Czech Technical University in Prague

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Bachelor Thesis

Visual development environment for multi-pass rendering

Supervisor: Ing. Ladislav Čmolík
Author: Tomáš Dřínovský

Acknowledgements

At this point I would like to give my thanks to my supervisor Ing. Ladislav Čmolík who guided me through this work, to my parents who supported me for whole my life, to my brother who always helped me and at last I would like to thank to my friends who always cheered me up.

I declare that I have created this thesis on my own and exclusively using literature presented in the references. I agree with lending and publishing this work or its parts with approval of the Department of Computer Graphics and Interaction and I do not have any reservations about using source codes for non-commercial purposes.

25. května 2010 Tomáš Dřínovský

Visual development environment for multi-pass rendering

by

Tomáš Dřínovský

Submitted to the Department of Computer Graphics and Interaction
in partial fulfillment of the requirements for the Bachelor degree.

Abstract

This bachelor thesis deals with the design of a visual representation of the elements of the multi-pass rendering. Next, the software for visual creation of effects of the multi-pass rendering is designed and implemented. This software is designed for the simplest and the most illustrative composition of the passes. At the same time, there is possibility to preview the final effect and to generate a simple code. This code can be use as a base for other applications.

Keywords: shader, multipass, rendering, visual development

Abstrakt

Tato práce se zabývá vytvořením vizuální representace prvků víceprůchodového renderování. Dále jsou navrhnuto a implementováno prostředí pro visuální tvorbu efektů víceprůchodového renderování. Toto prostředí je navrženo pro co nejjednudušší a nejnázornější skládání průchodů. Zároveň je zde možnost výsledný efekt zobrazit a vygenerovat jednoduchý kód, který může sloužit k dalším účelům.

Klíčová slova: shader, multipass, rendering, visual development

Contents

List of Figures

List of Tables

List of Abbreviations

List of used abbreviations in alphabetical order:

DMA	Direct Memory Access
FBO	FrameBuffer Object
GLSL	OpenGL Shading language
GPU	Graphics processing unit
IDE	Integrated Development Environment
OpenGL	Open Graphics library
PBO	Pixel Buffer Object
RGB	Red, green, blue color model

1 Introduction

Many of the objects which are nowadays processed and viewed by computer graphic are in the form of a 3D model. These 3D models are usually represented by the so called boundary representation. The model is not represented by it's volume, but by the surface of the model. The surface of the model is composed from the points, which are called vertices, from the edges created by connection of two vertices and from the surfaces made from these vertices. These surfaces are called faces or polygons.

During rendering the faces are being rasterized. It means the model is transformed from the 3D space into the space of the screen. Next, the face is converted into a set of 2D points in the buffer. These points are called fragments. Once the fragment is processed, it is shown on the screen as the pixel.

More information about boundary representation can be found [3].

Visual effects which originally took some time to process are now rendered in real time using GPUs. Small instruction-restricted programs called shaders are executed on graphics cards and by their combination we can get complex and visually attractive results. These shaders can be the vertex shaders for manipulating with vertices, the fragment shaders for manipulating with fragments and optionally the geometry shaders for manipulating with geometry. By the combination of the shaders it is meant that output of one shader is taken is a input for another shader. That means that the sets of shaders must by applied sequentially. Each step of rendering process when shaders are used to render some 3 dimensional model can be called pass. The Pass provides the abstract wrapping of the set of shaders.

In the last decade the visual development is being introduced in many branches of the graphic software. For example, Autodesk® Softimage® introduced ICE system. This system is used for the visual creation of the dynamic scene properties. Another example is Autodesk® 3ds Max® 2011, which introduced new material editor called Slate. This editor purveys visual development for standard materials as well as for MentalRay materials. The visual develepment in both of these cases increases speed of work and more importantly the lucidity of the solved problem. This thesis is trying to invent similar visual development for the multi-pass rendering.

1.1 Potential of multi-pass rendering

Multi-pass rendering is a rendering technique with lots of possibilities. One pass output is input to the other pass and this way the passes can be chained to create interesting effects.

One of these effects can be, for example, silhouette highlighting. First pass renders the object with one of the standard shading algorithms (for example phong shading algorithm). Second pass renders a mask of the object where the object is rendered in pure white and surroundings in black. The third pass performs edge detection on the mask pass result and combines the result with the standard shading pass result. Illustration of

silhouette highlighting with the results for each pass can be seen in Fig. 1.

Figure 1: Silhouette highlighting example. Image on the left is a rendered model with phong shading. The image in the middle is a flat color render. The image on the right is the final cartoon look created as a combination of the left image and the edge detection from the middle image.

Another example of multi-pass effect is a glow effect. This effect ensures that the selected part of the model glows. First, the part of the model which has to glow is rendered as a mask. This means that the selected part is rendered in white while other parts are rendered in black. In next pass this mask is heavily blurred and applied onto a regular render (for example phong shading algorithm with textures applied).

Another example is a rendering of shadow maps. All of the objects casting shadows are rendered from the position of light. The result of this pass gives the so called shadow map. Another pass renders the object which receives the shadow. Additional UV coordinates are generated and they are used for mapping of the previously generated shadow map. The shadow map is multiplied with the regular render. The practical illustration of shadow mapping can be seen in Fig. 2

All of the mentioned effects were quite simple. Only a small amount of passes were needed to archive the result. But more complicated and powerful effects can be archived if we use more passes.

1.2 Objectives and target user

Primary goal of this work is to design and create software for visual development of multi-pass effects. This program will be called EffectDesigner and this name will be used throughout this text.

Persons this program is aimed to are less skilled shader programmers or people who just want to quickly throw their effect idea into working preview. From this description we can point out two main attributes the work should fulfill: Simplicity and speed of work. Visual development is the tool to accomplish both of these attributes. It's probable the

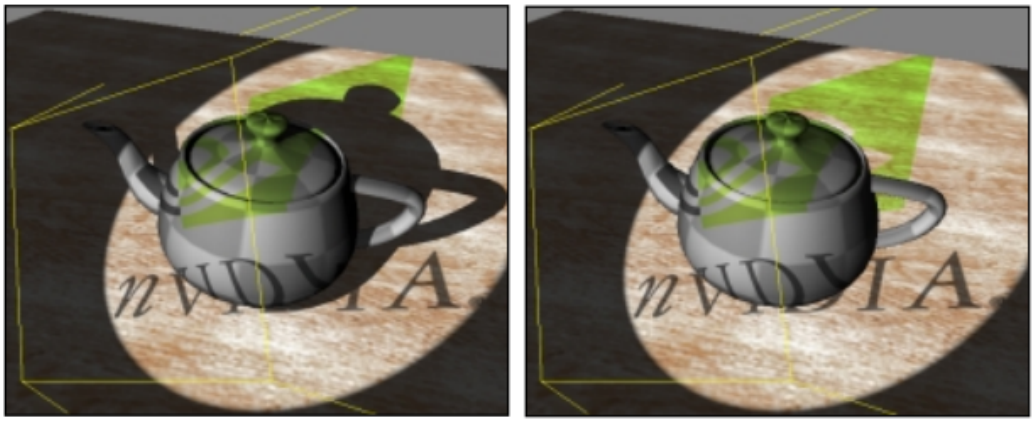

Figure 2: Shadow Mapping example (image is property of NVIDIA Corporation [10]).

users would like to use effects they created in some other application. That's the reason why this work should have feature of code generation. After the user successfully creates an effect, he will be able to generate the code. This code should be well arranged, editable and executable.

1.3 Technology

In this section the technology determined by the assignment is discussed.

The assignment determined that this work has to be written in Java programing language. The other point in the assignment was to use TigerLib. TigerLib is Java library for an easy creation of multi-pass effects and this library uses JOGL for graphic operations. JOGL stands for Java™ Binding for the OpenGL® API and it is a library which calls OpenGL commands from Java.

1.3.1 OpenGL

The Open Graphics Library or OpenGL is popular and widely used application programming interface for the graphic operations which is supported across many platforms. It offers many functions for producing the graphic images, specially those of the 3D objects. Unlike the Microsoft DirectX the OpenGL intermediate only a interface to graphic hardware not interfaces for sound, input and network. The advantage of using the OpenGL is a matchlessly faster render time than a raytracer or a software rendering engine.

From the programmer's point of view the OpenGL offers the set of functions for a specification and a determination of the geometric objects. Furthermore the OpenGL contains the set of functions that control how these objects are rendered into the framebuffer.

OpenGL renders so called primitives. These primitives are points, line segments, polygons or pixel rectangles. The primitives are defined by the group of one or more vertices.

The vertex is a 3-dimensional point with optional attributes such normals, colors and texture coordinates are.

The OpenGL was originally produced by the Slilicon Graphics, Inc. (SGI). Over the time other companies were involved in the development of the OpenGL and now the OpenGL is under patronage of the Khronos Group.

The csecification of the OpenGL are located in [4] and [5] . All the practical informations about the OpenGL can be found [2]

1.3.2 GLSL

OpenGL Shading Language (or GLSL) is set of several languages. These language are used for controlling the behavior of programmable processors contained in the OpenGL processing pipeline. These are vertex, geometry and fragment processors.

GLSL uses simple C-like syntax and supports constructions known from other programing languages as loops, branching and jumps. GLSL offers set of built-in functions for geometric, angle, trigonometry, matrix, vector, texture and other operations.

