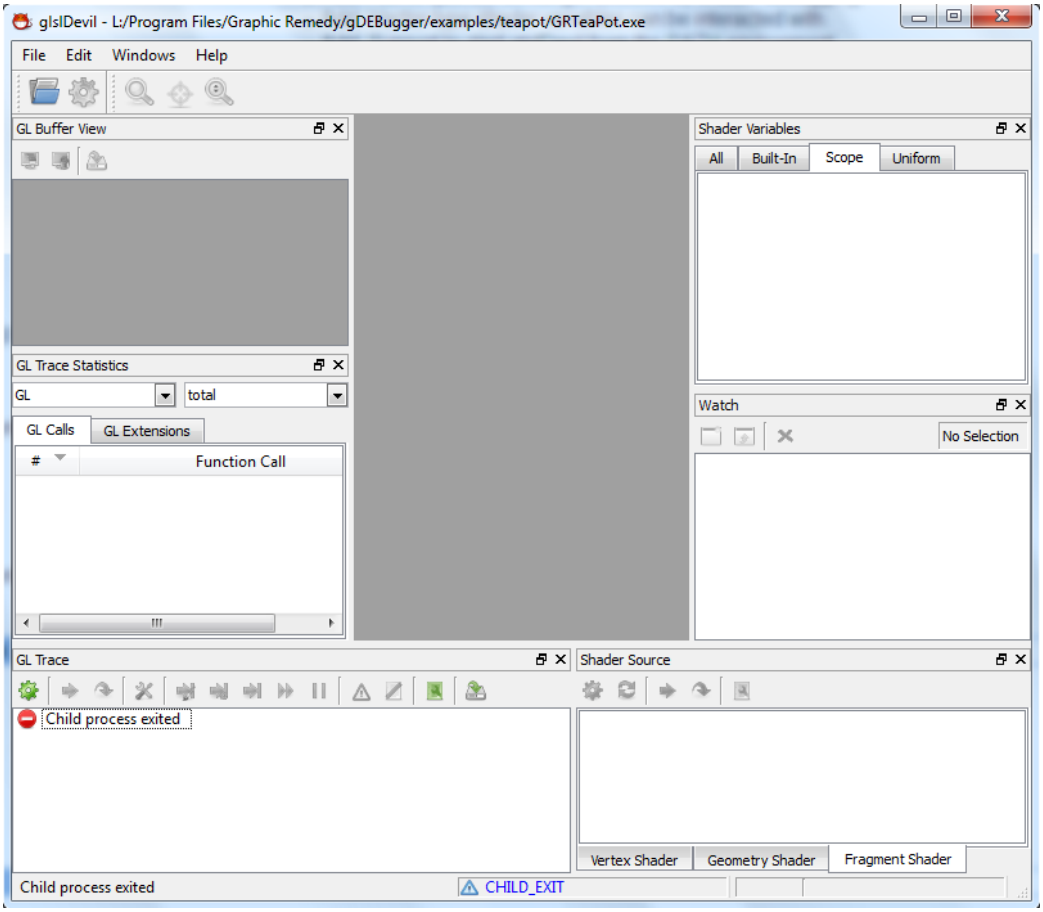
CodeXL (verze 1.3) [15]

Jako nástupce gDEBuggeru je CodeXL taktéž pokročilý nástroj pro ladění programů využívajících OpenGL nebo OpenCL. Stejně jako jeho předchůdce má mnoho nástrojů pro ladění, profilování a analýzu aplikací z pohledu na GPU, CPU a nově i APU. Nabízí rovněž statickou analýzu jádra OpenCL. Testovaný program vyšel 11. 11. 2013.

Tento program nabízí opět tři módy (ladění, profilování a analýza paměti). Na druhou stranu ale stále nedokáže měnit program shaderů. Vývoj tohoto programu se ubírá mnohem více směrem ladění a analýzy již zkompilovaných programů. První zkoumaná verze má výhodnou vlastnost, která nám umožňuje měnit program shaderů za běhu programu. Tato vlastnost se ale pravděpodobně autorům nejeví jako důležitá pro hlavní účel aplikace – profilování a zjišťování výkonnosti a možných chyb v programu. Proto následující verze programu již tuto vlastnost neimplementují.

4.4 GlslDevil (verze 1.1.5) [10]

Nástroj pro ladění OpenGL grafického řetězce. Stejně jako gDEBugger dokáže ladit OpenGL programy bez nutnosti opětovné kompilace nebo dokonce



Obrázek 4.4: Rozhraní nástroje GlslDevil

bez zdrojového kódu (data pro ladění jsou získávána z hardwarové realizace řetězce). Poslední verze (1.1.5) nepodporuje ladění aplikací, které jsou vícevláknové. Navíc, nelze ladit 32bitové aplikace za použití 64bitové verze GlslDevil. Poslední verze byla nahrána v roce 16. 2. 2010. Obrázek 4.4 ukazuje rozhraní nástroje.

4.4.1 Rozhraní

- OpenGL trace
  - poskytuje funkcionalitu pro ovládání ladění, zobrazuje proud OpenGL příkazů, umožňuje krokování, přeskakování funkcí a dokonce i úpravu OpenGL funkcí a jejich parametrů

  - umožňuje spustit program a zároveň vypisovat ladící informace, což ale velmi zpomaluje běh programu
  - je možné skočit na další vykreslení, další změnu shaderu nebo volání další funkce (zadané uživatelem)

- shader source – editory pro vertex, fragment a geometry shadery
- GL trace statistics – přehled volaných OpenGL funkcí
- Watch
- Shader Variables
- GL Buffer View

4.4.2 Ladění shaderů

Podporuje vertex shader, fragment shader a geometry shader. Při ladění geometry shaderu se můžeme podívat na detailní informace o každém primitivu, aktuální hodnoty dat a výstupní vrcholy. Lze dokonce vizualizovat data z geometry shaderu. Vizualizace může například používat různé barvy pro generované vertexy dle jejich indexu.

Ladění shaderu se vykonává na úrovni jednoho vykreslení, kde všechny dotyčné elementy jsou krokovány současně. Aby bylo možné ladit shader, je nutné, aby byl program ve specifickém stavu, a tudíž po volání vykreslovací funkce.

4.4.3 Funkcionalita

Tento program je bohužel neudržovaný a měl problémy s laděním shaderů na všech Testovaných stanicích. Velmi často se stávalo, že se program zastavil při spuštění na chybě nazvané„Internal debugger error“. V případě programů, které šly spustit, nebylo nikdy možné provést ladění shaderů. Aplikace vždy skončila při prvním pokusu o jejich krokování. Vzhledem k tomu, že se nedařilo naleznout řešení pro chyby, které nastávaly, a vzhledem k tomu, že je projekt neudržovaný, nemohlo být Testování programu dokončeno.

4.5 KickJs GLSL Shader Editor [11]

Obrázek 4.5: Rozhraní webové stránky KickJs

KickJs GLSL Shader Editor je aplikace, která se dá spustit ve webovém prohlížeči. Ukázka rozhraní prohlížeče je na obrázku 4.5. Podle dokumentace jsou podporovány (toho času) nejnovější prohlížeče Mozilla Firefox a Google Chrome. Internet Explorer verze 10 nepodporuje KickJS. Aplikace fungovala i v prohlížečích Firefox verze 23.0, Chrome verze 30 a Opera verze 17.0. Poslední aktualizace před Testováním aplikace proběhla 21. 7. 2013.

4.5.1 Funkce a charakteristiky software

- testování shaderů (fragment shader, vertex shader)
- nastavování uniform proměnných společných pro oba shadery
- v animační smyčce lze měnit pouze
  - textury (pouze obrázky přístupné ze stránky www.kickjs.org ve složce /example/shader_editor/)
  - globální nastavení pipeliny (face curling, z-TESt atd.)
  - ortogonální nebo perspektivní kamera
  - výběr jednoho z přednastavených modelů

– pohyb modelu (model bude statický nebo bude rotovat)

4.5.2 Rozhraní

Internetová aplikace rozděluje okno na několik pomyslných částí. V horní části okna jsou volby, které se u většiny aplikací nachází na liště pod volbou soubor (file). Bohužel, akce načíst a uložit ukládají pouze stav aplikace (lze je provést pouze po přihlášení na google účet). Výsledek se nedá nijak exportovat. Uložené záchytné body se mohou pouze použít pro další Testování chodu v aplikaci KickJS.

Výsledná scéna je vidět v malém okně vlevo nahoře. Je možno zapnout režim na celou obrazovku.

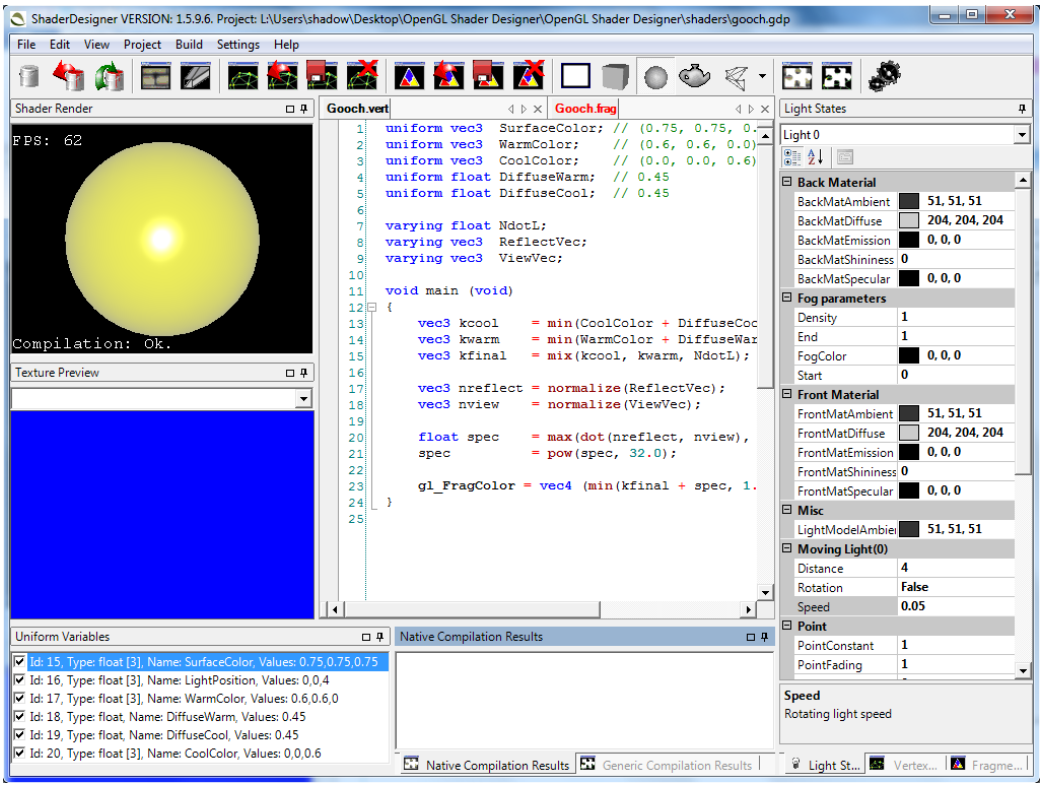
4.5.3 Editor

Jsou zvýrazňována vyhrazená slova, funkce a zabudované OpenGL proměnné. Našeptávání kódu není. Syntaktické chyby nejsou podtrhávány. Chyby jsou ohlašovány ve speciální chybové konzoli, která funguje jako log událostí. V popisu chyby je možné získat číslo řádku. Editoru kódu má číslování řádek. Výsledky kompilace nemají žádné zvýraznění u chyb ani interaktivní odkazy, přesto vypadá výpis velmi přehledně. Není zde možné ukládat změny v editoru jinak než automaticky. Editor okamžitě přeloží shader pokaždé změně. Pokud uživatel napíše několik znaků za sebou, změny se projeví až po tom, co uživatel udělá přestávku v psaní.

4.5.4 Ladění shaderů

Tato aplikace umožňuje pouze napsání vlastního shaderu pro vykreslování jednoho objektu (z několika přednastavených). Tento objekt může mít pouze jednu z několika přednastavených textur. Vykreslovací smyčka je pevně dána. Kód shaderů se dá libovolně editovat. Jinak zde nejsou žádné pokročilejší možnosti krokování programu.

4.6 OpenGL Shader Designer (Verze 1.5.9.6) [12]



Obrázek 4.6: Rozhraní nástroje ShaderDesigner

OpenGL Shader Designer je vývojové prostředí pro vytváření shaderů v OpenGL. Je zde možné vytvářet vertex a fragment shadery. Poslední dokumentace byla vytvořena k datu 12. 6. 2004. Obrázek 4.6 ukazuje grafické rozhraní nástroje.

4.6.1 Funkce a charakteristiky software

- testování shaderů (fragment shader, vertex shader)
- v animační syčce lze pouze
  - měnit textury (lze načítat vlastní)
  - měnit modely (lze načítat vlastní ve formátu gsd)

- model se pohybuje tažením myši
- lze exportovat jako AVI video

4.6.2 Rozhraní

- zobrazení scény
- náhled jedné z textur
- editor pro vertex shader nebo fragment shader
- přehled uniform proměnných
- výsledky kompilace
- nastavení světla

4.6.3 Editor

Jsou zvýrazňována vyhrazená slova, funkce a zabudované OpenGL proměnné. Dále program dokáže najít druhý pár znaků víceřádkového komentáře, pokud se kurzor umístí před znaky * nebo /.

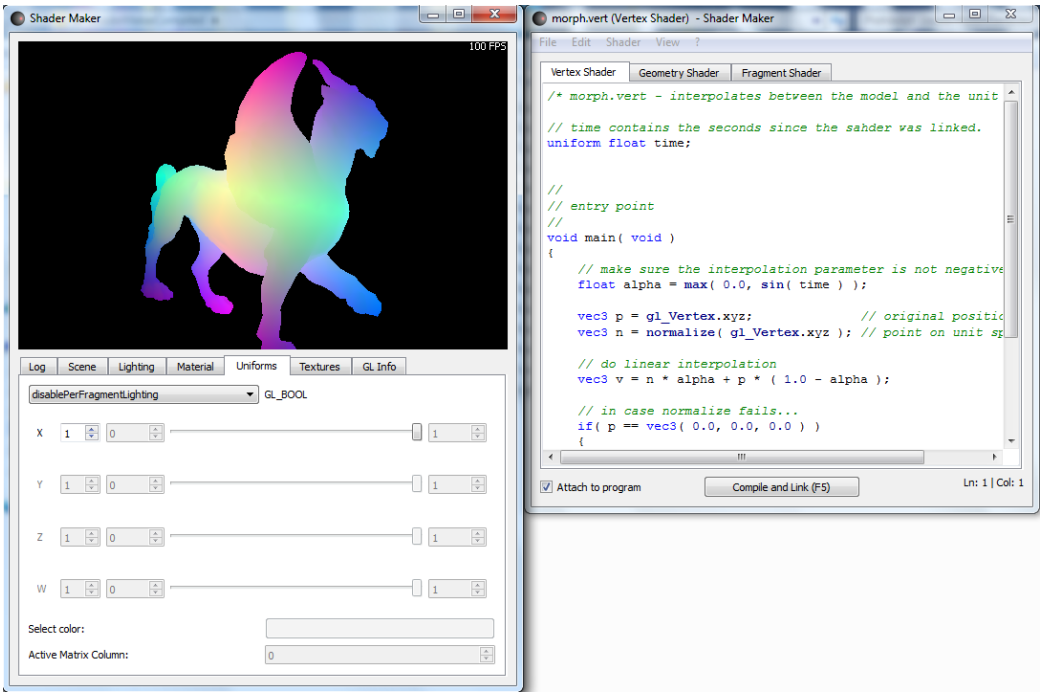
Program umí našeptávat symboly s výjimkou uživatelem definovaných proměnných. Chybou při našeptávání je to, že pokud se našeptávání vyvolá (Ctrl+Space), pokračuje se v psaní a uživatel se rozhodne, že chce napsat proměnnou, kterou našeptávač nezná, pak se bohužel automaticky doplní nějaký symbol z našeptávače, který má k napsanému řetězci nejblíže. Syntaktické chyby nejsou podtrhávány. Chyby jsou ohlašovány ve výsledcích kompilace. Bohužel nejsou více specifické v popisu chyby, než že vypíšou hlášení o nezdařené kompilaci fragment shaderu či vertex shaderu. Editoru má číslované řádky, ale bez odkazu na řádku s chybou je tato vlastnost mnohem méně užitečná. Editor dokáže zkolabovat nebo rozvinout funkční bloky. Změny v shaderu je nutné uložit a zkompilovat, než se změny uplatní. Scéna se pak automaticky překreslí.

4.6.4 Ladění shaderů

Tato aplikace umožňuje pouze napsání vlastního shaderu pro vykreslování jednoho objektu, který lze otáčet v okně výstupu. Vykreslovací smyčka je pevně dána. Kód shaderů se dá libovolně editovat. Nejsou zde žádné možnosti krokování vykreslovací smyčky či shaderů.

Záznamy pro vertex a fragment shader mají napojení na komponentu `PropertyGrid`. Můžeme nastavovat pouze stav OpenGL, který se váže na danou fázi, například test hloubky nebo test průhlednosti. Nastavování uniform proměnných není umožněno.

4.7 Shader Maker [13]



Obrázek 4.7: Rozhraní nástroje ShaderMaker

Shader Maker je multiplatformní editor shaderů vyvinutý v QT. Projekt je opensource. Program neumožňuje ukládání projektu. Umožňuje však rychlé vytváření prototypů shaderů, s čímž mu pomáhá jeho minimalistický

vzhled. Poslední update projektu proběhl 14. 3. 2013. N obrázku 4.7 je vidět rozhraní nástroje.

4.7.1 Funkce a charakteristiky software

- testování shaderů (fragment shader, vertex shader, geometry shader)
- v animační smyčce lze měnit pouze
  - textury (lze načítat vlastní)
  - modely (lze načítat vlastní ve formátu obj)
  - osvětlení
  - materiály
  - projekční matice
  - view port
- model se pohybuje tažením myši

4.7.2 Rozhraní

Nástroj umožňuje prohlídnutí scény v hlavním okně. Má záložky pro prohlížení informací o logu událostí, osvětlení, materiál, uniform proměnné a textury. Poslední záložka obsahuje informace získané z daného hardware a informace o nainstalovaném OpenGL.

V tabulce uniform proměnných se mohou nastavovat uniformy, které jsou obsaženy ve zkompilovaném programu (s výjimkou samplerů). Každá proměnná má v grafickém rozhraní možnost přistupovat až ke čtyřem jejím položkám. V případě matic lze přistupovat i k dalším sloupcům.

4.7.3 Editor

Jsou zvýrazňována vyhrazená slova, funkce a zabudované OpenGL proměnné. Nejsou zvýrazňovány uživatelské proměnné. Editor neumí našeptávat symboly.

Program je nutné překompilovat, aby se změny projevily. Pak jsou změny znatelné při běhu programu. Shader se dá opětovně nalinkovat (restartují se všechny proměnné).

Syntaktické chyby jsou popsány v logu událostí. Obsahují odkaz na řádku v editoru. Editor sice nemá číslování řádek, ale ukazuje číslo řádky, na které je kurzor. Není příliš vhodné, že okno událostí ukazuje jen poslední kompilaci a nijak nezvýrazňuje chybová hlášení nebo varování.

Všechny tři shadery (vertex, fragment, geometry) se dají uložit pouze zvlášť. Není možno nějakým způsobem ukládat projekt.

4.7.4 Ladění shaderů

Stejně jako předchozí aplikace Shader Maker umožňuje pouze napsání vlastního shaderu pro vykreslování jednoho objektu. Tento lze otáčet v okně výstupu. Vykreslovací smyčka je pevně dána. Kód shaderů se dá libovolně editovat za běhu programu. Aplikace neumožňuje krokování smyčky nebo shaderů.

4.8 Shrnutí funkcionality testovaných nástrojů

Testované programy většinou bylo možné použít podle jejich dokumentace. Výjimka však nastala u neudržovaného projektu GlslDevil, který nebylo možné použít podle instrukcí. Neměl pečlivě zdokumentovaná chybová hlášení a online podpora k daným problémům byla také nedostatečná.

Programy, které byly používány zejména na ladění, nebyly skutečně příliš vhodné pro vývoj GLSL aplikací. Tyto programy se hodí na ladění výsledné aplikace. Jen první Testovaná verze gDEBuggeru byla schopná upravovat GLSL kód za běhu aplikace. Ostatní verze byly speciálně vytvořeny hlavně na profilování, ladění a statistiky. Tyto funkce však umožňovaly provádět vývoj přímo v daném prostředí.

Většina ostatních programů byla jednoduše použitelné pro vývoj shaderů. Tyto programy však měly několik nedostatků. Nebylo možné upravovat zobrazovací smyčku. V jediném programu, ve kterém toto bylo možné, se kód

zobrazovací smyčky psal pouze v Pythonu. Tento program navíc neměl vůbec žádné pokročilejší vlastnosti editoru kódu. V aplikacích, které umožňovaly doplňování kódu, bylo toto omezeno pouze na pár klíčových slov či funkcí.

Nejkomplexnějším programem, který by mohl splňovat nejvíce bodů zadání této práce, byl AMD RenderMonkey. Tento program znovu ale neobsahoval žádné více upravitelné možnosti pro úpravu vykreslovací smyčky. Z toho plynulo mnoho problémů, zejména obtížnější práce s více modely (každý musel být přidán ve zvláštním průchodu). Navíc byl vývoj RenderMonkey zastaven a tento nástroj již nepodporuje vyšší verze GLSL než 2.0.

5 Analýza vlastností vyvíjené aplikace

Celá aplikace bude naprogramována v jazyku C# z důvodu rychlosti vývoje a jednoduchosti vytváření grafického rozhraní pro aplikaci.

V projektu se budou používat volání OpenGL. OpenGL je dynamicky linkovaná knihovna. Nejvýhodnější bylo použití knihovny OpenTK pro externí volání funkcí OpenGL v jazyce C#. Tato knihovna je příjemnější na použití než například knihovna Tao. OpenTK má rozdělené výčty podle typů ke kterým patří, má silné typování proměnných, zavedené struktury pro nejčastěji potřebné datové typy (vektory, matice), má implementované násobení vektorů a matic přetížením operátoru, může používat generické datové typy a je více udržované.

Funkcionalita aplikace bude z pohledu většiny vlastností motivována Testovaným programem Render Monkey. Uživatel musí být schopen vytvořit svůj vlastní jednoduchý efekt načtením modelu, textury a specifikací několika shaderů. Návrh musí umožnit, aby bylo možné specifikovat v jednom projektu více programů i více modelů.

Vytvářený GLSL editor musí mít následující vlastnosti:

- pomocí skriptu v jazyce C# může sestavit scénu v interaktivním editoru
- poskytuje možnost upravovat shadery
- jednoduchý způsob pro načítání objektů a textur
- zvýrazňování syntaxe programového kódu
- jednoduchá úprava uniform proměnných
- prohlížení načtených dat
- sada ukázkových úloh

Většina Testovaných nástrojů pro vývoj splňovala velkou část vyčtených bodů. Poměrně často ale však chyběla možnost definovat kód vykreslovací

smyčky. Kód v jazyce C# je dostatečně přívětivý pro vytváření prototypů jednoduchých shaderů díky správě paměti a rychlému překladu.

Je nutné vyřešit, jakým způsobem budou implementovány jednotlivé funkce. Aplikace, která je schopná splnit všechny předcházející požadavky musí splnit následující úkoly:

- použití a nastavení grafických komponent prozvýrazňování a doplňování zdrojového kódu
- kompilace C# kódu za běhu editoru a změna hodnot globálních proměnných za běhu vizualizačního okna
- spuštění programu napsaného za běhu editoru a komunikace s ním
- správa (shaderů, programů, modelů a textur)

V dalších kapitolách budou uvedeny podrobnosti řešení nastíněných problémů. Jazyk C# zjednodušuje řešení některých problémů, ale i tak je někdy nutné provést sofistikovanější nastavení existujících komponent nebo implementovat vlastní.

6 Analýza komponent pro zvýrazňování kódu

Jednou z hlavních vlastností implementovaného editoru je zvýrazňování textu a doplňování kódu. Existuje několik komponent použitelných v C#, které již tuto funkcionalitu implementují. v následujících kapitolách budou analyzovány vlastnosti komponent.

6.1 Scintilla [16]

Scintilla je free source komponenta použitelná pro editování kódu. Lze použít pro jakýkoliv komerční či nekomerční projekt.

Má vlastnosti potřebné pro standardní editory textu. Scintilla má vlastnosti, které jsou užitečné pro editování kódu:

- různé styly syntaxe (proporcionální/neproporcionální fonty, tučné písmo, kurzíva, barva popředí, pozadí, změna fontů)
- indikátor chyb
- doplňování kódu

Vývoj Scintilly začal jako snaha vylepšit text editor PythonWin. PythonWin používal Richedit grafickou komponentu, která změny stylu klasifikovala do stejné kategorie, do které se ukládaly změny zásobníku pro volbu *zpět*. Navíc tato komponenta nastavovala flag *dirty*. Jeden z programů, který byl vytvořen pomocí této komponenty je SciTE. Původně byl SciTE pouze vytvořen pro demonstraci Scintilly. Nejlépe se hodí pro menší projekty, do kterých se dají zahrnout Testovací programy.

Scintilla je psaná v C++. Její kód však používá pouze podmnožinu vlastností jazyka C++, protože Scintilla může být používána i v projektech psaných v C.

6.1.1 ScintillaNET [17]

ScintillaNET je wrapper Scintilly pro platformu .NET. Dle autorů jsou zde k nalezení funkce, které v jiných wrapperech neexistují:

- doplňování snippletů (doplňování často používané malé části kódu, například try/catch nebo property)
- integrované hledání a náhrada řetězců
- hledání regulárních řetězců
- dopředná či zpětná navigace v textu

Bylo již plně konfigurováno několik jazyků pro zvýrazňování syntaxe. Mezi jazyky s hotovou podporou zvýrazňování syntaxe je C#, GLSL mezi nimi ale není. Poslední update byl zaznamenán 18. 3. 2013.

6.1.2 Testovací aplikace

Projekt nemá k dispozici žádnou Testovací aplikaci. Proto byla naprogramována vlastní aplikace pomocí návodu k ověření funkcionality této komponenty.

Postup vytvoření programu:

- stáhnutí knihoven ScintillaNET.dll, SciLexer.dll, a SciLexer64.dll
- konfigurace proměnné Path, aby Windows mohl nalézt knihovny
- přidání komponenty do návrháře Windows formulářů ve Visual Studiu.

Dále je na stránkách ScintillaNET popsáno jak začít programovat aplikace s touto komponentou. Jsou vysvětleny příklady pro:

- vlastní konfigurační soubory
- zprovoznění zvýrazňování syntaxe

- zobrazení čísel řádek
- implementace vlastního lexeru (objektu, který provádí lexikální analýzu)

6.2 Fast ColoredTextbox[20]

Fast ColoredTextbox je komponenta pro editování textu. Je nejbohatší co se týká příkladů (poslední update proběhl 1. 11. 2013). Komponenta byla vytvořena jako lepší verze komponenty RichTextBox, který zvýrazňoval větší množství fragmentů (nad 200) jen velice pomalu. Celé vykreslování textu v nově vytvořené komponentě je naprogramováno pomocí GDI+, což je C/C++ API. GDI+ umožňuje aplikacím používat grafické formátování textu pro displeje i pro tiskárny [19].

K textu lze přistupovat použitím regulárních výrazů. Mimo jiné Je podporováno zalomení řádek, najdi/nahraď, či zpět/vpřed.

6.2.1 Testovací aplikace

Tato komponenta přichází s Testovacím příkladem. Testovací příklad je tvořen jedním hlavním formulářem aplikace, ze kterého se dají vyvolat různé verze editorů pro různé účely. Aplikace obsahuje příklady pro:

- zvýrazňování syntaxe (včetně zvýrazňování všech výskytů té proměnné v kódu, na které se nachází kurzor)
- nastavení vlastní syntaxe
- vytvoření zvýrazňovače řádky
- vytvoření vlastního stylu zvýrazňování textu
- zvýrazňování syntaxe pro dlouhé texty (statisíce řádek)
- práce s extrémně dlouhými texty (miliony řádek, stále zabírá pouze 120 MB v paměti)

- C# editor obsahující základní funkce (vpřed, zpět, ukládání, tisk, práce se schránkou, vyhledávání řetězce, zobrazování bílých znaků)
- vkládání obrázků do textového pole
- podpora logování
- split-screen
- načítání velkých souborů technikou líného načítání (anglicky lazyloading)
- ukázka jak formátovat kód obsahující řetězce z více jazyků (HTML a PHP)
- hypertextový odkaz
- animovaný text
- kolapse a rozvíjení funkčních bloků textu (anglicky codefolding)
- automatické doplňování kódu (doplňování staticky zadaných řetězců, snippletů dynamické doplňování)
- tooltip
- dynamické zvýrazňování kódu (zvýrazňování závorek, zvýrazňování proměnných)
- zvýrazňování syntaxe popsané XML souborem
- zobrazování cizích znaků (Arabských, Čínských, Japonských apod.)
- automatické odsazování
- správa záložek
- emulátor konzole
- nápovědy
- bloky jen pro čtení (nelze upravit strukturu tagů, pouze jejich obsah)
- text s předdefinovaným stylem a hypertextovými odkazy
- vytváření maker (nahraje se posloupnost stisknutých kláves, a takto vytvořené makro se dá použít pro opakovanou úpravu textu)

- otevřené fonty
- pravítko
- mapa dokumentu (náhled dokumentu)

6.3 ICSharpCode.TextEditor [18]

SharpDevelop má grafickou komponentu, která plní funkci textového editoru. Dá se použít na úpravu, ukládání či náhradu řetězců. Dokáže provádět akce s pomocí systémové schránky, umí zvýrazňovat řetězce, rozvíjet či kolabovat bloky textu, používat záložky a měnit nastavení zobrazení. Existuje jen jedna verze publikovaná 13. 11. 2008. Vyžaduje C# 3.0 kompilátor.