Specifications of GLSL v1.2 can be found in [2] and specifications of GLSL v4.0 in [2]

1.3.3 Jogl

JOGL purveys Java bindings for OpenGL and thanks to these bindings Java programmers have full access to 3D graphics accelerated by graphics card. JOGL up to version 1.1.1 was oriented on OpenGL 2.1. The development of the OpenGL continued an so did the development of JOGL. JOGL 2 offers system of profiles. The profiles represent abstraction of the original OpenGL version. The profiles allow to maintain compatibility with multiple OpenGL versions. JOGL exists in both 32-bit and 64-bit versions. (For more informations see [1])

1.3.4 Shaders

Shaders are small one-purpose programs executed on graphic hardware. These programs can affect vertices, pixels or the whole geometry of the rendered models. Shaders are tools of the so called programmable function pipeline.

For better understanding how shaders work lets review the situation before they were invented. Before the advent of programmable function pipeline, there was a fixed function pipeline. Schema of the processes in the fixed function pipeline is shown in Fig. 3. All of the vertices and polygons were processed by the same way with only little opportunity to alter this process. Pipeline starts with vertex processing. The inputs of this process are vertices, their normals, colors and UV coordinates. Vertices than have to be transformed to the screen space. It's done by multiplying the vertex positions with the modelview matrix (camera position and orientation) and the projection matrix. The projection can

be orthographic or perspective. Of course, the perspective projection can be altered depending on the FOV parameter. Than the lights are applied. Lighting is one of the most configurable parts of the fixed pipeline. A programmer can specify number of lights, attributes of each light or parameters of the material. Then the texture coordinates are created or transformed by the texture matrix. During clipping phase all of the triangles which are outside of the view are discarded and new vertices on the edge of the view are created. After clipping vertices are sent to the rasterization stage. The output of this process are fragments. After rasterization there are several processes that need to be done. During texturing phase graphic card determines which part of the texture is mapped to a particular fragment according to texture coordinates. Thereafter the color-sum phase just blends two fragment colors into one. If fog is enabled fragment color is modified with a computed fog factor. After that the anti-aliasing is applied and some per fragment operations can be done such as alpha testing (one color is chosen to be transparent), alpha blending, scissor test and others. Result of this pipeline goes to the framebuffer and is rendered on the monitor.

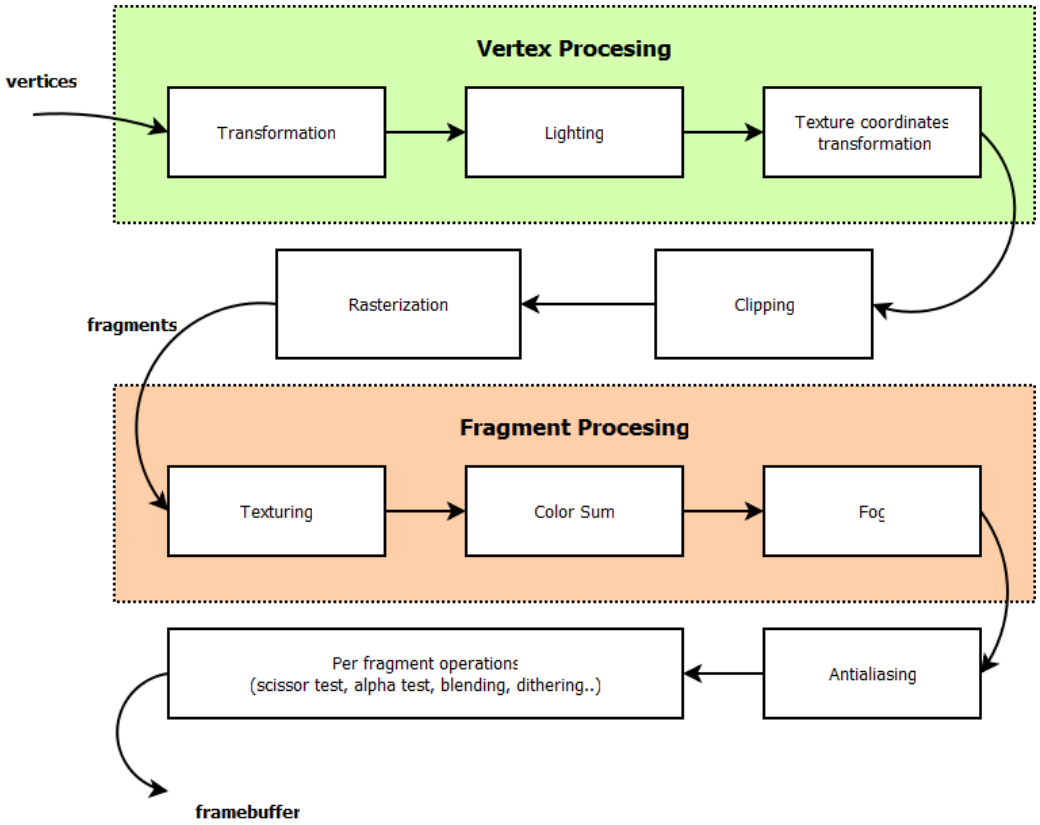

Figure 3: Fixed function pipeline.

In programmable function pipeline many of the fixed pipeline phases were removed and replaced with units for processing shaders. Schema of the programmable function pipeline

can be seen in Fig. 4. User can do whatever he wants and is restricted only by instruction set of the shading language and capabilities of the graphic hardware. All of the function of the fixed function pipeline can be archived with the shaders. Typically in vertex shader the most common inputs are world, view and projection matrices. By multiplying vertices with these matrices we get positions in screen space. The very same process was done in fixed function pipeline. But the shaders can do much more. For example user can add some small random number to the vertex position components and thus create simple noise function.

Fragments shaders are usually used in more interesting way then vertex shaders. For example, user can put several textures as inputs of fragment shader and combine them according to the vertex color. This is simple way to perform multi-texturing, which is beneficial in terrain simulation.

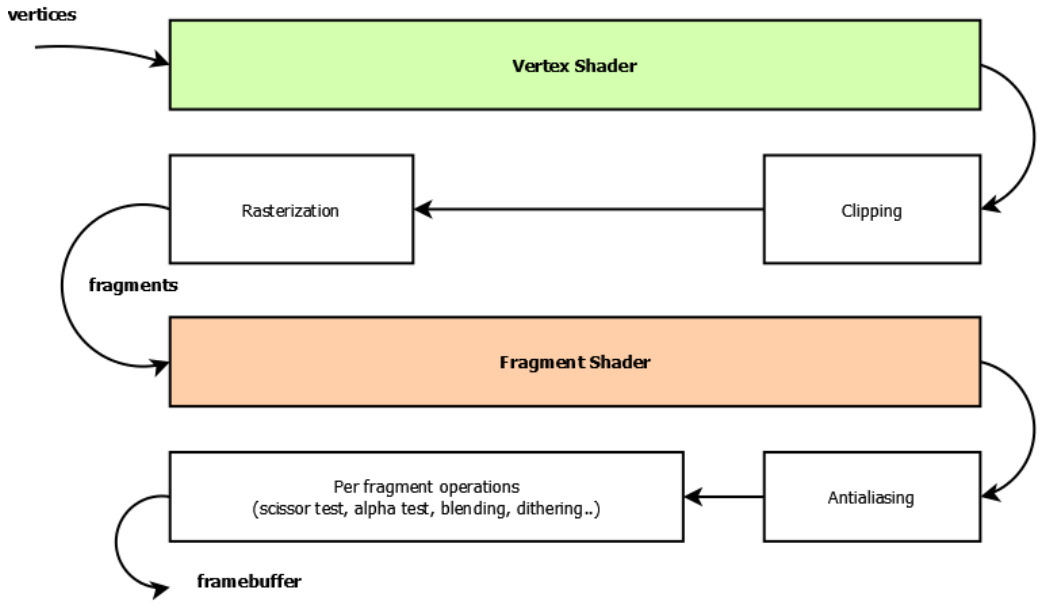

Figure 4: Programable function pipeline.

1.3.5 Multi-pass rendering

Multi-pass rendering can be archived by more than one method. In OpenGL there are two methods: Pixel Buffer Object (PBO) and Framebuffer Object (FBO).

The Pixel Buffer Object is available from OpenGL 2.1 and offers similar handling as the Vertex Buffer Object. The main advantage of PBO is the performance. PBO can be stored on GPU or memory depending on what is more profitable. If the data of the PBO are frequently rendered, the PBO is stored on GPU, and if data of the PBO are frequently written, the PBO is stored in memory. Transfers between GPU controlled

memory and CPU controlled memory are asynchronous. Transfer is done by DMA without CPU involved.

For more information about Pixel Buffer Object see [2].

The framebuffer object is the way how to render the image outside the framebuffer of the window. There are many advantages of using the framebuffer objects. FBOs can have many color attachments and the user can render to any of them directly from the shader. The maximal number of attachments in FBO differs by hardware. The attachments of FBO can have different size than the screen framebuffer, although all of the FBO attechments have to keep same size.

FBO allows to store two types of attachments - Textures and RenderBuffers. Texture can be bound to the shader as a sampler and allows mipmaps, filters and specification of the wrapping mode. RenderBuffers cannot do any of this, but they supports more data formats than Texture.

For more information about Framebuffer object see [2].

1.3.6 Tiger library

Tiger (stands for Toolkit for Interactive Graphics) is Java library made by Ing. Ladislav Čmolík. This library provides abstraction view over multi-pass rendering and uses framebuffer objects.

The core element of the Tiger library is a Pass. The Pass class covers GlslProgram which is a class responsible for linking shaders. The Pass class also keeps lists of parameters, scene, framebuffer, renderState and textures. RenderState is a class which stores many options for OpenGL renderer, for example whether the depth buffer is cleared, the background color and many more. Textures are stored as Texture2D class. This class ensures that each texture gets unique identifier in graphic context.

Besides, the Tiger offers classes for proper loading of OBJ type models, classes realizing Windows with basic navigation etc.

1.4 Content

The following text is divided into 4 main chapters. In Analysis and Design chapter the comparison of existing solution is made. From the analysis of this comparison some desired features are taken and these features are discussed. One of these feature is the visual development. The possibilities how to express the multi-pass effect in visual way are examined as well as the usage of framebuffer objects. Next, the solution for implementation of the user nodes is designed. Other features,Preview of the effect and Code generation, are also discussed. The recommendation for the generated code are determined. In the end of this chapter the program interface is drafted along with the suggestions for controls of the program. In the second chapter all of the implementation details of the features discussed in previous chapter will be examined. First the Visual library is described along with it's advantages for the EffectDesigner. Next, the program is review, all the important classes

of the EffectDesigner are described. The implementation details of parsing of the shader files and schema processing are depicted in detail. At the end of this chapter the format used for saving schemes and user nodes is described with the schematic pictures. Next chapter deals with the testing of the EffectDesigner. The simple user nodes are designed and implemented and by using of these nodes the complex effects are made. In the end, conclusion is made. The work is evaluated and the possibilities for further development are discussed.

2 Analysis and Design

In this chapter existing solutions will be reviewed at first, their advantages and disadvantages will be examined. Their comparison will be made and the conclusion with recommendation for this work will be deduced. After that, individual parts or features of the program will be discussed along with the suggestions how to acomplish them.

2.1 Comparison of existing solutions

There are several widely used environments for shader development. There are AMD Rendermonkey, nVidia FX Composer and Typhoon Labs OpenGL Shader Designer.

For objective comparison of these products with upcoming EffectDesigner we have to determine set of features. These features have to be chosen from the point of view of a target user and his needs.

The target user wants to design not only shaders itself, but also multi-pass effects composed by these shaders. So one of the monitored features will be composition of multi-pass effects. Target user typically wants to quickly and visually compose effects. Also it's handy for the user to edit shaders right in the program. Therefore another monitored features are visual development and shader editing/writing. Since it is not known where exactly would user use finished effects, other features are the support for OpenGL rendering and the support for DirectX rendering. Typically, for OpenGL there are GLSL shaders and for DirectX there are HLSL shaders. User unconditionally needs to see his effect during development. That is why another examined feature is the preview of the effect/shader. The last monitored feature is code generation. Typically the user needs to use created effect in some other application. He needs to have some starting point, which will show him how to implement such effect.

Table showing these mentioned features implemented in particulars programs is in Tab. 1.

2.1.1 AMD RenderMonkey

AMD RenderMonkey (formerly ATI RenderMonkey) is feature rich IDE with agreeable user interface. It supports various shader languages including GLSL, HLSL and others.

RenderMonkey user interface can be seen in Fig. 5. The center of the interface is dedicated for shader text and preview of the effect. On the left side of the interface user can find panel with list of all resources.

AMD RenderMonkey supports creation of multi-pass effects through the system of render targets and renderable textures. User creates renderable texture slot and is allowed to redirect rendering of the pass to this texture. Renderable texture can be used as any other texture. All of the effect assets are located in the tree-view panel on the left side. Choosing render target is made through context menu of the pass item. This approach cannot be called as truly visual development. Shader editing/creating is one of the core

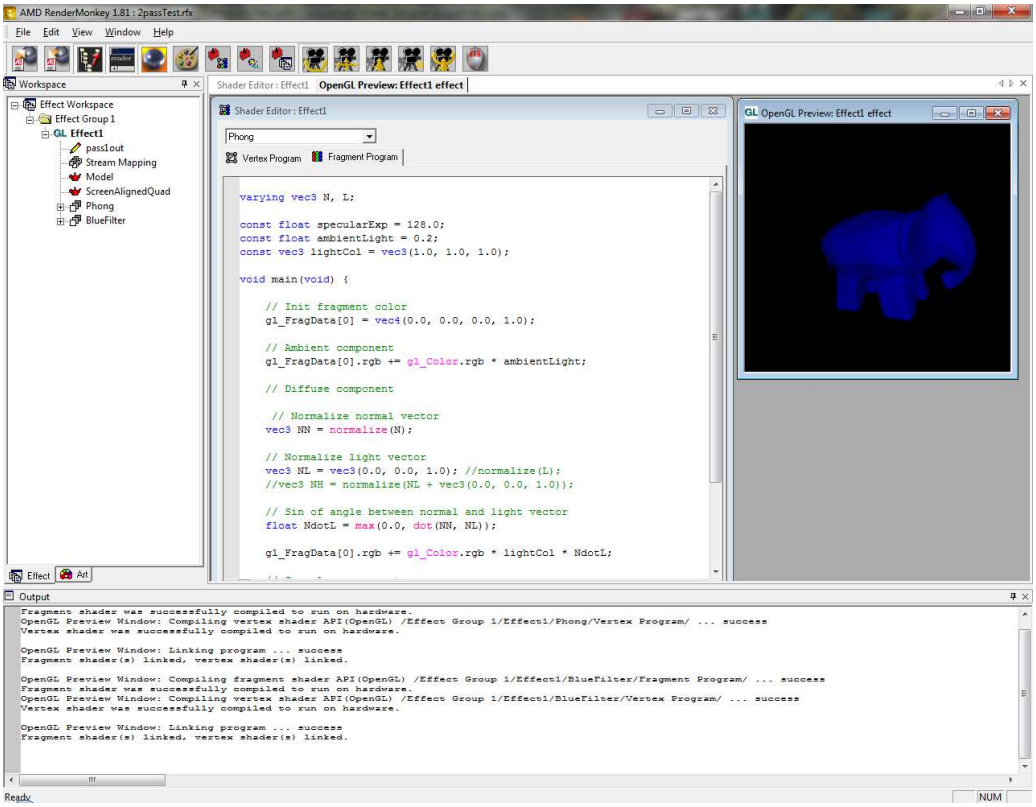

Figure 5: AMD RenderMonkey user interface.

feature of RenderMonkey. By default the text editor for shader is located in the center of the screen and takes most of the screen space. Text editor is equipped with syntax highlighting but unfortunately, there is no auto-complete tool. Compiled effects are shown in preview window. The preview window has many features as navigation, predefined views, display of bounding box or pivot, frame statistics and ability to define background color. RenderMonkey supports both OpenGL and DirectX rendering. For OpenGL IDE offers creation of GLSL shaders and for DirectX HLSL shaders. Despite RenderMonkey doesn't offer any kind of code generation, the user can export the whole effect into collada format. But this solution is not self-executable. User needs external viewer to execute the collada effect.

2.1.2 nVidia FX Composer

NVidia FX Composer is robust IDE with many features. This IDE offers creation of whole effects, materials and composing of the scene. It supports many formats like HLSL fx, Collada fx, CgFx. Unfortunately this program does not allow to write and use GLSL shaders. FX composer offers nice interface to manage lights, textures, models and other

assets of the scene. FX Composer feature rich IDE can be seen in Fig. 6.

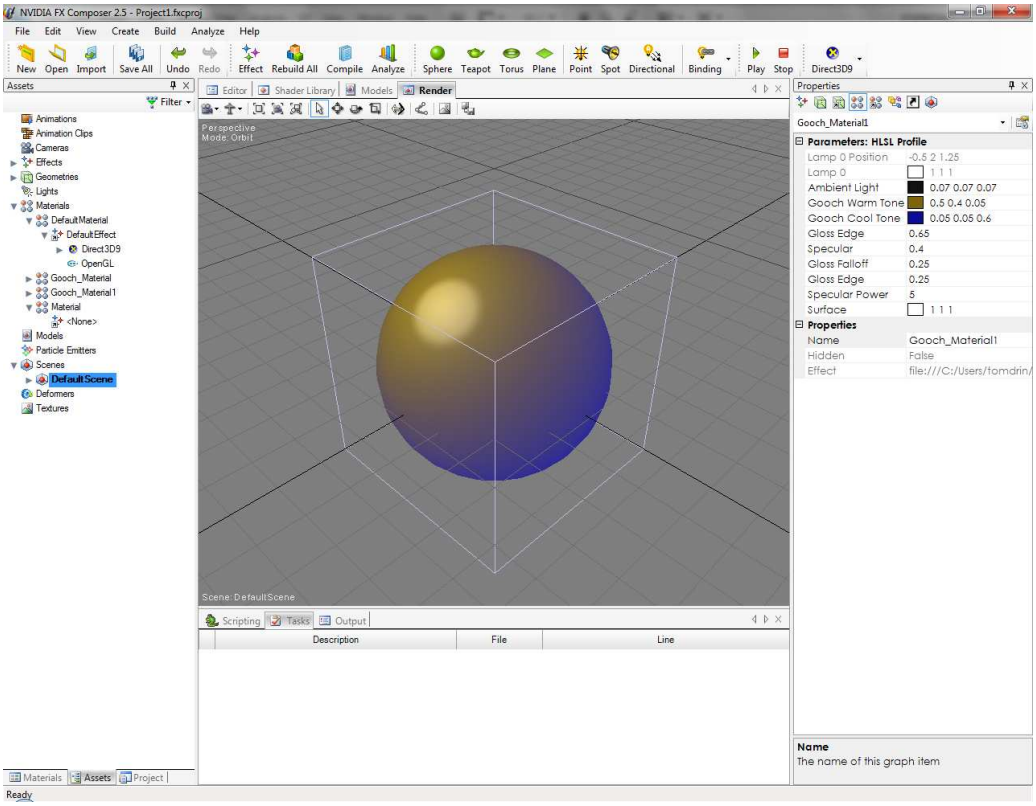

Figure 6: nVidia FX Composer user interface.