Celý projekt byl v poslední době předělán do WPF (Windows Presentation Foundation). Nyní se nazývá AvalonEdit.

6.3.1 Komponenty a třídy

Textový editor obsahuje propojené grafické komponenty `TextEditorControl`, `TextAreaControl` a `TextArea`. `TextEditorControl` je na nejvyšší úrovni. `TextAreaControl` zapouzdřuje komponentu `TextArea` a její scrollbary. `TextArea` zpracovává vstup z klávesnice a vykresluje text.

6.3.2 Vlastnosti editoru

Editor má implementovanou volbu zpět a opětovné vrácení změn. Dokáže dočasně vyznačit slova pomocí markerů. Implementuje záložky, skládání bloků kódu, automatické zarovnání kódu a zvýraznění syntaxe.

6.3.3 Testovací aplikace

Aplikace může otevřít několik záložek, ve kterých budou různé soubory se zdrojovým kódem. Umožňuje načítání a ukládání. V úpravách jsou volby

typické pro úpravy v aplikacích (práce se schránkou a hledání či nahrazení řetězce). Navíc je v úpravách možnost vkládat a navigovat mezi záložkami.

Můžeme dále zvolit další možnosti, které se týkají hlavně zobrazování kódu:

- split-screen
- zobrazení bílých znaků
- zobrazení čísel řádek
- zvýraznění aktuální řádky
- zvýraznění párové závorky
- nastavení velikosti tabulátoru
- nastavení velikosti fontu

6.4 ChameleonRichTextBox [21]

Tato komponenta je součástí projektu CleanCode. CleanCode jsou knihovny pro vývojáře .NET, PowerShell, SQL, Java, Perl, a Javascript aplikací.

ChameleanRichTextBox je komponenta, která dědí od RichTextBox. Je dalším vylepšením komponenty SyntaxHilightTextBox. Poslední update proběhl 30. 6. 2013. Mezi její vlastnosti patří:

- změna zvýrazňování syntaxe bez implementace nové komponenty použitím dědičnosti
- doplňování kódu
- výkonnější než SyntaxHilightTextBox
- doplňování klíčových slov
- najdi a nahraď
- podpora maker

- klávesové zkratky pro formátování
- konfigurace XML souborem

6.4.1 Testovací aplikace

Existuje Testovací aplikace [22]. Tato aplikace TEStuje mnoho implementovaných částí projektu CleanCode. Zajímá nás hlavně komponenta ChameleonRichTextBox. Testovanou komponentu bylo možné použít pro zvýrazňování syntaxe pro několik přednastavených jazyků (SqlServer, Oracle, MySql a Odbc). Ukazuje jak pracovat s tabulátory a zvýrazňováním. Bohužel tento příklad neukazuje žádné pokročilejší funkce. Nepodařilo se najít žádné obsáhlejší příklady k této komponentě.

6.5 Zhodnocení komponent

Všechny komponenty dokázaly zvýrazňovat syntaxi a doplňování kódu. Udržovanost těchto komponent se však liší.

- SCIntilla je sice velmi rozšířená, ale její .NET wrapper je udržovaný a dokumentovaný o něco méně kvalitněji. Příkladů použití je velmi málo.
- ICSharpCode.TextEditor splňuje základní požadavky, které budou potřeba v této práci. Tato komponenta má pro Window Form grafické rozhraní nejstarší verzi.
- ChameleonRichTextBox je součástí většího projektu a bylo velmi obtížné nalézt užitečný příklad nebo dokumentaci.
- FastColoredTextbox vychází jako nejlepší volba z daných komponent. Tento projekt je velmi rozsáhlý, udržovaný a má velké množství featur a příkladů, které všechny důležité funkce demonstrují.

7 Implementace editoru zdrojového kódu

Pro editaci zdrojového kódu se bude používat komponenta FastColoredTextbox [20]. Tato komponenta umožňuje zvýrazňování kódu a doplňování syntaxe. Byla vybrána kvůli velkému množství vlastností a příkladů, na kterých jsou vlastnosti demonstrovány.

7.1 Nastavení komponenty FastColoredTextbox

Komponenta je připravená pro zvýrazňování kódu C#. Stačí pouze přepnout její vlastnost `Language` na hodnotu `Language.CSharp`, a veškeré zvýrazňování syntaxe C# funguje. Přesto ale existují drobné vady. Stále se zvýrazňují řetězce uvnitř komentářů. Zdrojový kód od počátku souboru se zvýrazní jako víceřádkový komentář, a to pokud má víceřádkový komentář v kódu přítomen ukončovací token, ale žádný startovací token. Většina editorů v takovém případě podtrhne ukončovací token víceřádkového komentáře. I když se chování liší od většiny jiných editorů, stále upozorňuje uživatele, že je v kódu chyba.

Pro vlastní nastavení zvýrazňování kódu umožňuje komponenta přidat implementaci metody na událost TextChanged. Komponenta nezvýrazňuje kód na základě lexikální ani syntaktické analýzy, ale na základě regulárních výrazů. Po každé změně textu se musí zvýraznění znovu vyhodnotit.

7.2 Zvýrazňování vlastního jazyka

Jazyk GLSL není standardně podporován komponentou FastColoredTextBox. Nejprve musíme vytvořit regulární výrazy, které odpovídají konstrukcím jazyka GLSL a musíme jim nastavit různé styly. Styly se musí aplikovat ve správném pořadí.

Nastavením komponenty na jazyk `Language.Custom` komponenta už dále

```
GreenStyle = new TextStyle(Brushes.Green, null,
    FontStyle.Italic);
...
range.ClearStyle(GreenStyle);
range.SetStyle(GreenStyle, @"//.*$", RegexOptions.Multiline);
```

Tabulka 7.1: Ukázka zrušení a znovu nastavení stylu

```
range.SetStyle(GreenStyle, @"(/\*.*?\*/)|(/\*.*)",
    RegexOptions.Singleline);
range.SetStyle(GreenStyle, @"(/\*.*?\*/)|(.*\*/)",
    RegexOptions.Singleline | RegexOptions.RightToLeft);
```

Tabulka 7.2: Ukázka nastavení zýraznění víceřádkového komentáře

nezvýrazňuje kód na základě přednastavených pravidel.Dále je třeba připojit metodu na událost TextChanged. V napojené metodě musíme zrušit všechny používané styly danou instancí FastColoredTextbox. Stejné styly se znovu nastaví.

Na ukázce 7.1 vidíme zrušení a znovu nastavení stylu. Pro jednořádkové komentáře nastavíme volbu hledání regulárního výrazu víceřádkově. Může se zdát, že bychom měli nastavit volby obráceně a hledat jednořádkově. Regulární výraz ale při nastavení volby víceřádkového prohledávání Testuje každou řádku zvlášť. Znak dolaru máme na konci výrazu, protože hledáme u jednořádkového komentáře konec řádky. U víceřádkového komentáře naopak nastavíme volbu prohledávání jednořádkově. Je to proto, že celý text pak bude brát regulární výraz jako jednu řádku. Nebude se tedy zastavovat na znacích pro odřádkování.

Na ukázce 7.2 vidíme nastavení komentáře na více řádek. Nejprve se specifikuje začátek víceřádkového komentáře a nastaví se prohledávání celého textu jako by byl jednou řádkou. Následně se specifikuje regulární výraz pro konec víceřádkového komentáře a rovněž se hledá v celém textu jako by byl jednou řádkou. Na rozdíl od předchozího případu se hledá zprava doleva.

Pro zvýraznění dalších tokenů bylo třeba nalézt celý výčet všech předdefinovaných funkcí proměnných, klíčových slov a direktiv preprocesoru pro GLSL verzi 4.3. Vzhledem k tomu, že se používají pouze regulární výrazy,

```
@"\b(gl_NumWorkGroups|gl_WorkGroupSize|gl_WorkGroupID|...)\b";
```

Tabulka 7.3: Ukázka nastavení zýraznění všech předdefinovaných proměnných

není v žádném případě zvýrazňování zdrojového kódu dokonalé. Nemůžeme zvýrazňovat proměnné se stejným jménem v jednom bloku, nemůžeme odlišovat uživatelsky definované funkce od uživatelsky definovaných proměnných atd.

Bylo zvoleno, že všechny shadery budou mít stejné zvýrazňování syntaxe. Tento postup byl aplikován kvůli jednoduchosti implementace. Množství předdefinovaných proměnných a funkcí je poměrně velké. Regulární výraz, který hledá klíčová slova, musí obsahovat celou množinu klíčových slov.

Na ukázce 7.3 vidíme, jaký způsobem se vytvoří regulární výraz pro hledání předdefinovaných proměnných. Seznam všech proměnných, funkcí, klíčových slov a preprocesorových direktiv byl získán ze specifikace GLSL jazyka verze 4.3 [31].

7.2.1 Vyvolání nabídky pro doplňování kódu

Doplňování kódu je implementováno stejně jako zvýrazňování syntaxe na principu hledání regulárních výrazů. Třída `AutocompleteMenu` má speciální proměnnou `SearchPattern`, která obsahuje regulární výraz pro výčet množiny znaků, které si bude komponenta ukládat do svojí vnitřní dočasné proměnné `fragment`. V případě, že napíšeme znak, který není ve výčtu, ukládání do fragmentu se restartuje. Když uživatel napíše minimálně dva znaky a přestane na chvíli psát (nebo přikáže vypsání nabídky pro doplňování kódu klávesami SHIFT + mezerník), začne komponenta ze svého seznamu vybírat položky, které nabídne k doplnění.

Přednastavený obsah proměnné `SearchPattern` pro doplňování C# kódu:

```
@"[\w\.:=!<>]"
```

Význam uvedeného řetězce je takový, že se mají do fragmentu ukládat znaky slov (písmena, číslice), tečky, dvojtečky, rovnítka, vykřičníky a zna-

```
AutocompleteMenu popupMenu =
    new AutocompleteMenu(fastColoredTextBox);
DynamicCollectionCs items = new DynamicCollectionCs(
popupMenu, autoCompleteCs, fastColoredTextBox, prjFile);
popupMenu.Items.SetAutocompleteItems(items);
```

Tabulka 7.4: Ukázka vytvoření menu pro doplňování kódu

ménka větší a menší. Speciální znaky jsou zde proto, že kromě proměnných ukazují příklady komponenty FastColoredTexbox použití při doplňování snippletů (rychlých částí kódu, například vypsání anotace pro klausule `if` nebo `switch`) a vkládání mezer mezi operátory a čísla.

Pro účely této práce musíme změnit `SearchPattern` na následující:

```
"[\\w\\.\\[\\]\":=!<>]"
```

Zavináč, který v jazyce C# pomáhá vypnout vlastnost zpětného lomítka jako escape symbolu nemůžeme využít. Je nutné přidat do řetězce uvozovky, které se jinak berou jako konec řetězce. Uvozovky musíme přidat proto, aby uživateli editor pohodlně našeptával řetězcové konstanty, které se budou používat při vybírání objektů ze slovníků.

7.2.2 Nastavení dat pro doplňování kódu

Všechny řetězce, které nabídka doplňuje, uložíme do číselníku a nastavíme třídě `AutocompleteMenu` jako vlastnost `Items`. Tento číselník se může dynamicky měnit. Abychom mohli doplňovat řetězce a reagovat na změny v projektu, vytvoříme vlastní číselník `DynamicCollectionCs`.

Na ukázce 7.4 je vidět jednoduchá posloupnost příkazů, která vytvoří menu doplňující kód pro instanci editoru FastColoredTextBox. Objekt typu `DynamicCollectionCs` poskytuje seznam všech řetězců, které se mohou objevit v nabídce. Objekty `ProjectFile` přistupují k souborům se zdrojovými kódy. Proměnná `prjFile` v příkladu je instance třídy ProjectFile. Objekt `popupMenu` je používán na získání fragmentu a objekt `autoCompleteCs` obsahuje uložené neměnné přednastavené proměnné, metody a klíčová slova.

```
public class DynamicCollectionCs : IEnumerable<AutocompleteItem>
{
...
    public IEnumerator<AutocompleteItem> GetEnumerator()
    {
        ...
        foreach (var modelFile in listOfModels)
        {
            name = StrinGS.DICT_VAR_MODELs +
                "[\"" + modelFile.NameWithoutExtension + "\"]";
            yield return new DictionaryAutocompleteItem(name);
            yield return new DictionaryAutocompleteItem(
                name + StrinGS.CLASS_MODEL_METHOD_DRAW);
        }
        ...
```

Tabulka 7.5: Ukázka vráceného číselníku třídy DynamicCollectionCs

Na ukázce 7.5 je vidět základní myšlenka, jak budeme vracet seznam řetězců, které chceme doplňovat. Platforma .NET nám poskytuje klíčové slovo `yield`. Když nepoužíváme `yield` a máme metodu, která vrací výčet (číselník) hodnot, musíme číselník nejdříve vytvořit, pak naplnit a nakonec vrátit. v případě, že `yield` používáme, můžeme vracet položky číselníku průběžně pomocí `yield return`. Vrácená hodnota se tak nastaví implicitně vytvořeného číselníku.

V uvedeném příkladě vracíme objekt typu `DictionaryAutocompleteItem`, který dědí od třídy `MethodAutoCompleteItem` poskytované knihovnou distribuovanou s komponentou FastColoredTextbox. Úkolem objektu `DictionaryAutocompleteItem` je uložit řetězec nabízený k doplnění kódu a zároveň porovnání s aktuálním fragmentem, který vznikl při psaní kódu.

Při vybírání položek, které budou nakonec zobrazeny v seznamu s nabízenými řetězci, se porovná již napsaný fragment s první částí řetězce v seznamu. Pokud dosud napsaný fragment může pokračovat tak, že složí výsledný řetězec, tak se tento řetězec v seznamu ponechá. Řetězce se porovnávají standardními metodami C# pro práci s řetězci. Pro zjištění, zda řetězce začínají stejně, se používá metoda StartsWith. Nejsou poskytovány žádné nástroje pro optimalizaci.