FX Composer supports creation of multi-pass effects through CgFx definition. CgFx, contrary to HLSL or GLSL, is language which is able to define multi-pass rendering techniques. There are ways how to define attributes of the shaders or texture paths. Multi-pass techniques are written in CgFX effect by the user. There is no visual development of the effect. The core feature of FX Composer is the text editor for shader development. This editor has syntax highlighting and also offers limited auto-complete tool. As mentioned before FX Composer doesn't support GLSL shaders and OpenGL. FX Composer is based on DirectX renderer and has brilliant support for its shader technology. As well as other solutions FX Composer has effect preview window. Preview window in this IDE is loaded with many features. User can add many models, attach different materials to them and place them arbitrarily to the scene. User can even place various lights to the scene. Unfortunately, there is no feature which can be called code generation.

2.1.3 Typhoon Labs Shader Designer

Shader Designer is simple and well arranged IDE. The user interface is to be found in Fig. 7. On the left side of the screen there are shader render window and texture preview window. The center of the screen is occupied by the text shader editor. It has also nice front-end for modifying common parameters as are light parameters located on the right side of the screen.

Typhoon Labs OpenGL Shader Designer was one of the first GLSL shader editors. Unfortunately the development of this tool stopped in year 2004 with version 1.5. That is the reason why new features of GLSL language or geometry shaders are not supported at all and in comparison to other solutions the Shader Designer offers less features.

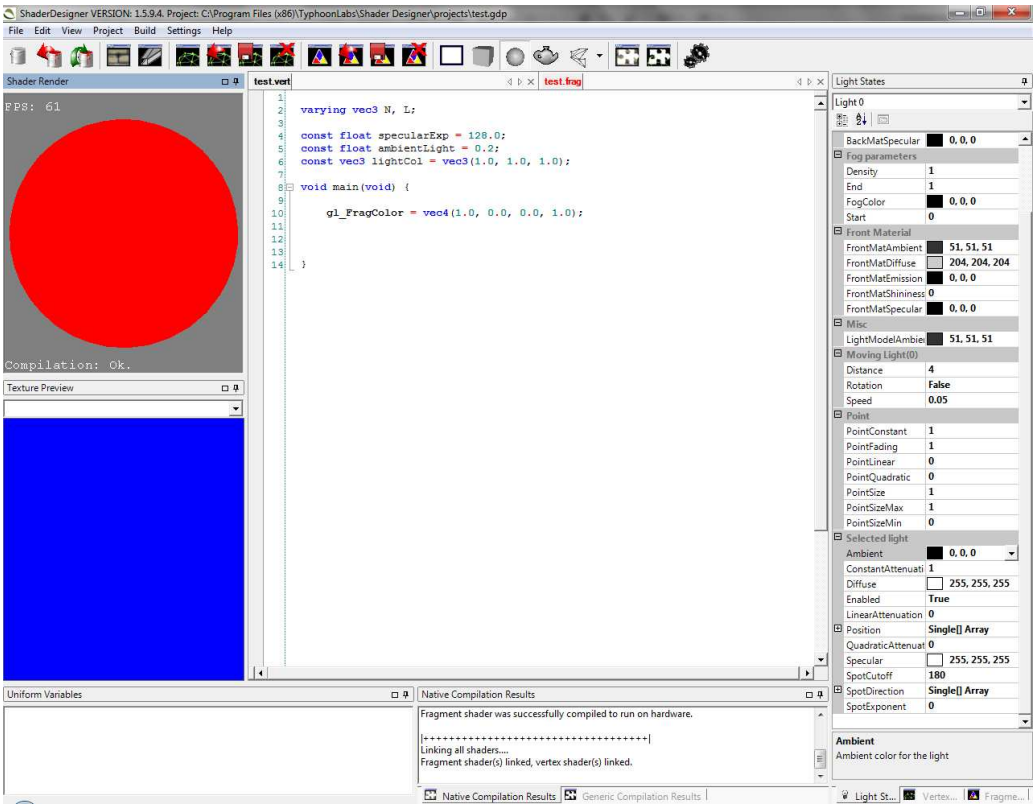

Figure 7: Typhoon Labs OpenGL Shader Designer user interface.

Shader designer is simple IDE. It's downside is that there is no support for multi-pass rendering. Therefore, there is no visual development of effects either. The main and only purpose of the ShaderDesigner is to write GLSL shaders. For this purpose there is a nice text editor with syntax highlighting. The text editor offers limited auto-complete tool. As mentioned above, the Shader Designer allows to create GLSL shaders only. Therefore, only OpenGL rendering is supported and DirectX is not. The preview of the shader is

present with basic scene navigation handling. This IDE does not have any kind of code generation or even export implemented.

2.1.4 Recommendation for EffectDesigner

All of the programs mentioned above provide tools for developing shaders. Some of them even provide tools for combining shaders into complicated effects. But all of them are text IDEs only. There is no truly visual IDE. It is because the main purpose of these IDE is to create shaders not effects.

The result of this bachelor thesis, as distinct from programs mentioned above, is centered around design of multi-pass effects. That is why it is not planned to implement text editor for shader writing. The user can use traditional tools like notepad or VIM or he can use one of the competitive IDE.

Due to the assignments the EffectDesigner will offer no support for DirectX rendering. This environment will use only OpenGL renderer and therefore the passes have to be formed from GLSL shaders.

Other planned feature is to offer some way how to pack whole effect into compilable code. None of the programs mentioned above have similar feature. This particular function is handy for people who want to quickly use their effect. Generated code could be part of the presentation or can be modified to more complex visualization.

2.1.5 Evaluation of comparison

Table 1: Comparison of monitored features.

Feature	RM[1]	FXC[2]	SD[3]	ED[4]
composing of multi-pass effects	●	●	○	●
shader editing	●	●	●	○
visual development	○	○	○	●
support for OpenGL rendering	●	○	●	●
support for DirectX rendering	●	●	○	○
effect preview	●	●	●	●
code generation	○	○	○	●

[1] AMD RenderMonkey

[2] nVidia FX Composer

[3] Typhoon Labs OpenGL Shader Designer

[4] Planned features of EffectDesigner

As seen in Tab. 1, EffectDesigner will have two features which are not present in any other aforementioned program. These features are visual development of multi-pass

effects and code generation. These features are key in the design and development of Effect Designer. Main downside of EffectDesigner is that it is not going to have any text editor for shader creation.

If the user feels comfortable composing multi-pass effects by the non-visual way the best solution for DirectX rendering is the nVidia FX Composer and for OpenGL rendering the AMD RenderMonkey. The Typhoon Labs OpenGL Shader Designer suffers from the long time out of the development. Offers only basic functionality and is not able to produce multi-pass effects.

2.2 Visual Development

The Key feature of the Effect Designer is visual development. The visual development means that the effect is represented by various blocks and these blocks are connected to create the effect logic.

Multi-pass effects are composed from several passes. Hence, it is essential that the core elements of the effect schema are pass nodes. The pass node is a block which contains references on vertex, fragment and geometry shaders.

Shaders usually have inputs. For example light direction vector or position, camera position, color and others. Basic data types which GLSL shader allows are booleans, floats, integers. These types can be also wrapped in vectors (with dimensions of 2,3 and 4) and matrices (with dimensions of 2x2 up to 4x4). GLSL data types are defined in [6]. Since there is a limited number of data types which can be used as inputs, it is possible to represented these data types as nodes too. Most used data type in shader programing are floats. Others single data types can be substituted by float. Therefore other important nodes are float node, float vectors and float matrices.

Other type of inputs GLSL shaders are allowed to use are samplers. The sampler is a data type which handles texture and it's filtering. The samplers can be one, two or three dimensional. Also in GLSL version 4.0 there are special versions which are using integer and unsigned integer as base. Since textures are very important part of the multi-pass effect creation, it is necessary that EffectDesigner contains texture node. The most common ones are simple 2D images, which can be used as sampler2D inputs for shaders.

Complicated operations have to be always written in shader itself. Nonetheless, there are some basic operations which should be provided as nodes. These operations are fragment multiplication, addition, inversion etc. Other useful operations could be edge detection and blur. These nodes can be defined using user node system. This feature is discussed later in this work.

2.3 Framebuffer Objects in Effect Designer

As mentioned before this bachelor thesis uses TigerLib. TigerLib is Java library for easy creation of multi-pass effects. Internally, TigerLib uses framebuffer objects for storing textures.

Each pass is either rendered on screen or into the framebuffer object. The framebuffer objects can be shared among passes. Since there is no explicit number of maximum FBOs there are no problems with creation of many of them and the only limitation is the size and number of textures on graphic card.

The first way is to display the framebuffer object as a regular node. The advantage of this approach is that the user can create the effect with less framebuffer objects and do the optimization by himself. The disadvantage is that the user can read and render into the same framebuffer object and into the same texture. The result of this operation is not specified by OpenGL specifications and thus it may differ depending on the hardware.

The second approach is to provide some way how to choose the framebuffer object in the pass node. This solution needs to add some form which will be used for managing framebuffers. As in the previous case the user can create effect with less framebuffer objects and do the optimization. The Disadvantage is the same as in the previous solution. The user can read and render into the same framebuffer object and into the same texture. The draft of this solution can be seen in Fig. 8.

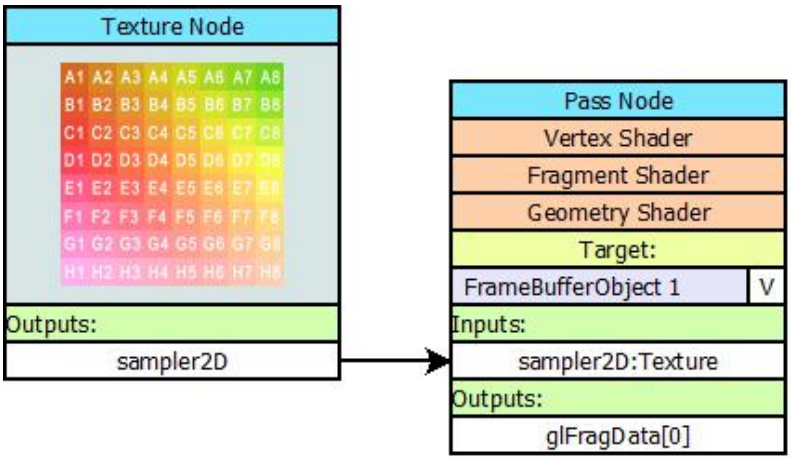

Figure 8: Draft of the Pass Node interface with a selection of the target.

The last option is to hide the framebuffer objects from the user completely. The framebuffer objects are created for each pass. Hence, there is no possibility of overwriting the read buffer. This solution does not leave space for optimization. The advantage is that the user is not able to render the pass into the framebuffer, which is used as an input. Since there is no way how to determine the target framebuffer object, it is necessary to provide some way how to render into the screen framebuffer. This solution adds new kind of node - screen node. There can be only one screen node.

The last described solution is the best way how to achieve simple and fast workflow of the effect creation. The user is allowed to do only the minimal number of operations with minimal chance to make a mistake.

2.4 User nodes

One of the points in the assignment was to provide some kind of ability to define new operation blocks. The user should be able to use not only the predefined blocks, but also make his own. There are several ways how to make features like this, each with a different impact on the user.

Comparison of user block solutions

The first way lets user to create user block in the same IDE with the same tools used for the creation of a regular effect. The user block is sort of an incomplete effect. Some of the unconnected inputs and outputs are then pronounced to be inputs and outputs of this user block. This solution has several advantages and disadvantages. One of the advantage is certainly the easiness of the creation. Once the user gets used to using main IDE, he gets ready to create his own user nodes. User does not need to learn some new tool or programing language. Also the implementation of this solution is more simple than implementation of the other solutions. The user blocks are converted into a set of regular blocks before the processing of the effect. The disadvantage of this way of creating user blocks is a limited ability to create new functionality. The user can create only the user nodes, which are made by regular blocks. It's questionable if this is a small or a big disadvantage.

The second way is to create a plugin system and let the user to program the user nodes as plugins. This solution creates additional requests to the implementation side of the work. The classes need to be ready for this approach and the whole process of parsing blocks into executable effect needs to be programmed more complexly. Another disadvantage of this approach are the requirements which are requested from the user. The plugins will have to be created in Java or in some other language. On the other hand, one of the biggest advantage of this way is the ability to define more complex situations. The user will be able to create loops, branches and other constructions known from programing languages.

The third way is to develop some kind of scripting language. The user nodes are defined in a script file. This script file is parsed and performed in time of processing the effect. This solution is similar to the previous one, but there is no need for external language such as Java. Scripts can be very simple with a small instruction set. The user can write scripts in any type of text editor. This solution offers less amount of freedom than the second one, but the advantage is that the scripting language could be very simple and easy to learn. The implementation of such kind of scripting language brings similar demands as plugin solution. In addition to that, there would be need to create effective parser for these scripts.

To fairly judge what solution is more appropriate for this work it is necessary to consider how much possibilities the users need in an effect design. The shaders and multi-pass effects are tools of the real time rendering. Every pass means a significant drop in

frame rate. That is the reason why every single pass must be justified. In most cases there is no need to repeat the same pass over and over in loop. There are special cases but even they are repeated only several times (less than ten times). Loops are superfluous feature. Branching is also redundant. Output of passes are buffers or textures. It's very slow operation to withdraw buffer and analyze it. Therefore, it makes sense to use as a branch condition solely numerical values. The most of the time user knows the values of these variables before processing. They are color, float or boolean inputs of the shaders. It is only logical for user to make an effect or a user node for one branch and make another one for the other branch of the intended schema. For pass combination there is no need for loops or branches. They are redundant. If the user needs branching based on texture, he can do such thing in shader.

Since the main concept of this work is to create IDE for fast and intuitive multi-pass effect creation, the advantages of the first solution are most valuable. The user can create effect and user block in the same program, using the same blocks in a the same way.

External files used in user blocks

When the user is creating custom nodes he may need to use some external files. For example, if the user uses texture node during creation of custom node the requirement for the picture arises. Since the custom node will be made in a same way as a regular effect, the custom node schema should be saved in one file with similar format as a regular effect format. The external files could be stored inside or outside the custom node file. Both of these solution have their advantages and weaknesses.

If the external files are stored outside the custom node, the user blocks are less transferable. The paths to the external files can be stored in both the absolute or the relative format, but once the file with the custom node is transfered, there is no guarantee the external files will be located on same path location or even exist. On the other hand having external files stored separately means that the user can change the file and thereby easily change the behavior of the user node.

The second option is to have the external files incorporated into the file with the custom node. This option does not offer such comfort in changing the custom node behavior. It is questionable whether such feature is needed. If it is probable that some of the images will be changed, then such images should be marked as inputs of the custom node. The advantage of this solution is that the user blocks are fully transferable. The user can spread the user blocks on other computers and does not need to worry about related files.

Since the custom nodes will be used for the extension of the EffectDesigner functionality the mobility is the key feature here. For this bachelor work the more suitable option is to have the external files incorporated into the file with the custom node. The reason is that the users can easily interchange custom nodes if all of the data is located in a single file.

2.5 Preview of the effect

The preview of the effect is very important feature, which is inexpendable in shader development IDE. The concept of the preview is dependent on the way how the framebuffer objects are selected. From programing point of view the framebuffer objects and the screen are similar, since the screen is represented by it's own framebuffer.

If the framebuffer objects are selected right at the pass nodes the option to render to the screen is presented among list of available framebuffer objects.

If framebuffer objects are hidden from the user and they are created for each pass there must be some way how to determine what will be rendered into the screen. This could be done by adding a special node to the scheme which will represent the screen. Since the hidden framebuffer objects are the best solution for the EffectDesigner (as described above) it is necessary to implement the second solution with the screen node. A sketch of the scheme with the screen node can be seen in Fig. 9.

There is also a posibillity to let the user preview the content of the buffers from FBOs at some step of the rendering process. This feature would be very beneficial for the debugging of the effect. This feature needs more complicated control mechanisms. Usually the output of the fragment buffer is one texture accessible through glFragColor. But in some cases it is necessary to use more then one output texture. This outputs are then accessible through the glFragData array. If the preview of the content of the framebuffer object is presented it is necessary to determine which texture is viewed. The sketch of the pass node interface which provides mechanism for choosing the viewed texture can be seen in Fig. 10. This control mechanism will complicate the node interface and such convolution is against the concept of simple and clear workflow. That is why this concept has been abandoned.

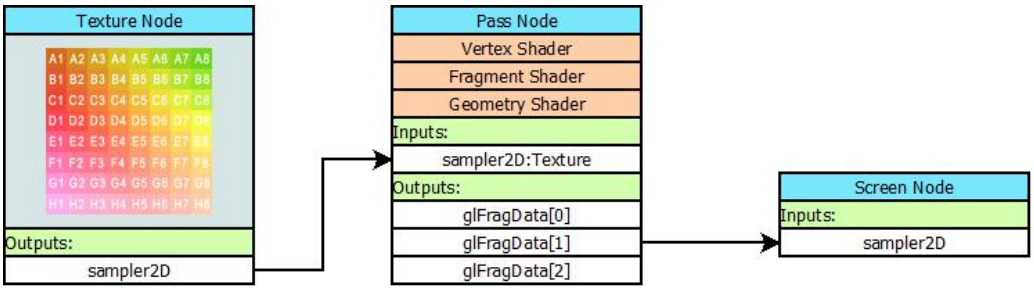

Figure 9: Draft of the effect scheme with screen node mechanism.