8 Online kompilace C# kódu

Hlavní funkce, bez které se aplikace neobejde, je možnost kompilovat C# kód za běhu editoru. Grafická komponenta pro zvýrazňování a doplňování kódu pomáhá uživateli s vývojem kódu. Bez možnosti zkompilovat a spustit kód je však nástroj nefunkční.

Původní představa byla taková, že skoro veškerý kompilovaný kód bude mít uživatel přístupný k editaci. Tento kód by se objevil ve formě několika metod (například metody pro aktualizaci, metody pro vykreslení a podobně). Výsledek by se zkompiloval do dll knihovny. Tato knihovna by se načetla, vytvořila by se její instance v editoru, a tato instance by se použila k běhu vizualizace.

Problém zamýšleného přístupu je ten, že upravovat kód již běžící aplikace není triviální problém. Pro docílení tohoto efektu musíme být schopni zkompilovaný kód za běhu načíst a spustit. Protože se ale bude kód mnohokrát předělávat během vývoje, je nutné předchozí verze kódu odstranit z paměti.

Existuje jen jedna možnost, jak odstranit z paměti assembly (zkompilovaný C# kód ve spustitelné formě), které bylo jednou připojeno k běžící aplikaci. Je nutné načíst assembly do zvláštní aplikační domény, které poskytuje určitou izolaci od ostatního kódu. Kód v další vytvořené aplikační doméně se může spustit nezávisle na kódu v hlavní aplikační doméně. Když chceme odstranit starou přeloženou verzi programu z paměti, musíme běžící přeložený program ukončit a pak zrušit aplikační doménu.

Je nutné řešit dva problémy. Jak vypadá kód, který chceme kompilovat a jakým způsobem bude probíhat komunikace mezi hlavní programem (editorem) a vizualizačním programem (vizualizačním oknem).

8.1 Návrh způsobu kompilace

Pro kompilaci kódu používáme třídu `Microsot.CSharp.CSharpCodeProvider`. Tato třída nám umožňuje přístup k překladači. Vytvoříme její instanci, dodáme zdrojové soubory a parametry překladu pro provedení kompilace. Výsledkem kompilace může být spustitelný soubor nebo dynamicky linkovaná knihovna.

```
public class GlobalVarSetCustom : GlobalVarSet
{
    [DisplayName("Blue Background Color")]
    public InterfaceVector3 vColor { get; set; }
    [DisplayName("Matrix Projection")]
    public InterfaceMatrix4 mProjection { get; set; }
    [DisplayName("Matrix View")]
    public InterfaceMatrix4 mView { get; set; }
    [DisplayName("Matrix Model")]
    public InterfaceMatrix4 mModel { get; set; }

    public Init()
    {
        this.vColor.Val = new Vector3(0.5f, 0.5f, 0.0f);
    }
}
```

Tabulka 8.1: Ukázka implementace globálních proměnných

Výsledná aplikace by neměla fungovat tak, že uživatel bude psát veškerý kód vizualizačního okna. Většina kódu bude již napsaná předem. Uživatel bude specifikovat pouze kód dvou tříd. Jedna z těchto tříd implementuje souhrn globálních proměnných, druhá popisuje inicializaci, aktualizaci a vykreslování vizualizace.

8.1.1 Implementace globálních proměnných

Třída GlobalVarSetCustom obsahuje globální proměnné, jejichž hodnoty bude uživatel chtít za běhu měnit bez toho, aby musel měnit kód (měnit C# implementaci za běhu ve výsledné aplikaci nelze). Protože chceme nastavovat několik typů globálních proměnných, je nutné poskytnout uživateli nějaké aplikační rozhraní, které se postará o přístup k těmto proměnným. Kompilátor očekává, že tato třída bude mít proměnné dostupné přes vlastnosti a konstruktor, který všechny proměnné inicializuje.

Na ukázce 8.1 je vidět implementace třídy `GlobalVarSetCustom`. Třída, kterou bude uživatel takto implementovat, musí dědit od třídy `GlobalVarSet`.

Rodičovská třída má automatický konstruktor, který vytvoří instance objektů pro všechny uživatelem deklarované vlastnosti. Pro inicializaci objektů, které ani nemá rodičovská třída v sobě definované se používá `System.Reflection`.

Instance třídy `GlobalVarSetCustom` se nastaví jako zdroj komponenty PropertyGrid, která se pak používá pro nastavování hodnot proměnných za běhu programu.

Existuje několik tříd, které jsou připraveny pro použití s komponentou PropertyGrid. Tyto třídy v sobě obsahují data v podobě OpenTK struktur. Jména těchto tříd skládají ze slova Interface a datového typu, který třída obaluje. Každá třída pak má jeden veřejný atribut Val. Název třídy napovídá o tom, jakého je atribut datového typu. Existuje několik tříd, které obalují primitivní datové typy nebo struktury OpenTK. Jsou to třídy pro obalení čísel float, double, integer a pro obalení vektorů a matic.

Většina z těchto tříd má tři konstruktory. Jeden prázdný, který inicializuje strukturu nebo jednoduchý datový typ na nulové hodnoty. Další konstruktor přijímá čárkami oddělené hodnoty čísel, která jsou použita na inicializaci struktur. Matice a vektory přijímají jako datové typy pouze čísla float. Matice a vektory mají ještě třetí konstruktor, do kterého můžeme vložit rovnou celou OpenTK strukturu, dle typu třídy.

8.1.2 Implementace vizualizační smyčky

Třída `VisualizationModulCustom` dědí od třídy `VisualizationModul`. v rodičovské třídě je definováno mnoho chráněných atributů, ke kterým má uživatel při implementaci přístup:

- instance třídy s globálními proměnnými `globalVarSet`, popsaná výše
- slovníky (`System.Collections.Generic.Dictionary`)
  - `shaderPrograms` – hotové programy obsahující zkompilované a připravené shadery
  - `models` – modely, které bude aplikace vykreslovat
  - `textures` – textury, které jsou již nahrané na grafickou kartu
- časy pro animační smyčku

  - `elepsedMilliseconds` – celkový čas od startu vizualizace v milisekundách
  - `elapsedSeconds` – celkový čas od startu vizualizace v sekundách
  - `elapsedMinuTES` – celkový čas od startu vizualizace v minutách
- `azimuth` – azimut vypočtený z tažení myši se stisknutým levým tlačítkem
- `zenith` – zenith vypočtený z tažení myši se stisknutým levým tlačítkem
- `screenWidth` – šířka zobrazovací plochy komponenty `glControl`
- `screenHeight` – výška zobrazovací plochy komponenty `glContorl`
- `near` – ořezávací rovina near nastavovaná ze speciální komponenty na vizualizačním okně
- `far` – ořezávací rovina far nastavovaná ze speciální komponenty na vizualizačním okně
- `distance` – vzdálenost objektů od kamery nastavovaná ze speciální komponenty na vizualizačním okně
- `size` – velikost objektů nastavovaná kolečkem myši
- `loopUpdateAllowed` – povolení aktualizace metody LeepUpdatenastavováno zaškrtávacím políčkem na vizualizačním okně

Slovníky jsou naplněny připravenými objekty. K těmto objektům můžeme přistoupit pomocí řetězcových klíčů. Každý objekt je ve slovníku uložen pod stejným jménem, pod jakým jménem je jeho zdroj uveden v editoru. U textur a modelů se jedná o názvy souborů. U programů se jedná o názvy programů uložených v nastavení projektu.

Mnoho proměnných se nastavuje automaticky při aktualizaci, která nastane při manipulaci s uživatelským rozhraním. Proměnné jsou pojmenovány podle jejich zamýšleného užití. Nikdo však uživatele nedonutí používat například proměnné pro azimut a zenit. Tyto atributy byly pouze vybrány jako nejčastější potřebné při jakékoliv vizualizaci. Některé z proměnných je příhodnější nastavovat speciálním způsobem, místo nastavování všech v komponentě `PropertyGrid`. Například otáčení modelu tažením myši je mnohem příjemnější než psaní číselných hodnot do různých polí transformačních matic zobrazených v komponentě `PropertyGrid`.

Většina ze zmíněných atributů je nastavována pouze při uživatelském vstupu. Aktualizace proběhne, pokud uživatel zadá na klávesnici číselné hodnoty do připravených komponent, pokud táhne myší se stisknutým levým tlačítkem nad komponentou pro zobrazení vizualizace, pokud roztáhne okno, pokud klikne na šipku u komponent `NumericUpDown` nebo pokud roluje kolečkem myši. Uplynulý čas aplikace je jediná informace, která se aktualizuje zvlášť každý snímek.

Uživatel může přetížit následující metody:

- `Init` – nastavení jakýchkoliv proměnných při inicializaci; aktualizace slovníků a globálních proměnných probíhá automaticky před zavoláním této metody
- `InputUpdate` –aktualizace proměnných ze vstupů; proběhne až po aktualizaci chráněných atributů třídy z uživatelského rozhraní
- `LoopUpdate` – implementace aktualizace proměnných při animaci závislé na čase; proběhne každý snímek po tom, co se aktualizují chráněné proměnné pro čas; z uživatelského rozhraní (nebo i programově) lze volání této metody zakázat
- `Render` – implementace vykreslení na plochu komponenty glControl; probíhá každý snímek, nelze zakázat

Pokud v kterékoliv z těchto metod vznikne výjimka, je odchycena ve fixním kódu, který vyjmenované metody volá. Důvodem popsaného chování je, že uživatel by mohl napsat kód, který skončí výjimkou v každém snímku. Kdyby se vykreslování nezastavilo, zaplavil by se log výpisy výjimek. Jediná možnost jak v případě výjimky pokračovat je restartovat vizualizační okno nebo nahrát znovu zdroje. Pokud je ale výjimka způsobena něčím jiným než chybějícími zdroji, nastane pravděpodobně znovu.

8.1.3 Změna hodnot uniform proměnných za běhu vizualizace

Protože změnou nastavení globálních hodnot měníme zdrojový kód vizualizačního okna, nemůžeme na seznam globálních hodnot napojit žádnou komponentu v editoru. Místo toho se hodnoty globálních proměnných mění

v komponentě, která se nachází na zkompilovaném a spuštěném vizualizačním okně.

Pro nastavování uniform proměnných musí uživatel napsat kód v C# použitím metod OpenTK. Pokud nechce používat animace, bude tento kód psát pouze do metody `InputUpdate`. Pokud chce aktualizaci v závislosti na čase, bude muset tento kód napsat do metody `LoopUpdate`. Pokud chce uživatel kód napsat tak, aby mohl hodnoty uniform proměnných měnit za běhu vizualizace, musí získávat data pro nastavení uniform proměnných z objektu `globalVarSet`. Když se uživatel rozhodne, že se tyto proměnné zobrazované v komponentě `PropertyGrid` budou aktualizovat na základě chráněných proměnných třídy `VisualizationModul` v každém snímku, pak jsou tyto proměnné prakticky zobrazeny pouze pro čtení. Pokud uživatel upraví jejich hodnoty, tak se v dalším snímku aktualizují na základě kódu, který napsal sám uživatel. Uživatel si tedy musí dávat pozor na to, kdy a jakým způsobem se mění hodnoty globálních proměnných.

Navíc zde existuje možnost vypnout volání metody `LoopUpdate`, aby mohl uživatel měnit za běhu i proměnné, které se sami ve smyčce automaticky aktualizují na základě jiných hodnot.

9 Komunikace s vizualizačním oknem

Uživatel bude během jednoho běhu nástroje editoru GLSL potřebovat několikrát zdrojový kód upravit, několikrát jej zkompilovat a několikrát jej načíst a využít. Platforma .NET sice dokáže dynamicky načítat knihovny v rámci jedné aplikační domény během běhu aplikace, nedokáže však tyto načtené knihovny během běhu aplikace uvolňovat z paměti stejné aplikační domény. Pokud by uživatel mnohokrát zkompiloval celý projekt i třeba s malými změnami, po každé by vytvořil novou dynamicky linkovanou knihovnu, kterou by načetl do aplikace a její paměťová náročnost by se nekonečně zvyšovala. Navíc takováto knihovna by se uložila na disk a byla by zamčená – chráněná proti přepsání nebo smazání.

Pokud chceme, abychom mohli nějakou dobu vykonávat určitý kód a po libovolné době přestat kód používat a odstranit jej z paměti, musíme použít třídu `System.AppDomain`. Kromě stávající aplikační domény, ve které běží editor, vytvoříme novou aplikační doménu, ve které bude běžet pouze vizualizační okno.

9.1 Třída AppDomain

Jiným způsobem, než užíváním aplikačních domén, není možné uvolňovat z paměti jednotlivé assembly nebo definice typů. Veškerý kód, jehož assembly chceme uvolnit, se musí nacházet uvnitř aplikační domény a zároveň musí běžet uvnitř aplikační domény. Pokud vytvoříme v rámci jedné aplikační domény instanci určitého typu, nemůžeme definici této instance dostat z paměti jiným způsobem než zrušením celé aplikační domény. Pokud vytvoříme novou aplikační doménu, vytvoříme určitou izolaci, která nám odděluje nově vytvořený typ od hlavní aplikace, ale zároveň stále umožňuje určitou úroveň interakce[23].

Někdy může být problémem garbage collector, který uvolní z paměti stále ještě používané objekty při vzdálené komunikaci. Po určité době se uvolní z paměti objekty serverů. Aplikace, která s tímto chováním nepočítá, pak skončí na výjimce. U aplikací se vzdáleným přístupem se počítá s tím, že může být vytvořeno obrovské množství klientů. Zde je tradiční garbage collector nahrazen systémem, který uvolňuje objekty po určitém čase. v na-

šem případě ale počítáme s aplikací, která bude mít pouze jednoho klienta za jakýchkoliv okolností. z toho důvodu můžeme problém vyřešit jednoduchým přetížením metody `GetLifetimeService`. Toto je metoda třídy, která dědí od `MarshalByRefObject`. v našem případě je to formulář. Pokud přetížíme metodu tak, aby místo objektu vrátila `null`, nebude žádný objekt pro vzdálenou komunikaci z paměti vymazán [25].

9.2 Výhody užívání aplikačních domén

Hlavní výhodou je hlavně již dříve zmíněná možnost uvolnění použitých assembly z paměti. Na rozdíl od komunikace mezi procesy poskytuje používání aplikačních domén prostředek ke komunikaci na vyšší úrovni. Data můžeme serializovat a posílat do druhé aplikační domény.

Aplikace běžící v jiné doméně se dá zastavit bez toho, aby toto ovlivnilo celý proces. Chyby v jedné aplikaci nemohou ovlivňovat druhé aplikace. Kód běžící v jedné doméně nemůže ovlivnit jiné domény ve stejném procesu, co se týče chyb v paměti. Pokud uživatelský kód bude způsobovat chyby s pamětí (neuvolněné objekty, přístup na nepovolená místa v paměti díky využívání chybných ukazatelů při používání OpenGL funkcí), hlavní aplikace editoru by měla v pořádku pokračovat. Při problémech stačí pouze restartovat vizualizační okno.

Kód, který běží v jedné doméně, nemůže přímo přistupovat ke kódu ani zdrojům druhé domény. Nelze používat přímá volání mezi objekty. Objekty, které se posílají mezi doménami, jsou kopírovány nebo je k nim přistupováno přes proxy. Pokud je objekt zkopírován, veškerý přístup k němu je pouze lokální. Pokud je k objektu přistupováno přes proxy, používají se vzdálená volání na objekt. V takovém případě je objekt v jiné doméně než ten, kdo ho volá.

Aplikace v hlavní doméně může udělovat oprávnění, s jakými bude kód v druhotné doméně běžet.

9.3 Návrh komunikačního rozhraní

Při používání aplikačních domén můžeme využít dvou způsobů komunikace:

- nastavením dat do aplikační domény funkcí `SetData()` a následným získáním dat v druhé doméně přistoupením k objektu domény a zavoláním metody `GetData()`
- použitím tříd, které implementují `MarshalByRefObject`

První způsob posílání dat nastaví data, ale aplikace běžící v druhé doméně nezíská žádné upozornění, že data byla poslána. Příhodnější je využívání implementace třídy `MarshalByRefObject`, díky čemuž můžeme využívat vzdálená volání na objektech v jiné doméně. Dá se využít toho, že třída Form pro vytváření formulářů již implementuje `MarshalByRefObject`. Zde se bohužel nachází problém.Není vždy bezpečné používat vzdálený přístup na objekty, které mají složité uživatelské rozhraní, jako jsou právě formuláře. Ve výsledné aplikaci je na vizualizačním okně značné množství komponent, což ještě potvrzuje nutnost řešit tento problém.

Nestačí, že je formulář potomek třídy pro komponentu, která dědí od MarshalByRefObject. Není totiž podporován vztah rodič/potomek mezi doménami. Rozhraní, která pracují s formulářem, mohou být úspěšně použita pro vzdálený přístup, ale API takovéto komponenty vzdálený přístup neumožňuje. Pokud si vývojář tuto skutečnost neuvědomí, může při vývoji narazit na výjimky při serializaci [24].

Způsob, jak řešit popsaný problém je vždy použít rozhraní. Formulář (vizualizační okno) bude implementovat rozhraní pro manipulaci s oknem. Při vytváření proxy objektu nebudeme přetypovávat objekt na `Form`, ale na rozhraní, které `Form` implementuje.

Rozhraní pro komunikaci specifikuje následující metody:

- `ShowWindow` – zobraz okno
- `CloseWindow` – zavři okno
- `SetTexture` – nastav texturu
- `SetShader` – nastav shader
- `SetMesh` – nastav model
- `SetShaderProgram` – nastav program
- `InitializationFinished` – oznam, že inicializace byla dokončena

- `SetCloseDelegate` – nastaví delegát pro oznámení o zavření vizualizačního okna
- `SetLogDelegate` – nastaví delegát pro poslání zpráv z vizualizačního okna
- `SetErrorLineDelegate` – nastaví delegát pro posílání chyb vzniklých při kompilaci GLSL kódu (pošle zprávu o chybě, soubor a řádku ve které vznikla)

Metoda na zobrazení okna má vyvolat stejnou událost, jakou by vyvolala stejně pojmenovaná metoda u formuláře. Stejné je to i u zavření okna. Nastavení zdrojů posílá vizualizačnímu oknu data pro následné zobrazení. Data musí být jednoduché datové typy, nebo objekty, které mají nastavený atribut `Serializable`.

Metoda pro oznámení, že všechny zdroje již byly nahrány, je nutná, protože vizualizační okno musí vědět, ve kterou chvíli jsou již všechny zdroje přítomny. Pokud by se pokusilo inicializovat (nebo obnovit) inicializaci dříve, než se nahrají všechny zdroje, nastala by chyba. v takovém případě by se všechny zdroje stejně museli nahrát znovu. Případně by se muselo okno restartovat.

Při prvním spuštění se nejdřív nastaví všechny zdroje, pak se zavolá `InitializationFinished` a až po té se zavolá metoda pro zobrazení formuláře. Pokud se metody `SetTexture`, `SetShader` nebo `SetMesh` zavolají během doby, kdy je okno zobrazeno, nastaví se příznak, že jsou dostupné nové zdroje. Ve chvíli, kdy se zavolá metoda `InitializationFinished` je vykreslování pozastavené do té doby, dokud nejsou všechny zdroje načteny. Jakmile jsou zdroje načteny, zpracovány a přidány do slovníků objektů, vykreslování se opět spustí.

Je nutno poznamenat, že při jednom běhu okna není možné měnit kód vizualizační smyčky, její inicializace ani žádné jiné metody či deklarace v C# jazyku. z tohoto důvodu je naprosto nutné, aby případné změny v datech byly pouze změny obsahu a nikoliv změny v rozhraní. Tím se myslí, že uživatel nesmí přejmenovávat shadery, programy ani modely, nesmí rušit uniform proměnné, nemůže očekávat, že po vytvoření nových proměnných se aktualizuje rozhraní na jejich editaci atd. Pokud chce uživatel provést změny v C# implementaci, musí restartovat okno vizualizace.

Metody `SetCloseDelegate`, `SetLogDelegate`, a `SetErrorLineDelegate` nastaví delegát, který se dá používat pro zpětnou komunikaci mezi vizualizačním oknem a editorem. Editor si vytvoří delegát, který bude po zavolání vykonávat určitou činnost. Tento delegát se pošle vizualizačnímu oknu při vytváření okna. Vizualizační okno pak delegát může zavolat a tím spustí událost v editoru. Můžeme rovněž posílat parametry s hodnotami. Platí zde znovu přirozený předpoklad, že parametry budou jednoduché datové typy nebo instance tříd s atributem `Serializable`.

10 Správa zdrojů

Editor bude obsahovat podporu pro tři typy zdrojů – shadery, 3D modely a textury. Správa kódu shaderů již byla vysvětlena v předcházejících kapitolách o editorech zdrojového kódu. Dále bude uveden způsob zpracování souborů, do kterých se zdroje ukládají.

10.1 Systém projektů

Pro vytvoření jakéhokoliv efektu musí uživatel spravovat několik souborů najednou. Všechny soubory, které uživatel potřebuje na vytvoření jedné vizualizace, se ukládají do projektu. Projekt obsahuje souhrn všech uživatelských zdrojových kódů (GLSL i C# souborů), textur a modelů.

Další informaci, kterou v projektu potřebujeme uložit, ale nenáleží logicky žádnému z uvedených zdrojů, uložíme do vlastností projektu. Ve vlastnostech projektu je uložen seznam programů a seznam referencí.

Seznam programů obsahuje shadery, ze kterých se programy skládají. Navíc obsahuje seznam atributů pro daný program, které vstupují do vertex shaderu. Každá položka v seznamu má řetězcový název a celočíselný index, na který je atribut navázán. Když se posílá informace o programech do vizualizačního okna, okno nastaví indexy atributů podle uložených záznamů v seznamu. Uživatel samozřejmě může stejnou informaci napsat do zdrojového kódu vertex shaderu pomocí layout kvalifikátoru, jak je uvedeno v teoretické část (kapitola 2.2.5).

Reference potřebuje projekt pro kompilaci zdrojových kódů. Standardní reference odkazují na systémové knihovny a na knihovny OpenTK. Uživateli je umožněno přidávat reference na jiné další knihovny.

10.2 Načítání textur

Editor podporuje pouze načítání dvourozměrných textur. Textury jsou načítány v podobě obrázků. Editor dokáže obrázky pouze zobrazovat, nedokáže

je upravovat.

Obrázky se mezi aplikačními doménami nedají posílat jako objekt třídy `Bitmap` [30]. Pokud otevřeme obrázek na disku a načteme jej do objektu `Bitmap`, existuje zde propojení mezi daty na disku a mezi objektem v paměti. Celý obrázek se nekopíruje do paměti, protože by mohl zabrat příliš místa.

Bitmapy, které se posílají vizualizačnímu oknu, se převedou na pole bytů, které se následně dá serializovat. Vizualizační okno si opět vytvoří bitmapu z došlých dat. Aby textury nezabraly příliš mnoho paměti, tak se po inicializaci vizualizace nahrají na grafickou kartu a smažou se z operační paměti. Každá textura je pak přítomna v kódu vizualizace jen jednou, i když ji používají různé programy.

Na ukázce 10.1 je vidět nastavení textury pro program SP1. Program SP1 využívá tuto texturu na pozici `TextureUnit.Texture0`. Pro použití ji musíme před vykreslením objektu navázat na stav OpenGL pro program SP1. Když chce uživatel pro jeden program použít několik textur, musí před vykreslením všechny potřebné textury navázat na OpenGL. Každou texturu musí navázat pod jiným indexem. OpenTK verze 1.1 dovoluje nastavit textury `TextureUnit.Texture0` až `TextureUnit.Texture31`.

```
textures["scales"].SetShaderProgramUniform(shaderPrograms["SP1"],
    "scales", TextureUnit.Texture0);
...
textures["scales"].Bind(shaderPrograms["SP1"]);
```

Tabulka 10.1: Ukázka nastavení textury

10.3 Načítání a používání 3D modelů

Správa modelů souvisí se správou shaderů. Když potřebujeme vykreslit 3D model na scéně pomocí některého efektu, musíme propojit atributy programu s atributy modelu. Buď bude mít vextex shaderpro každý vstupní atribut kvalifikátory zarovnání, nebo uložíme informaci zvlášť do programu pomocí grafického rozhraní.

Aby mohly mít 3D modely propojené atributy s atributy shaderů, je

vhodné ukládat index atributů modelu do samotného souboru s modelem. V takovém případě není nutné ukládat tuto informaci někam stranou do zvláštního souboru s nastavením modelů a můžeme spravovat jediný soubor. Existující formáty pro 3D modely ale nebudou podporovat ukládání této dodatečné informace.

Pokud však nechceme, aby byla vizualizace modelů specifická pouze na implementovaný nástroj, je vhodné použít některý z existujících a používaných formátů pro 3D polygonální modely. Nejvýhodnější se jevilo použít formát collada, protože se ukládá ve formátu XML. Platforma .NET umožňuje jednoduché načítání XML souborů.

Formát collada má ale velmi složitou strukturu. Dokáže ukládat velké množství vlastností, které se v implementované aplikaci nevyužijí a nemůže jednoduše uložit informaci o indexu atributů. Pracovat s objektem, takového typu, který by v paměti uchovával všechny informace ze souboru collada je plýtvání pamětí.

Z toho důvodu byl zaveden jednoduchý vnitřní formát modelů. Formát má příponu rsm (render shader model). Objekt obsahující data ve formátu rsm používá editor i vizualizační okno. z toho důvodu je jeho implementace součástí knihovny `Interfaces`, která je zodpovědná za komunikaci mezi editorem a vizualizačním oknem. Aplikace umí upravovat hodnoty objektu formátu rsm v editoru a zároveň umí použít instanci objektu rsm formátu přímo ve vizualizačním okně pro vykreslení modelu. Použití vnitřního formátu má tu výhodu, že při případné implementaci načítání z dalších formátů bude přítomna jednotná reprezentace modelů.

10.3.1 Popis vnitřního formátu rsm

Formát rsm má omezenější vlastnosti než formát collada. Byl navržen tak, aby neobsahoval žádné jiné informace než popis geometrie a aby využíval technologie .NET (export do xml, nastavování jeho hodnot pomocí komponenty PropertyGrid a serializaci). Zároveň byl navržen tak, aby uživatel, který bude používat editor GLSL byl schopný jej intuitivně používat v programovém kódu. Každý rsm model obsahuje následující:

- seznam atributů

  - jméno (řetězcová proměnná, sémantická informace o druhu atributu)
  - seznam parametrů (seznam řetězcových jmen parametrů, například X, Y, Z)
  - index pro svázání s atributem shaderu (číslo, které využívá první fáze grafického řetězce – vertex fetch)
  - index pro svázání s OpenGL bufferem při vykreslování
- pole indexů
- typ primitiva

Model rsm, v jehož seznamu atributů se žádný atribut nenachází, je stále validní. Soubor s prázdným seznamem atributů ale nemůže popisovat žádný polygonální model. Předpokládá se, že hodnoty v polích atributů jsou vždy čísla s pohyblivou řádovou čárkou typu float. Pole indexů je vždy pole celých nezáporných čísel typu uint.

Formát rsm může pouze uchovat jednu geometrii. Nemůže v sobě uchovávat více částí. Předpokládá se, že jeho atributy jsou vždy hodnoty jeho pole vrcholů. Počet hodnot v poli lomeno počtem parametrů každého atributu musí dávat konstantní hodnotu. Standardně bude v seznamu atributů nejméně jedno pole float čísel, které určuje pozice vrcholů.

Na rozdíl od formátu collada je vnitřní formát poměrně náročný na velikost uložené souboru. Každé float číslo ukládá do souboru zvlášť mezi dvojici tagů, které udávají jeho typ. Jako vlastnost to může být nepříjemné při ukládání větších modelů. Efektivita vykreslování ani náročnost na paměť ale ovlivněna není, protože tato vlastnost se týká pouze textového xml souboru na disku. Formát XML souboru je navržen kvůli standardnímu chování třídy `XmlSerializer` a snadnému uložení. Ve chvíli, kdy je objekt načten do paměti, není v ní uchovávána žádná redundantní informace.

Příklad

Máme 10 vrcholů, které mají tři souřadnice X, Y, z a zároveň mají texturovací souřadnice U a V. Uložený model bude mít právě dva atributy v seznamu atributů (standardně pojmenované POSITION a TEXCOORD, ale mohou mít libovolná řetězcová jména). První atribut pro pozici bude mít přiřazen

jedno pole 30 float čísel. Druhý atribut pro texturovací souřadnice bude mít přiřazen jedno pole 20 float čísel.

Pokud bude mít model nastaven typ primitiva jako trojúhelníky, bude se pole indexů chápat, jako že jsou hodnoty sdruženy ve skupinách po třech. Každá trojice indexů ukazuje na tři vrcholy, které tvoří jeden trojúhelník.

10.3.2 Formát collada

Collada podporuje nejvíce vlastností v porovnání s ostatními formáty. Hlavním důvodem pro výběr formátu collada byla možnost jednoduše načíst data z tohoto formátu do paměti. Collada soubor je XML soubor. Jazyk C# nabízí celkem jednoduché nástroje pro získávání dat z XML souborů. Pokud jsou všechny vlastnosti objektu veřejné a neexistuje žádný konstruktor s parametry, je ukládání do XML a načítání z XML souborů triviální. Pro načtení formátu do struktury v paměti byla stažena již existující implementace struktury collada [28].