2.6 Code generation

One of the features which makes EffectDesigner special among other IDEs is a code generation. Since the program is using TigerLib and JOGL, we can assume the user will

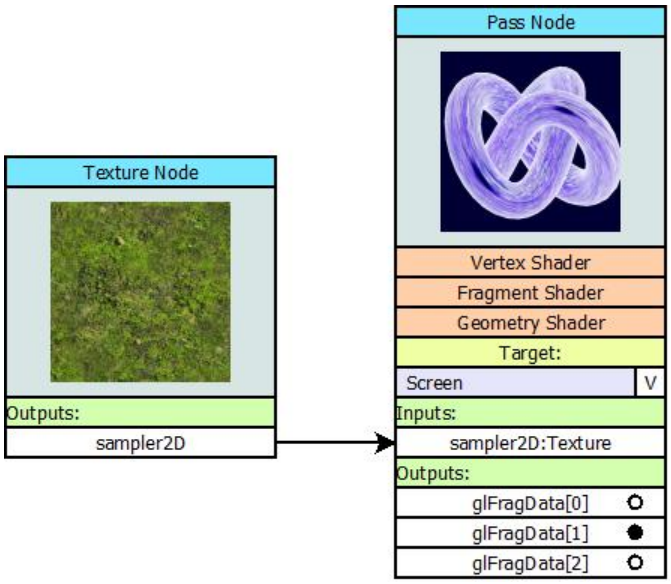

Figure 10: Draft of the pass node interface with buffer preview.

have access to these libraries too. This presumption allows to use these libraries in the generated code. The code should have strictly separated sections - section for variables, section for constructor operations and section for render loop operations. Such segregation will result in a simple and well arranged code.

The result of the code should be program with one main window. This window will show the desired effect and will also have a simple navigation providing, for example, rotations of the model.

2.7 Program interface

The result of this bachelor thesis called Effect Designer is to be used by various users. In spite of that, target users are usually tech-artists and effect programmers. As mentioned before the program has to provide simple and effective interface for quick work. The interface should be clear and simple, with minimum of buttons.

The most spacious area of the EffectDesigner has to be the schema viewport. This area has to support moving and zooming since some of the effects can be very complex and they could exceed space determined by the viewport resolution.

There are many types of the nodes which can be used in the effect scheme. The interface of the program has to provide some type of menu where the user can choose the node which is to be added into the scheme. One of the solutions is to have a special panel which will be visible all the time. This panel contains icons, which represent nodes for the creation. The buttons or the icons should be draggable into the schema viewport. The mentioned solution takes a lot of space and this space has to be taken from the

viewport. On the other hand, user can see all of the available nodes for the whole time of the development and so is able to think about all the possibilities how to achieve a desired effect. Also the different icon for each type of the node makes the node buttons more distinguishable.

Other solution is to offer all the nodes through the context menu. The user uses a right-click on the viewport and the menu with the nodes shown. This solution is space effective. The space on the screen stays for the scheme viewport. The disadvantage is that the types of nodes are hidden by default and the user is not able to see all the possibilities he has. In this regard it would be more comfortable for the user to devote special space on the screen to a panel which would contain icons.

It is clear the program will have to show messages about the status of an effect generation. For example, warnings should be shown when there are some unused nodes in the schema and errors when the effect generation fails. This means the interface has to have some kind of a text output frame. The location of this frame must not take space and attention from the schema viewport. The ideal location is the bottom of the screen.

Other than that there is needed only one button. This button will serve for triggering the effect generation process. Since it is the only button on the screen and it's significance is great, the button can be big and visible.

The user is used to the locations of some of the functions, such as Save and Load, New and Close. These functions are usually formed in the top menu. Since the idea of the EffectDesigner is to not make the user confused it is suitable to locate these functions in the top menu too.

The draft of the user interface of EffectDesigner can be seen in Fig. 11.

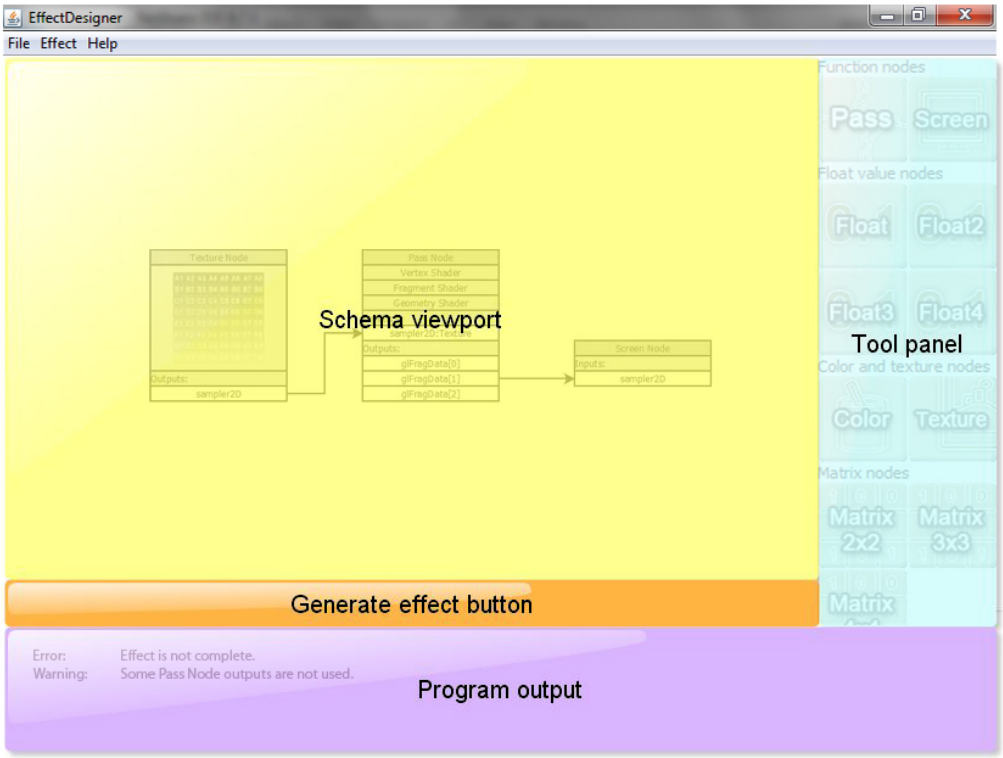

Figure 11: Sketch of the interface areas.

3 Implementation

This chapter describes the technical solution of each individual piece of the created program. At first, the overview of the EffectDesigner classes will be made and then some of the interesting program parts will be mentioned and discussed.

3.1 Visual Library

Visual library is part of the NetBeans API. More precisely Visual Library 2.0 is a part of the NetBeans Platform 6.0. This library offers simple workflow for creating graphics and their controls. The core class is the Scene class. This class holds all the visual data of the scene. The visual data is represented as widgets. The widgets are primitive visual elements. These elements hold information about its location, boundary, preferred location/boundary, preferred/minimal/maximal sizes, layout, border, foreground, background, font, cursor, tooltip, accessible context and many others parameters. It is possible to say the widgets are adequate replacement for JComponent class. Each widgets can hold another widgets and so they can create hierarchical structure. The visual Library allows to simply add the so called actions to the widgets. These action replace the behavior of the classical JComponent Action Listeners. There are various actions, from the basic such as the hover action to more advanced ones as the move action, the popUpMenu action and many more. The Visual Library is able to use the Java Swing components in a widget. This is done by the special ComponentWidget. This solution brings one disadvantage. Swing components are drawn separately from widgets. This could lead to overlaping. Also some of the features, for example zooming, are not available for the Swing components.

For more information about the Visual Library see it's documentation [9].

For the purposes of the EffectDesigner it was needed to use some library for easy visualization of graphs, nodes and connections between these nodes. The Visual Library and it's vmd package have a great set of tools for this purpose. The VMDNodeWidgets from the vmd package has a unique visual style and many abilities which are useful, for example, ability to minimize the node. The VMDConnectionWidget from the vmd package is a widget which servea as a visual connection between two arbitrary widgets. The route of such a connection can be straight or with angles.

3.2 Program overview

The program is designed to be a visual editor in the first place. That means the majority of the classes are made to support the visual development. The overview of the EffectDesigner classes can be seen in Fig. 12.

The Designer class is the main class of the EffectDesigner. The Designer inherits from the JFrame class and so it handles the creation of all the static interface of the main window. The side panel with the nodes buttons, the output text area, the control buttons and the menu – all is created and placed as was designed in the previous chapter

of this work. For positioning the EffectDesigner uses mostly the BorderLayout and the GridBagLayout. The BorderLayout is a simple layout where the area is divided into 5 sections – 1 for each side and one for the center. When the size of the window is increased the center area gets additional space. The GridBagLayout in opposite to the BorderLayout is a very complex and adjustable layout. Each component is placed in a layout with informations about the location in the grid, the weight of the component and the size. These informations are passed as the GridBagConstraint object. The Designer class contains other EffectDesigner classes, such as the GraphScene, the EffectGenerator and is responsible for their handling and cooperation.

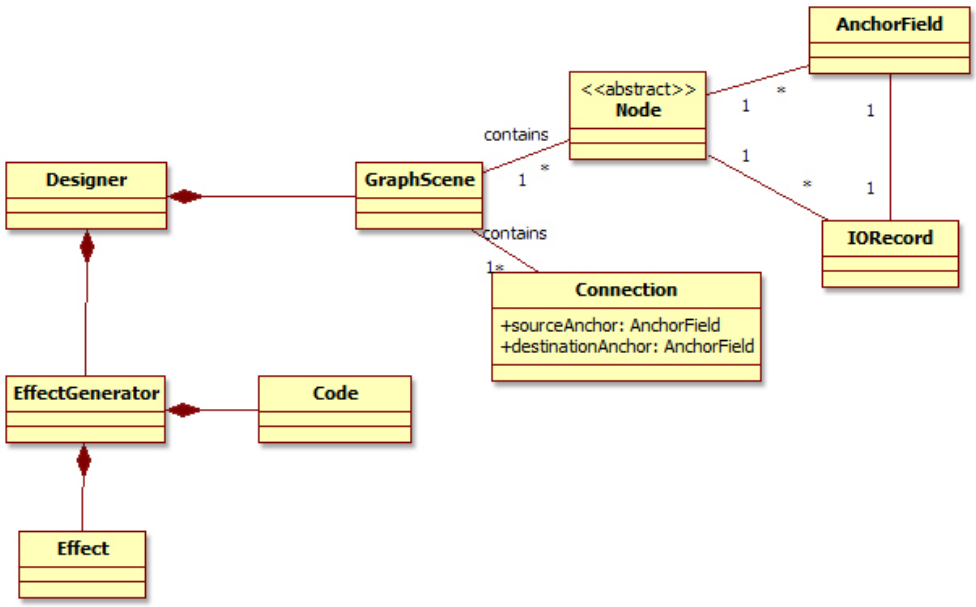

Figure 12: Overview of the EffectDesigner classes.

The GraphScene class is a child of the Visual Library Scene class.This class purveys several methods for handling nodes and connections. Since neither the scene nor the widgets are descendant of the JComponent they cannot be added directly into the layout. The scene class provides the createView() method which handles this problem completely. This method returns a JComponent object.

As mentioned before, the Scene class holds up the visual data represented by the widgets. The widgets can be laid on the layers by using the LayerWidget. The GraphScene has two layers. One layer serves for storing nodes and another layer serves for connections. Since the GraphScene uses this layer system it is necessary to add new methods for handling and managing the nodes and the connections.

3.3 Components of the schema

The Node class is a class inherited from the VMDNodeWidget and serves as a graphic representation of the node. Each type of node is an independent class inherited from the parent Node class. The overview of the Node class and it's descendants can be seen in Fig. 13. All of the nodes create the so called Properties Window stored in the propWindow variable. This window as available through the context menu and provides option for renaming the node. The Node class purveys the methods for parsing and storing basic data from and to an XML file. This data contains location, name and id. The Node class also keeps the list of inputs and outputs of the node in a form of an ArrayList¡IORecord¿. The IORecord is a class which is composited of an input or output variable name, a variable type, a parental node and optionally a shader type. Alongside with the IORecord there is an AnchorField class. The AnchorField objects are widgets which graphically represent the inputs or the outputs of the block. This separation is necessary because of the method of solution of a graphical connection between the nodes. The Pass node is the key type of a node and thus it has to offer extended functionality. It is the only node type with variable inputs and outputs. Once the user chooses a shader file, the file is parsed using regular expressions. The regular expressions can be used because the variables used for the inputs have the definition in the format *uniform variableType variableName* or *attribute variableType variableName*. The regular expression capable of catching these lines is:

```
(uniform|attribute)\\s+(\\w+)\\s+([^;]+)\\s*;
```

The outputs can be only two. The first one is the glFragColor, the second one is the glFragData[] array. Once the glFragData[] is used it is necessary to create as many node outputs as the highest index used in glFragData[]. The regular expressions used for this purpose are:

```
gl_FragData\\[(\\d+)\\].*;
gl_FragColor.*;
```

The inputs and the outputs are found and the corresponding anchors are created.

The pass node creates a special version of Properties Window. This PassNodePropertiesWindow adds ability to change the rendered model. This model can be either a user defined obj model or a screen aligned quad. Another choice is to use some of the prepared models distributed with the EffectDesigner. These models are a cube, a sphere, a knot and a teapot. The pass node also supports different RenderState settings, therefore the JPanel with a unique interface adjusted for each renderState setting is created.

The texture node class in addition to the Node class contains an ImageWidget. This widget is used for displaying a preview of the texture. The texture node has only one output and it is the output of the sampler2D type.

The screen node class is used for terminating the effect. The most of the time the user wants to display an effect on the screen and this type of node is a way how to achieve

it. The screen node is created with only one input of the sampler2D type and with no outputs.

The FloatValueNode class and it's versions are used for passing the numerical values into the shader. These nodes are created with one or more fields for entering the numbers. These fields are realized through the JTextField class and they are resistant against entering a non-numerical value.

The Matrix2Node class and it's versions are created the same way as the FloatValueNode. The fields for entering numerical values are placed according to the dimensions of the matrix.

The ColorNode class is a special node for vec3 input type. The interface of this nodes is equipped with a ImageWidget which shows the actual color. If the user clicks on this ImageWidget the JColorChooser Dialog is shown. Since the GLSL does not have a variable of type color, it can be substituted for vec3 type.

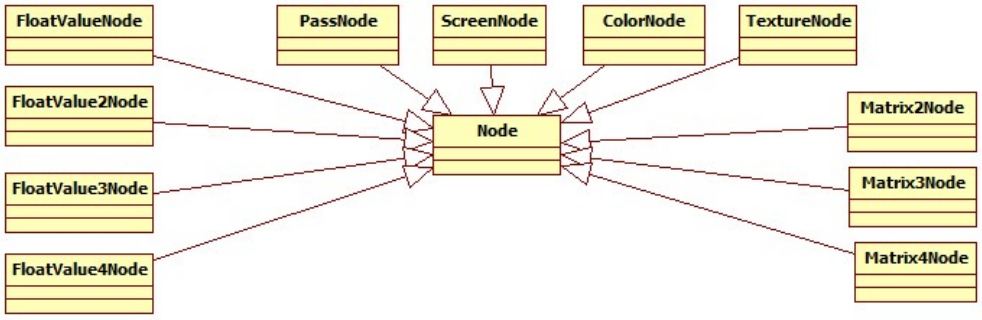

Figure 13: Overview of Node classes.

The nodes, respectively the inputs and the outputs of a node need to be connected to create the effect schema. The connections are realized through the VMDConnectionWidget. For effect generation the VMDConnectionWidget is transformed into the Connection class.

3.4 RenderStates

The RenderState is a TigerLib class which describes some options of how the OpenGL renderer should behave. Since there are many of these options and they have different values (boolean, floats etc.), it was necessary to invent some way how to generalize their settings.

This generalization is provided by the IRenderState interface and the renderStateAlternators, which are classes implementing this interface.

Each pass node holds a list of the renderStateAlternators. This list is filled once the pass node is created. The class PassNodePropertiesWindow uses this list for generation

of its visual interface.

Once the effect is being generated and a pass node comes on the queue, the IRenderState method *alterRenderState(RenderState rs)* is called by all of the elements in the list of renderStateAlternators. This method ensure that the inputed RenderState gets changed according to the pass node settings. After that, the method *alterRenderState(String name)* is called. This method ensures that the code for altering the RenderState is generated.

Although many of the RenderState options are just in form of an enable/disable option, there are also some non-boolean options, for example clear color. This option sets the basic color of the buffer, i.e. the color which is used for clearing the buffer. This diversity of setting needs also a diversity of control interface. That is why the IRenderState provides the method *createUI(JComponent parent)*. This method creates the user interface as JPanel. This JPanel is adjusted individually for each RenderState setting. For example, if it is the color settings, then the JColorChooser is added to the interface. If the setting requires boolean value, then the interface is a set of radio buttons. The created JPanel is then added into the parent component. Sometimes, it is profitable to not change the RenderState and leave it as it was with the previous pass. That is why each implementation of *createUI(JComponent parent)* method should add an option to "don't change selected" option. This option should be prescribed as default.

The IRenderState interface also provides methods for saving and loading values from the file. These methods are the *printRenderState(BufferedWriter out)* and the *parseRenderState(Node node)*.

3.5 Effect handling

The class Effect is the only class which directly uses TigerLib and JOGL. The JOGL handles the rendering through the GLEventListener interface. This interface has 4 main methods for rendering. In the *init(GLAutoDrawable drawable)* method all of the assets have to be initialized including the framebuffers objects, the texture objects and the passes. The Init is called only once, when the effect is activated. The *reshape(GLAutoDrawable drawable, int x, int y, int width, int height)* serves for adjusting the framebuffer objects and it's textures to the screen resolution. This method is automatically called when the size of the target viewport is changed. The *display(GLAutoDrawable drawable)* method ensures the rendering itself and so it is called repeatedly in a loop. The last method is *displayChanged(GLAutoDrawable arg0, boolean arg1, boolean arg2)*. This method has no functionality in the Effect class.

The Effect class has to keep lists of all the passes, textures and framebuffers. These lists are realized through the ArrayList class. The Effect class is working on a simple concept. The first pass is added by the *addPass(Pass pass)* method. This pass is added to the list of passes and is also stored in the currentPass variable. Then all of the assets are applied to this pass by the *addFloatParameterToCurrentPass(float[] values, String name)*, *addFloatMatrixParameterToCurrentPass(float[] values, String name)*, *addRenderStateToCurrentPass(RenderState rs)*, *addFrameBufferToCurrentPass(FrameBuffer frameBuffer)*, *addTex-*

tureToCurrentPass(Texture2D texture, String name) methods.