Používané elementy[27]

Protože chceme převádět model z formátu collada do formátu rsm, bude nejprve uveden a vysvětlen seznam relevantních elementů, které collada používá pro uložení dat týkajících se geometrie:

- `library_geometries`
  - obsahuje elementy geometry
  - collada formát může mít definováno několik geometrií v jednom souboru
- `geometry`
  - obsahuje element mesh
  - popisuje vizuální tvar a vzhled objektu na scéně
- `mesh`
  - obsahuje popis geometrie – elementy source, vertices, triangles a všechny ostatní elementy podle typu primitiv

  - popisuje polygonální model na základě vrcholů a informací o použitých primitivech

- `source`
  - obsahuje například elementy `float_array` nebo `technique_common`
  - deklaruje data, na která se odkazuje element input
- `input`
  - využívá se například v elementu vertices nebo v elementu triangles
  - popisuje sémantiku datového zdroje
- `vertices`
  - obsahuje elementy input
  - popisuje atributy a identitu vrcholů polygonálního modelu
- `triangles`, `trifans`, `tristrips`, `lines`, `line_strips`, `polygons`, `polylist`, `spline`
  - obsahuje elementy input
  - poskytuje informaci potřebnou k svázání atributů vrcholů a vytvoření jednotlivých primitiv z daných vrcholů (trojúhelníků, trojúhelníkových vějířů, trojúhelníkových pásů, čar a dalších druhů primitiv)
- `technique_common`
  - v rámci geometrie obsahuje elementy accessor
  - specifikuje informaci o atributu, všechny implementace formátu collada musí podporovat tento element
- `accessor`
  - obsahuje elementy param
  - popisuje přístup k polím jako například `floar_array`
- `param`
  - využívá se v elementu accessor
  - popisuje jméno a typ jednoho parametru

- `float_array`
  - využívá se v elementu source
  - obsahuje jednorozměrné pole float číselných hodnot

Formát collada může mít složité indexování. Sice podporuje pouze jedno pole idexů pro jednu geometrii, ale v tomto poli může ukládat několik skupin najednou. V takovém případě nastaví u každého elementu `input` vlastnost `offset`, která určí pozici, nakteré začíná v poli indexů první hodnota pro daný element `input`. Hodnota `stride` se pak rovná počtu elementů input v jednom z elementů pro specifikaci primitiva.

10.3.3 Konverze do rsm souboru

Formát collada je velice obsáhlý. Při importu se bude počítat pouze s nejčastějšími zápisy modelů. Model ve formátu collada musí splňovat některé požadavky, aby bylo možné jej převést. Model musí být složený z trojúhelníků nebo záplat. Veškerá jeho data musí být uložena v polích float čísel. Model musí být navíc vždy indexovaný.

Formát rsm podporuje právě jedno pole indexů. Z toho vyplývá, že při převodu se může ztratit některá informace. Formát rsm navíc může uložit pouze jednu geometrii. Pokud má soubor dae uloženo v sobě více částí geometrie, exportér vybere všechny geometrie a z každé vytvoří jeden rsm soubor.

10.3.4 Rozdíl v indexování mezi formáty collada a rsm

Formát collada může indexovat vrcholy a jeho dodatečné informace zvlášť. Počty vrcholů a počty normál se například nemusí rovnat. Pokud máme vrcholy tvořící krychli a chceme mít v souboru uloženy i normály, budeme pravděpodobně chtít, aby každý vrchol krychle měl tři různé normály. Ve vrcholech krychle se ostře mění směr normály, a nemůžeme pro dobrý efekt použít průměrnou hodnotu normál sousedních povrchů nebo stěn.

Formát collada v takovémto případě umožňuje mít uloženo třeba jen pouze 8 vrcholů a 6 normál. Výsledné pole indexů bude popisovat 12 trojúhelníků (6 stěn, každá se skládá z 2 trojúhelníků), tudíž bude mít 36 hodnot.

Potřebujeme ještě jedno pole indexů, které bude popisovat normály. Toto pole bude stejně dlouhé, ale bude se muset skládat z jiných hodnot.

Formát rsm takto strukturovaný není, protože OpenGL nedokáže s podobnou informací pracovat. Pokaždé, když pošleme do vertex shaderu vrchol, předpokládáme, že jeho pozice je poslána zároveň se všemi atributy daného vrcholu. Nemůžeme shaderu poslat 8 pozic a 6 normál. Musíme v tomto případě shaderu poslat nejméně 24 záznamů (3 krát 8 vrcholů) Každý fyzický vrchol se objevuje ve třech kopiích. Každá z těchto kopií má jinou normálu.

Algoritmus pro převod indexů z formátu collada do formátu dae funguje následujícím způsobem:

1. rozdělíme pole indexů
   (a) $I_{vrcholy}$ – pole indexů indexující souřadnice vrcholů
   (b) $I_1 \ldots I_n$ – pole indexů indexující hodnoty dalších atributů v jiných polích
2. máme pole atributů
   (a) Vvrcholy – pole float čísel se souřadnicemi vrcholů se složkami X, Y, Z
   (b) $V_1 \ldots V_n$ – pole float čísel s hodnotami dalších atributů náležících vrcholům
3. chceme vytvořit nová pole atributů, která mají jiné velikosti a jiná pořadí hodnot
   (a) $W_1 \ldots W_n$ – pole float čísel s hodnotami dalších atributů uspořádána tak, aby aby na ně místo indexů $I_1 \ldots I_n$ odkazovaly pouze indexy Ivrcholy
4. každé pole hodnot Vvrcholy, $V_1 \ldots V_n$ má jiný počet float čísel na jeden atribut
   (a) $a_{vrcholy} = 3$; vrcholy mají z pravidla tři souřadnice X, Y, Z
   (b) $a_1 \ldots$ an jsou různá čísla; závisí na tom, kolik v souboru collada obsahuje element accessor elementů parametr pro každé pole atributů
5. pro všechny i od 1 do n (vytváříme W_i za pomocí I_i)

(a) W_i = new uint[$V_{vrcholy}$.Length / $a_{vrcholy}$ * a_i]

(b) pro všechny j od 1 do I_i.Length

i. starý index $o = I_{vrcholy}$[j]

ii. nový index $p = I_i$[j]

iii. zkopírujemprvek z pole V_i a pozice o do pole W_i na pozici p

Ve výše nastíněném algoritmu je vynechán pouze implementační detail, kdy kopírujeme element z pole V_i do pole W_i. Elementem se myslí ai po sobě jdoucích čísel, protože pole V_i i W_i jsou pouze jednorozměrná pole float čísel. Pokud máme pole barev s hodnotami RGB, bude $a_i = 3$ a proto budeme kopírovat tři hodnoty z pole V_i do pole W_i. Index o i index p se pak násobí třemi a kopírují se tři čísla.

Nevýhoda předešlého algoritmu je ta, že se spoléháme na to, že pole vrcholů udává správné pořadí a správný počet vrcholů. Pokud by se na jeden vrchol (se stejným indexem) v poli hodnot $V_{vrcholy}$ odkazovalo při indexování vícekrát a po každé by mu byla přiřazenajiná skupina čísel z pole V_i, pak se dostáváme do situace, kterou nejde popsaným algoritmem řešit. v tomto případě aplikace pouze oznámí, že došlo k případu, kdy je jednomu vrcholu přiřazováno více různých atributů a použije pouze první přiřazený.

11 Sestavení aplikace

Aplikace se skládá z editoru programového kódu a z vizualizačního okna. Hlavní projekt implementuje vzhled a funkcionalitu editoru.

Vizualizační okno se vytváří až za běhu aplikace kompilací zdrojových kódů přiložených k aplikaci. Existuje fixní část zdrojových kódů, která vytváří vzhled vizualizačního okna a jeho základní funkčnost. Knihovny, které jsou distribuovány s vytvořeným editorem::

- Interfaces.dll – vlastní knihovna, která se používá k manipulaci s vizualizačním oknem; důvod pro používání rozhraní je vysvětlen v kapitole 9.3
- OpenTK.dll 1.1.0 – manipulace s OpenGL funkcemi [29]
- FATabStip – grafická komponenta záložky pro otevřené soubory [26]
- FastColoredTextbox – grafická komponenta pro editování a doplňování kódu [20]

Editor má pro zobrazování zavedeny čtyři perspektivy, které odpovídají složkám v projektu:

- zobrazování C# kódu – složka customCode
- zobrazování GLSL kódu – složka shaders
- zobrazování modelů – složka models
- zobrazování obrázků – složka textures

Každá perspektiva nastaví jednu z komponent FATabStrip [26] na viditelnou. Podle perspektivy se přepneme buď na editor C# kódu, editor GLSL kódu, zobrazování modelů nebo zobrazování obrázků. Pro každý soubor, který otevřeme z adresářového stromu vlevo, se nám otevře nová záložka. v záložce se objeví buď editor pro úpravu zdrojového kódu, `PropertyGrid` na úpravu vlastností modelu, nebo bitmapa na prohlédnutí textury.

12 Závěr

Předmětem diplomové práce bylo vyvinout editor pro snadnou úpravu GLSL kódu a následnou vizualizaci výsledků práce.

Bylo provedeno vyhodnocení existujících nástrojů, které se pro podobné účely používají. Nejprofesionálnější a nejrozsáhlejší projekty, které umožňují uživateli měnit GLSL kód a sledovat provedené změny, jsou nástroje, které jsou v první řadě určené pro ladění již hotových OpenGL aplikací. Nástroje pro ladění aplikací se zaměřují především na sledování výkonu a krokování. Z aplikací, které dávaly důraz na úpravu shaderů a celé scény, se nejvíce osvědčil program AMD RenderMonkey. RenderMonkey má několik nevýhod. Zejména jeho ukončený vývoj a neschopnost upravovat vykreslovací smyčku.

Byl implementován nástroj pro rychlé prototypování GLSL projektů. Vyvinutý nástroj umožňuje vývoj efektů psaných v GLSL a jejich vizualizaci. Editor dává uživateli při vývoji více možností než jiné testované nástroje používané pro rychlé zobrazování efektů. Uživatel si může napsat svůj vlastní kód pro inicializaci vizualizace a vykreslovací smyčku. Uživatel může vytvořit projekt s několika objekty, texturami a programy, které budou použity k vykreslení scény. Protože se řídící kód vizualizační smyčky píše v jazyce C#, není uživatel omezený při vytváření vlastních efektů tolik, jako při práci s jinými nástroji, které umožňují pouze fixní nastavení vizualizace. Editor pomáhá uživateli se orientovat v kódu pomocí zvýrazňování a zároveň umožňuje uživateli do jisté míry doplňovat kód.

Omezení využití tohoto nástroje plynou hlavně z použité komponenty pro editaci zdrojového kódu. Zvýrazňování i doplňování kódu funguje na základě regulárních výrazů. Není implementován přístup k žádné abstrakci zdrojového kódu. Při doplňování kódu není implementovaný žádný sofistikovaný systém pro zapamatování uživatelsky definovaných proměnných.

Dalším omezením nástroje je nutnost přeložit celý zdrojový kód a restartovat vizualizační aplikaci pokaždé, když se provede změna v C# zdrojovém kódu. Pro jednoduché vizualizace není problém okno restartovat, protože překlad trvá pouze pár sekund. Kdyby bylo umožněno uživateli měnit probíhající kód přímo při běhu, byla by kladena na uživatele mnohem větší zodpovědnost. Uživatel by se v některých případech musel sám starat o správné načítání, používání a uvolňování zdrojů.

Přehled termínů

wrapper – tenká vrstva kódu, který převádí rozhraní nativní knihovny do rozhraní kompatibilní s jiným jazykem

IDE – vývojové prostředí či software usnadňující práci programátorů

proxy – prostředník při vzdáleném volání mezi klientem a serverem

assembly – kompilovaný kód .NET aplikace připravený pro nasazení; může se jednat o spustitelný soubor (exe) nebo dynamicky linkovanou knihovny (dll); tento kód je kompilován za běhu aplikace do strojového jazyka

shader – množina řetězců kompilovaná pro určitou fázi v grafickém řetězci (GLSL terminologie)

Vs – vertex shader

TCs – tessellation control shader

PG – generátor primitiv

TES – tessellation evaluation shader

GS – geometry shader

Fs – fragment shader

GLSL – jazyk pro programovací grafické karty pro OpenGL

OpenGL – API grafické knihovny pro vykreslování 2D a 3D grafiky; je multiplatformní a nezávislé na programovacím jazyku

API – programové rozhraní knihovny, které specifikuje, jak mají jednotlivé

softwarové komponenty spolu komunikovat

MDI – anglicky Multiple Document Interface je popis grafického rozhraní, kdy jedno hlavní okno obsahuje jedno nebo více oken; okna, která jsou potomky hlavního okna není možné zobrazit samostatně bez aplikace a nelze je zobrazit mimo plochu pokrytou hlavním oknem

snipplet – malá část kódu znovu použitelného programového kódu

Literatura

[1] ROST, Randi J. *OpenGL Shading Language*. Hillsboro, OR, U.S.A.: Addison-Wesley, 2004. ISBN 0-321-19789-5, str. 34.

[2] SELLERS, Graham, WRIGHT, JR. a Nicholas HAEMEL. *OpenGL SuperBible: Sixth Edition*. Portland, OR, U.S.A.: Addison-Wesley, 2014. ISBN 978-0-321-90294-8.

[3] *OpenGL - The Industry Standard for High Performance Graphics* [online]. 1997, 2014 [cit. 2014-05-14]. Dostupné z: http://www.opengl.org

[4] *OpenGL Shading Language - OpenGL.org* [online]. 2009, 2014 [cit. 2014-05-14]. Dostupné z: http://www.opengl.org/wiki/OpenGL_Shading_Language

[5] *OpenGL Shading Language* [online]. 1997, 2014 [cit. 2014-05-14]. Dostupné z: http://www.opengl.org/documentation/glsl/

[6] KESSENICH, John, Dave BALDWIN a Randi ROST. *The OpenGL Shading Language* [online]. 2004 [cit. 2014-05-14], str 56. Dostupné z: http://www.opengl.org/registry/doc/GLSLanGSpec.Full.1.10.59.pdf.

[7] *RenderMonkey^{TM} Toolsuite | AMD* [online]. 2008 [cit. 2014-05-14]. Dostupné z: http://developer.amd.com/resources/archive/archived-tools/gpu-tools-archive/rendermonkey-toolsuite/

[8] *GLSL Hacker - Pixel hacking with GLSL, Lua and Python - Scriptable 3D Game Engine | Geeks3D.com* [online]. 2012, 2013 [cit. 2014-05-14]. Dostupné z: http://www.geeks3d.com/glslhacker/overview.php

[9] *GDEBugger Tutorial - OpenGL Debugger and Profiler* [online]. 2004, 2010 [cit. 2014-05-14]. Dostupné z: http://www.gremedy.com/tutorial/

[10] *GlslDevil - OpenGL GLSL Debugger* [online]. 2007, 2010 [cit. 2014-05-14]. Dostupné z: http://www.vis.uni-stuttgart.de/glsldevil/

[11] *Kick.js | Shader Editor* [online]. 2012, 2014 [cit. 2014-05-14]. Dostupné z: http://www.kickjs.org/example/shader_editor/shader_editor.html

[12] *TyphoonLabs' OpenGL Shader Designer* [online]. 1997, 2012 [cit. 2014-05-14]. Dostupné z: http://www.opengl.org/sdk/tools/ShaderDesigner/

[13] *GLSL Shader Maker* [online]. 2013, 2013 [cit. 2014-05-14]. Dostupné z: http://shadermaker.codeplex.com

[14] *GDEBugger | AMD* [online]. 2012, 2012 [cit. 2014-05-14]. Dostupné z: http://developer.amd.com/tools-and-sdks/archive/amd-gdebugger/

[15] *CodeXL | AMD* [online]. 2014, 2014 [cit. 2014-05-14]. Dostupné z: http://developer.amd.com/tools-and-sdks/opencl-zone/opencl-tools-sdks/codexl/

[16] *Scintilla and SciTE* [online]. 1999, 2014 [cit. 2014-05-14]. Dostupné z: http://www.scintilla.org

[17] *ScintillaNET* [online]. 2007, 2014 [cit. 2014-05-14]. Dostupné z: http://scintillanet.codeplex.com

[18] *Using ICSharpCode.TextEditor - CodeProject* [online]. 2008 [cit. 2014-05-14]. Dostupné z: http://www.codeproject.com/Articles/30936/Using-ICSharpCode-TextEditor

[19] *GDI+ (Windows)* [online]. 2012 [cit. 2014-05-14]. Dostupné z: http://msdn.microsoft.com/en-us/library/windows/desktop/ms533798(v=vs.85).aspx

[20] *Fast Colored TextBox for Syntax Highlighting - CodeProject* [online]. 2013, 2014 [cit. 2014-05-14]. Dostupné z: http://www.codeproject.com/Articles/161871/Fast-Colored-TextBox-for-syntax-highlighting

[21] *CleanCode C# Libraries v1.2.03 API* [online]. 2001, 2013 [cit. 2014-05-14]. Dostupné z: http://cleancode.sourceforge.net/api/csharp/

[22] *Clean Code* [online]. 2013 [cit. 2014-05-14]. Dostupné z: http://en.sourceforge.jp/projects/sfnet_cleancode/downloads/cleancode/csharp/cleancode-csharp-v1_1_04.zip

[23] *Application Domains* [online]. 2014 [cit. 2014-05-14]. Dostupné z: http://msdn.microsoft.com/en-us/library/2bh4z9hs(v=vs.110).aspx

[24] *Do The Right ThinGS* [online]. 2009 [cit. 2014-05-14]. Dostupné z: http://do-the-right-thinGS.bloGSpot.cz/2009/07/cross-appdomain-winform-implementation.html

[25] *LifetimeServices Class (System.Runtime.Remoting.Lifetime)* [online]. 2014 [cit. 2014-05-14]. Dostupné z: http://msdn.microsoft.com/en-us/library/system.runtime.remoting.lifetime.lifetimeservices(v=vs.110).aspx

[26] *FATabStrip.cs - A TabControl in the Visual Studio 2005 S* [online]. 2006 [cit. 2014-05-14]. Dostupné z: http://www.codeforge.com/read/54729/FATabStrip.cs__html

[27] *COLLADA - 3D Asset Exchange Schema* [online]. 2014 [cit. 2014-05-14]. Dostupné z: http://www.khronos.org/collada/

[28] *C# Collada 1.5 Classes* [online]. 2013 [cit. 2014-05-14]. Dostupné z: http://sourceforge.net/projects/csharpcollada

[29] *The Open Toolkit library | OpenTK* [online]. 2014 [cit. 2014-05-14]. Dostupné z: http://www.opentk.com/project/opentk

[30] *Bitmap Class (System.Drawing)* [online]. 2014 [cit. 2014-05-14]. Dostupné z: http://msdn.microsoft.com/en-us/library/system.drawing.bitmap(v=vs.110).aspx

[31] KESSENICH, John, Dave BALDWIN a Randi ROST. *GLSLanGSpec.4.30.6. OpenGL Shading Language*. 2012, str. 10 - 72. Dostupné z: www.opengl.org/registry/doc/GLSLanGSpec.4.30.6.pdf

Přílohy

Uživatelská příručka

Požadavky

Pro spuštění aplikace je nutné, aby uživatel měl nainstalováno následující:

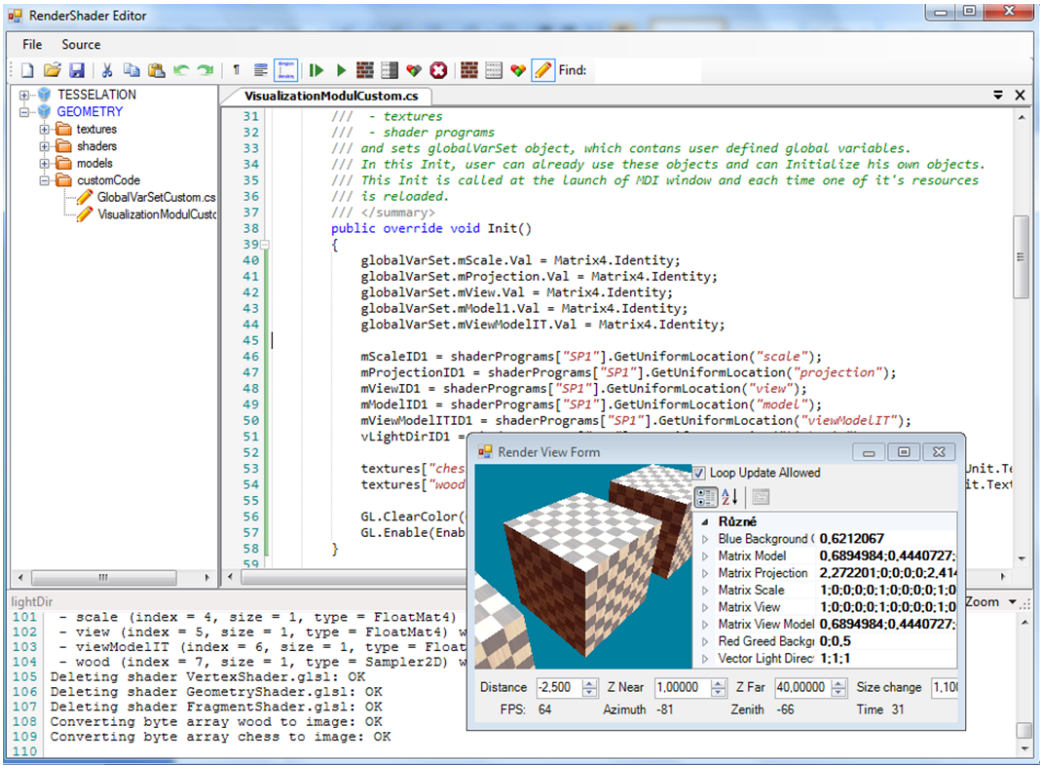
- operační systém Windows Vista nebo vyšší verzi
- platformu .NET verzi 4.5
- knihovnu OpenGL verzi 4.3

Grafické rozhraní

Aplikace spouští editor pro úpravu zdrojového kódu a dalších zdrojů (obrázek 12.1). V horní části okna je hlavní lišta. Pod ní panel nástrojů. V levé straně okna je strom projektů. V pravé části okna je správa programů (zobrazuje se pouze při spuštěné perspektivě pro shadery). V dolní části okna je logovací box pro popis událostí.

Panel nástrojů

Na panelu nástrojů je mnoho voleb znázorněných ikonkami. První skupina obsahuje volby pro práci s projektem. Druhá skupina obsahuje volby pro úpravy zdrojových souborů. Třetí skupina ovládá zobrazování pomocné informace, jako zobrazování netisknutelných znaků, zvýrazňování řádky s kurzorem nebo vodící tečky u bloků kódu.

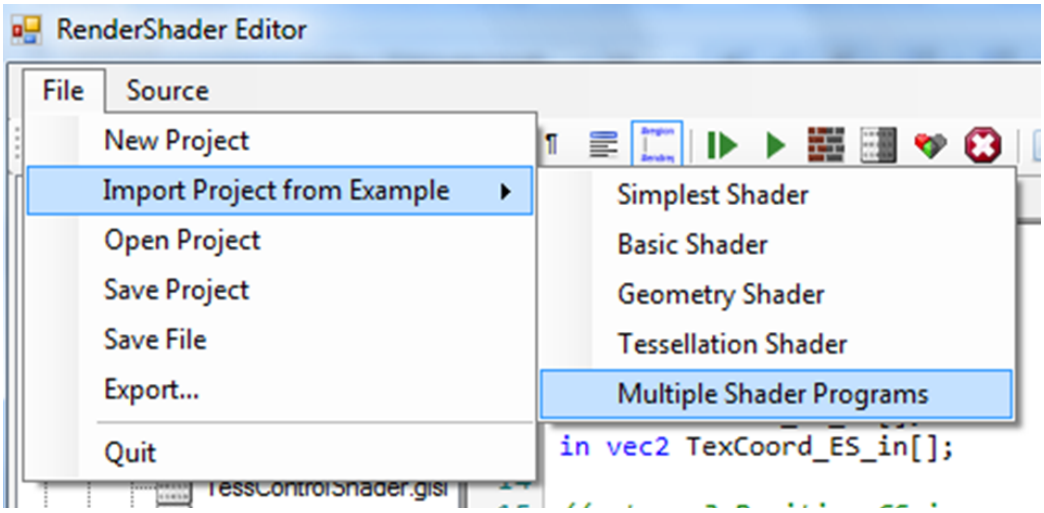


Obrázek 12.1: Vzhled aplikace

V hlavní nabídce můžeme načíst existující projekt, vytvořit nový projekt nebo vytvořit projekt z ukázkového příkladu pod jiným jménem. Pro první používání se doporučuje vytvořit kód z ukázkového příkladu, protože pro spuštění a rozběhnutí prázdného projektu je zapotřebí udělat několik úprav. Ukázkové příklady je možné spustit ihned.

Je možné vytvořit projekt z poskytnutých příkladů. Nejprve načteme jeden z ukázkových projektů (File/Import Project from Example/...). Nový projekt je možno vytvořit podle obrázku 12.2. Pojmenujeme projekt novým jménem. Všechny soubory, které tvořily ukázkový projekt, se zkopírují do složky s projekty (projects) a projekt se přejmenuje.

Další skupina obsahuje nástroje pro ovládání vizualizace. První volba zkompiluje projekt a okamžitě ho spustí. Druhá volba spustí poslední přeloženou verzi projektu. Pokud ještě žádná předešlá verze není, nebo pokud je poslední přeložený projekt ze složky manuálně smazán, volba spuštění poslední přeložené verze skončí chybou. Další případ, kdy volba skončí chybou je tehdy, když není načtený projekt v lokální složce s ostatními projekty.



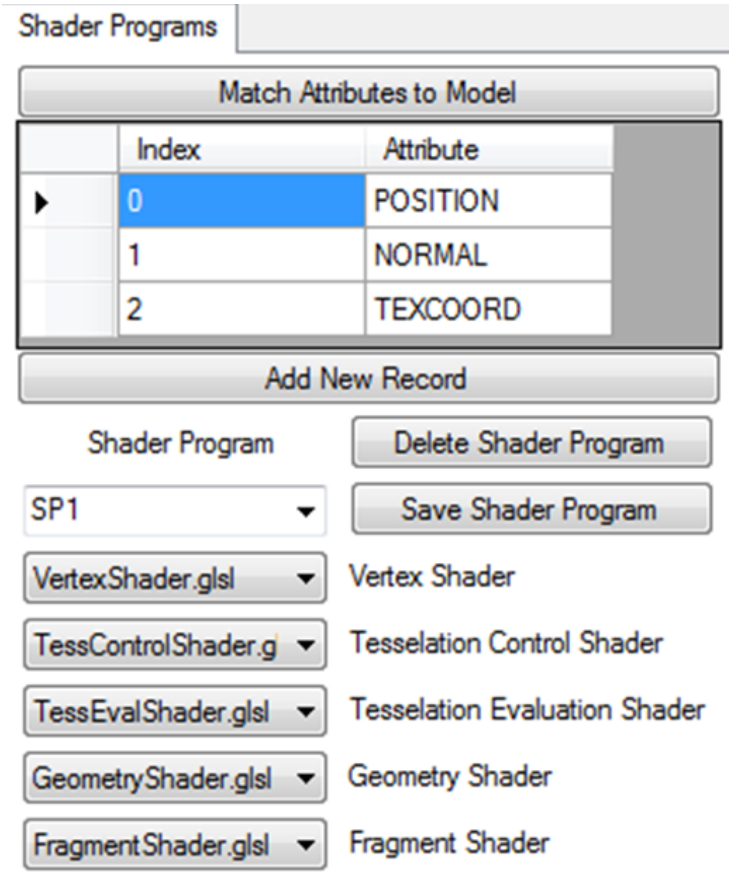
Obrázek 12.2: Vytvoření nového projektu z příkladu

Volba kompilace provede kompilaci pouze C# zdrojových kódů. Kompilace shaderů probíhá a při spuštění vizualizace, protože shadery jsou kompilovány ve spuštěném okně.

Další tři volby fungují jen pokud je vizualizace spuštěna. Slouží pro aktualizaci textur, shaderů a modelů za běhu programu. Všechny zdroje jedné kategorie se aktualizují najednou. Pokud je v nových zdrojích chyba nebo jsou jinak pojmenovány, vizualizace se zastaví. Zdrojový kód aplikace zůstává stejný, takže automatické zpracování zdrojů je musí najít podle předchozích řetězcových jmen.

Správa programů

Správa programů se objevuje pouze při aktivní perspektivě pro shadery (obrázek 12.3). v horní části je tabulka, která popisuje mapování atributů mezi programy a modely. Program se skládá ze shaderů, které jsou vybrány v dolní části panelu pro správu programů. Pokud chceme vytvořit nový program, musíme nejdříve napsat jeho název (totožný název budeme používat v C# kódu při vybírání ze slovníku), dále vybereme shadery, které bude používat, a nakonec klikneme na tlačítko `Save Shader Program`. Pokud chceme některý program vymazat, vybereme jej a klikneme na tlačítko `Delete Shader Program`. K existujícím programům můžeme přidat do tabulky mapování atributů buď manuálně přidáním nového záznamu, nebo



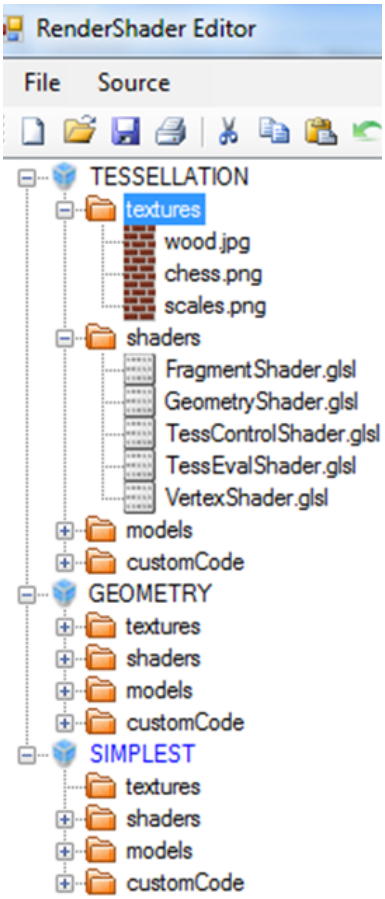
Obrázek 12.3: Panel správy programů

automaticky stiskem tlačítka `Match AttribuTES to Model`. Zobrazí se nabídka, ve které vybereme model, jehož atributy chceme použít. Atributy modelu a jeho indexy se zkopírují do tabulky. Pokud bychom některý atribut použít nechtěli, označíme řádek v tabulce kliknutím nalevo od daného řádku a stiskem klávesy delete smažeme záznam. Pokud zjistíme, že model obsahuje chybné atributy, musíme je opravit přímo v nastavení modelu.

Strom projektu

V levé části okna je strom otevřených projektů (obrázek 12.4). Otevíraný projekt automaticky načte do stromu všechny soubory se správnými příponami ve svých složkách. v tomto stromu můžeme vybrat jakýkoliv zdroj dvojím

poklepáním na jeho ikonu soubor.



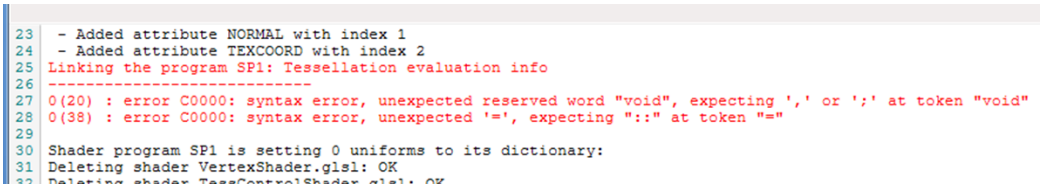
Obrázek 12.4: Strom projektů

Pravým kliknutím kdekoliv na strom se objeví dialogové okno, které umožní nastavit označený projekt jako hlavní projekt, změnit reference projektu, přidat nový zdrojový soubor, importovat model a smazat nebo přejmenovat označený soubor. Přejmenovat soubor lze také dvojím kliknutím na text souboru. Změnit příponu souboru je zakázáno. Přejmenovávat složky je také zakázáno.

Textový log

Vypisuje některé důležité události. Nejdůležitější událostí jsou chyby při kompilaci. Vypíše typ chyby a řádku, ve které chyba nastala. Editor zároveň

otevře soubor, ve kterém se nachází chyba a označí červeně chybnou řádku v kódu.



Obrázek 12.5: Vzhled logovacího okna

V logovacím okně (viz 12.5) jsou obyčejné události vypisovány černou barvou, chyby červenou barvou a varování modrou barvou. Logovací okno se automaticky posunuje na konec když v něm není umístěn kurzor.

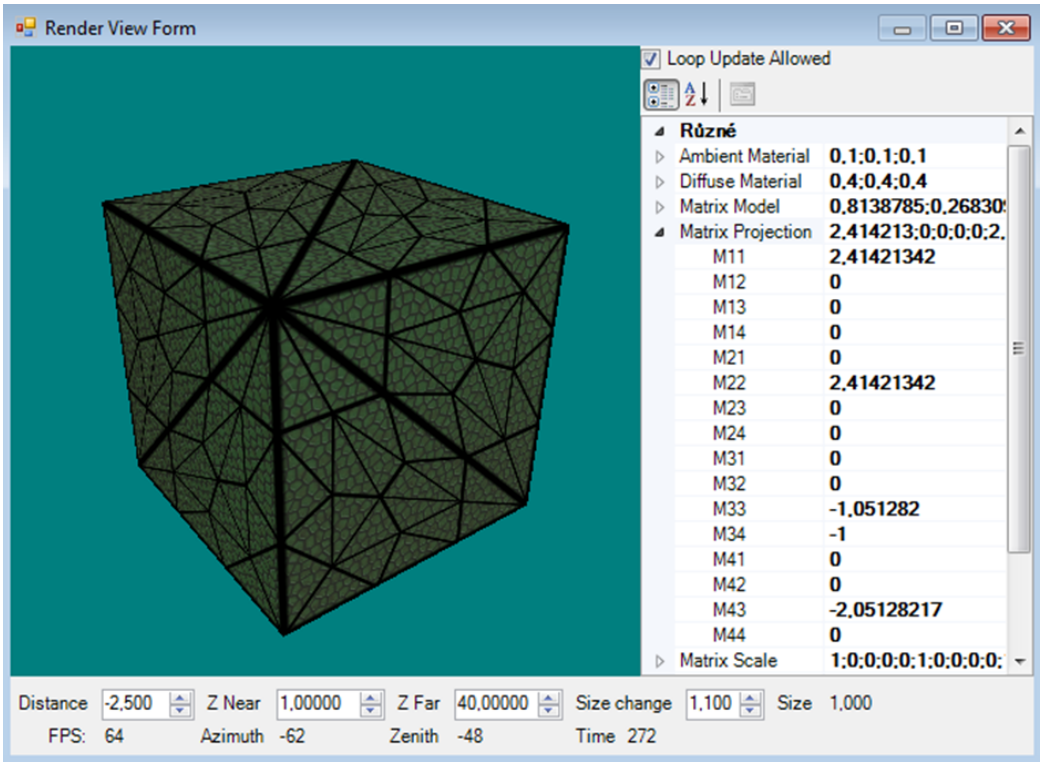
Práce s vizualizačním oknem

Při spuštění projektu se objeví nové okno ve formě MDI potomka. v dolní části vizualizačního okna se nachází komponenty pro vkládání nejčastěji používaných vstupů, jako vzdálenost objektu od kamery, rychlost změny velikosti při rolování kolečka myši a ořezávací roviny.

V pravé části okna (viz 12.6) vidíme komponentu `PropertyGrid`, která zobrazuje a nastavuje uživatelské globální proměnné.

Otáčení modelu v poskytovaných příkladech (jen v těch, ve kterých jsou implementovány transformace) se provádí stisknutím a držením levého tlačítka myši nad kreslicí plochou a následným pohybem myši. Když pracujeme s vizualizačním oknem a kurzor není v žádné komponentě pro uživatelský vstup, kolečko myši ovlivňuje škálování scény. v pravé horní části vizualizačního okna se dá vypnout aktualizace scény vyvolané příchodem dalších snímků.

Všechny vstupy vytvořené v grafickém rozhraní mají svojí funkcionalitu implementovanou v kódu, který může uživatel měnit. Automaticky probíhá pouze propagace hodnot do tříd, ke kterým má uživatel přístup. Uživatel se může rozhodnout napojit úplně jiný obslužný kód na změnu číselné hodnoty pro škálování, vyvolané pohybem kolečka myši. Rozhraní vizualizačního okna bylo takto navrženo pouze z toho důvodu, že škálování je jedna z akcí, kterou uživatel bude chtít provádět nejčastěji, a napojit ji na posuv kolečka myši je



Obrázek 12.6: Vzhled vizualizačního okna s načteným ukázkovým tessellačním projektem

nejvíce intuitivní chování. Měnit škálování scény nebo natočení modelu psaním číselných hodnot do komponenty `PropertyGrid` není příliš uživatelsky přívětivé.

Úpravy shaderů v editoru

V adresářovém stromě projektu si můžeme najít soubory GLSL, které projekt používá. Editor shaderů doplňuje přednastavené proměnné, funkce a vyhrazená slova. Navíc editor doplňuje názvy uniform proměnných, které jsou uložené v programu. Názvy uniform proměnných se pošlou našeptávači pokaždé, kdy je uložen shader. Uniform proměnné se sdílí pouze mezi shadery v rámci jednoho programu, proto je nutné nejdříve shadery spojit, než se může začít využívat této vlastnosti. Shadery se spojují do programu v pravé části okna při zapnuté perspektivě pro shadery. Po nastavení shaderů pro vybraný (nebo nový) program je nutné kliknout na tlačítko pro uložení programu.

Úpravy C# zdrojového kódu v editoru

Zdrojový kód napsaný v C# musí být poskytnut nejméně pro implementaci vlastního vizualizačního modulu a vlastního souboru globálních proměnných. Uživatel si může vytvořit další cs soubory s vlastním kódem, který se dá využívat při implementaci kódu projektu.

Uživatel by se měl řídit podle nápovědy psané v komentářích ukázkových projektů. U každé metody, kterou může uživatel implementovat, je popsáno, kdy se volá a pro jakou činnost se hodí.

Metoda Init

Metoda Init se volá po automatické inicializaci všech zdrojů. Uživatel si v ní bude chtít hlavně uložit indexy uniform proměnných pro následné nastavování jejich hodnot. Je samozřejmě možné neustále přistupovat ke všem hodnotám přes řetězcové konstanty, ale tento přístup může být velice pomalý. V případě, že provádíme aktualizaci v každém snímku, tak se jeví nesmyslné zjišťovat index uniform proměnné z její řetězcové konstanty před každým nastavením její hodnoty. Tato informace by se neměla ukládat nikde jinde než v metodě Init. Pokud se změní množina nebo pořadí uniform proměnných, mohlo by se stát, že stejné uniform proměnné budou mít po aktualizaci jiné indexy. Pokud nastavení indexů bude probíhat v metodě Init, změny nezpůsobí potíže, protože Init se volá při každém restartu zdrojů.

Metoda InputUpdate

Metoda InputUpdate se volá při každé změně hodnot způsobené interakcí s uživatelským rozhraním. Je vytvořena za účelem volat kód, který propaguje hodnoty získané z uživatelského rozhranní do shaderů. Všechny hodnoty z uživatelského rozhraní jsou uloženy buď v chráněných atributech třídy nebo v objektu `globalVarSet`. Volání metody nastává v případech:

- držení levého tlačítka myši a tažení po zobrazovací ploše
- otáčení kolečka myši

- roztahování okna
- nastavování hodnot v komponentě PropertyGrid
- nastavování hodnot ve všech ostatních komponentách pod zobrazovací plochou (s výjimkou rychlosti škálování, protože se nemění hodnota škálování ale pouze velikost kroku při otáčení kolečka myši)

Metoda LoopUpdate

Metoda LoopUpdate se volá před každým vykreslením snímku, pokud uživatel v grafickém rozhraní má zaškrtnuté políčko (vpravo nahoře) pro aktualizaci ve smyčce.

Důvod pro možnost zakázat volání metody je v aktualizaci globálních proměnných, které se zobrazují v komponentě `PropertyGrid`. Pokud je výpočet jejich hodnot prováděn v této metodě na základě pouze již nastavených hodnot, nemůže uživatel změnit žádné hodnoty, které se v této metodě aktualizují. V dalším snímku po vykonané změně by se hodnoty opět přehrály na hodnoty vypočtené při aktualizaci animace. Komponenta `PropertyGrid` je nastavena tak, aby se přestala aktualizovat v okamžiku, kdy do ní vstoupí kurzor. Zdroj dat pro komponent se neustále aktualizuje. Uživatel může díky zastavené aktualizaci uživatelského rozhraní měnit hodnoty globálních proměnných, které nejsou počítány ve smyčce.

Doporučuje se do této metody psát pouze kód, který se opravdu musí aktualizovat v každém snímku, jako například animace závislá na čase. Pokud stačí aktualizace na uživatelský vstup, měl by se kód psát do metody `InputUpdate`.

Metoda Render

Metoda Render se volá v každém snímku. Je možné zde psát kód pro aktualizaci proměnných, ale účel této metody je napsat kód pro vykreslení scény z již vypočítaných hodnot.

V metodě Render se bude volat používání programů, nastavování textur, vykreslování modelů a případě změny nastavení OpenGL.

Ukázkové příklady

V projektu je přednastaveno několik ukázkových příkladů pro ilustraci a jako nápověda při používání nástroje. Příklady jsou vytvořeny jako ukázky, jak psát jednotlivé shadery, jak nastavovat zdroje aplikace, jak implementovat transformační matice a jak používat načtené zdroje v kódu vizualizační smyčky.

Nejjednodušší shader (Simplest Shader)

Nejjednodušší shader ukazuje vykreslení jednoho trojúhelníku. Není implementována žádná animace ani interaktivní funkcionalita.

Základní shader (Basic Shader)

Ukázka nejjednoduššího projektu s vertex shaderem a fragment shaderem. Soubor s globálními proměnnými obsahuje pouze proměnné potřebné pro geometrické transformace a směr světla. Zobrazovaná krychle nepoužívá barvy ani texturu.

Geometry shader (Geometry Shader)

Ukázka použití geometry shaderu. Tento shader pouze demonstruje jednoduché generování dalších trojúhelníků, které jsou posunuté ve směru osy x na obě strany od původního trojúhelníku. Výsledkem jsou tři krychle z původní jedné. Tato ukázka navíc předvádí práci s animací, kde se mění barva pozadí.

Ukázka používá všechny globální proměnné jako předchozí ukázka a navíc se používá textura. Textura se musí v metodě Init svázat s programem, který ji bude používat a musí se nastavit pozice textury pro tento program. Každý program může v jednu chvíli používat pouze omezené množství textur. OpenTK verze 1.1 obsahuje konstanty pro `TextureUnit.Texture0` až do `TextureUnit.Texture31`.

Tessellation shader (Tessellation Shader)

Ukázka TESselačního shaderu. Každou sekundu se mění vnitřní a vnější dělení trojúhelníků. Po zastavení animace (zaškrtávacím políčkem vpravo nahoře) může uživatel zadat svoje vlastní dělení do komponenty PropertyGrid. Ukázka používá stejné globální proměnné jako nejjednodušší shader.

Několik programů (Multiple Programs)

Poslední ukázka předvádí možnost použití několika modelů, několika textur a několika programů najednou. Indexy uniformů každého programu se musí uložit. Pokud budou některé programy používat společné hodnoty proměnných, bude se jim v metodě InputUpdate přiřazovat stejná hodnota.

Jeden z programů používá texturu dřeva na pozici `TextureUnit.Texture1`, jiný program používá texturu dřeva na pozici `TextureUnit.Texture0`. Zde je vidět důvod, proč při používání textur v metodě Render musíme specifikovat, pro který program texturu používáme.