Diagram of the Effect class can be found in Fig. 14.

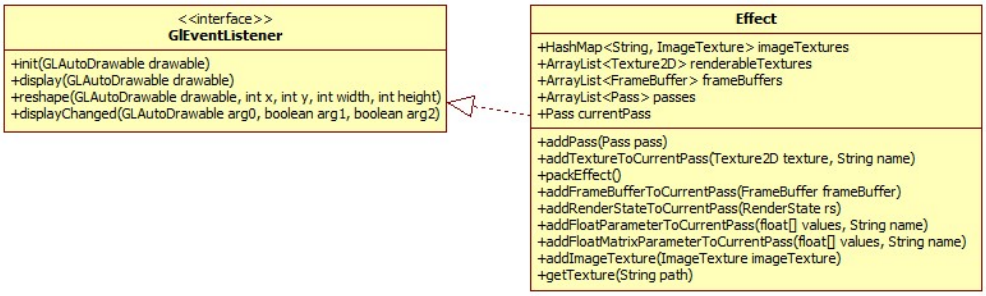

Figure 14: Diagram of the Effect class.

3.6 Effect generator

The Effect generator is a complex class. It's purpose is to convert a set of visual nodes and connections into a renderable effect. Once the user executes the function under the Generate Effect button the Designer class will retrieve a list of nodes and a list of connections and they are used as inputs for the*generateEffect(ArrayList¡Node¿ nodes, ArrayList¡Connection¿ connections)* method. At first this method creates the Effect object and the default RenderState object. Next, the ImageTexture object is created for each Texture node.

The ImageTexture is a class responsible for loading the actual image data. The image is loaded into the BufferedImage class by the *ImageIO.read(File file)* method. This method gives an easy access to the standard image formats such as jpeg, png, gif and others. Unfortunately the tga format is not supported by default. The image is then converted so as it's dimensions are powers of two and it is square shaped. This conversion ensures that the Effect will be renderable on larger spectrum of graphic hardware. During the Effect init method, the ImageTexture's init method is called. This method creates a texture object for the texture and copies data from the BufferedImage into a memory of a graphic card.

In the next step the generateEffect method works in a loop until all of the shader nodes which are able to be processed are actually processed. Each cycle of the loop the private method *getNextShaderNode()* is called. This method returns yet unprocessed node which is ready. It means that the node inputs are connected only to the variable nodes such as Texture nodes or the FloatValue nodes or they are connected to the already processed pass nodes. In order to keep track of the already processed nodes the new structure has been established – ProcessedPassRecord. The first thing done once the pass for the processing is selected is a check of the inputs. It is not possible to continue the work with

the pass unless all of the inputs are defined. Otherwise the error message is shown and the processing is canceled. If the inputs are connected default RenderState is changed by using all of the pass node's renderStateAlternators. All of the paths of shader files are retrieved and the TigerLib Pass object or the Saq object is created using these paths as input. This depends on the scene option of the PassNode. If the scene is chosen then the scene is loaded by the ObjLoader class and inserted into the Pass. If not then the Saq object is created instead of the regular Pass object. This pass is added to both the Effect and the Code objects.

For each input of the PassNode the matching nodes are found. If the input is the ColorNode, the FloatValueNode, the Matrix2Node or their versions, the values are added to the Effect through the *addFloatParameterToCurrentPass(float[] values, String name)* and the *addFloatMatrixParameterToCurrentPass(float[] values, String name)*. These methods internally retrieve glslUniformParameters and add new parameters to them. If the input is a TextureNode the *addTextureToCurrentPass(Texture2D texture, String name)* method is used.

Last, the framebuffer object is created for this pass and added to both the Effect and the Code object. After all these steps the pass is considered processed. The correspondent ProcessedPassRecord is created and added to the list of processed nodes.

After the loop is finished and all of the PassNodes are processed it is necessary to take care of the ScreenNode. The ScreenNode is processed in a similar way as regular PassNodes except for the fact that all of the settings are fixed. The ScreenNode always renders screen aligned quad. That is why the Saq object is created instead of the Pass object. Also the shaders are fixed. The vertex shader handles the proper creation of UV coordinates. The fragment shader is simple and ensures that the unchanged data from the input sampler2D are rendered onto the screen.

3.7 Generation of the code

As mentioned before, JOGL needs implementation of the four methods of GlEventListener for proper rendering. That means the code must be divided into several areas.These areas are variables, init, reshape, display, constructor and they are created as a StringWriter class. The Code class uses similar methods as the Effect class. Once the pass is added through the *addPass(Pass pass, String name, String vertexShaderPath, String fragmentShaderPath)* method the code is generated for each of the areas mentioned above. The proper variables are created in the StringWriter object called "variables", the code for the pass initialization is added into the "init"StringWriter and so on. This process is analogous to the other effect components such as textures, float parameters etc.

Some parts of the generated code are always fixed. These parts are, for example, methods for image loading, imports and the main method. It is useful to store these parts of code in a separate file as a template. The EffectDesigner uses a simple text file so that the user can easily change it for better results. The EffectDesigner will find out where to put the text from the StringWriters due to the delimiters in the template. These delimiters

are:

```
//*variables
//*constructor
//*init
//*display
//*reshape
```

It is important to mention that the code is generated during the process of generation of the effect, and not when the window with the code viewer is opened. This means that all of the changes are not projected into the generated code until the effect is properly generated.

3.8 File format

The program will be useless without the ability to save and load the work. It is a common and necessary feature. The most advantageous approach is to use some kind of self-describing data. The reason for that is that the program will not loose compatibility to the previous save formats if we add new features to the file format. That is a very important fact in programing of such IDEs. The XML is such a format and because it is heavily standardized, there are many tools and libraries for working with this format. Some libraries use sequential approach for parsing XML files, some use more object approach. For the EffectDesigner it is more effective to use the object approach – the DOM parser.

The root element of the eff file format is a schema element. This schema contains two obligatory elements – nodes and connections. Nodes element can contain various elements – one for each type of the node. There can be many nodes, but only one screen node.

Each type of the node element has at least three obligatory children. These elements are name, position and id. The position element has two attributes: location on x axis and location on y axis. The other elements are various depending on the node type. The PassNode element has in addition the scene element, which contains a path to the scene, the vertexShader element, which contains a path to the vertex shader file, the fragmentShader element, which contains a path to the fragment shader file, geometryShader element, which contains a path to the geometry shader file. In addition the passNode elements can contain series of renderState elements. The RenderState elements serve for storing information about the RenderState object adjustments and they have 2 attributes: name and value. The User Node contains, in addition to the name, position and id elements, the path element. This element stores path to the edn file location. The FloatValueNode, the Matrix2Node and their versions contain values element. This element may contain several value elements. Number of these elements depend on the type of the node. For example, the Matrix2Node will contain 4 value elements. The Value element stores a single number.

The Connections element contains only the connection elements. The Connection contains two obligatory elements – the sourceAnchor and the targetAnchor. These elements

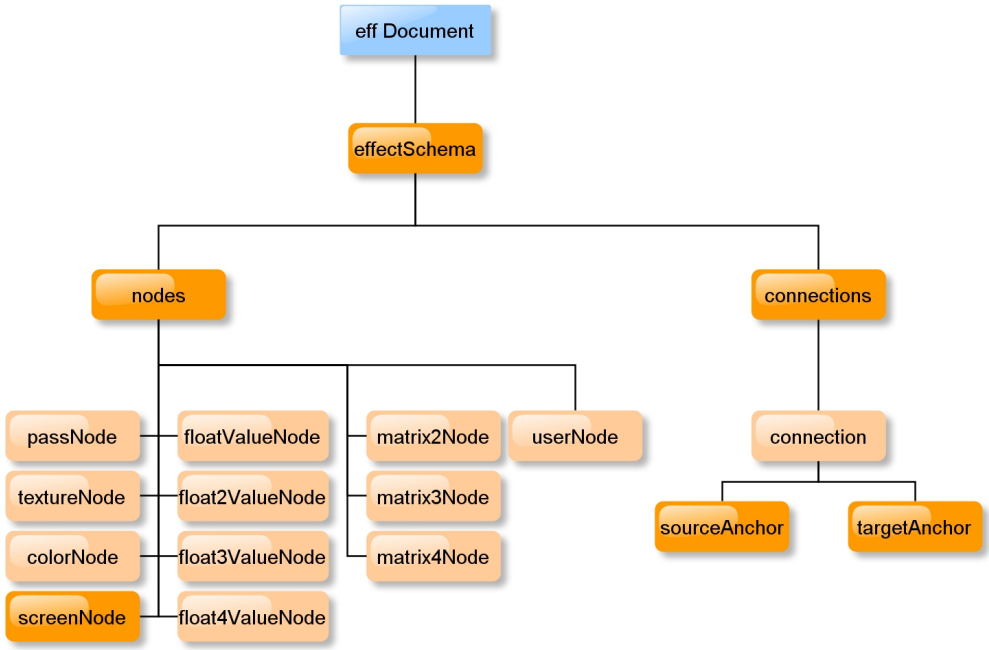

Figure 15: Hierarchical order of the elements in eff file format.

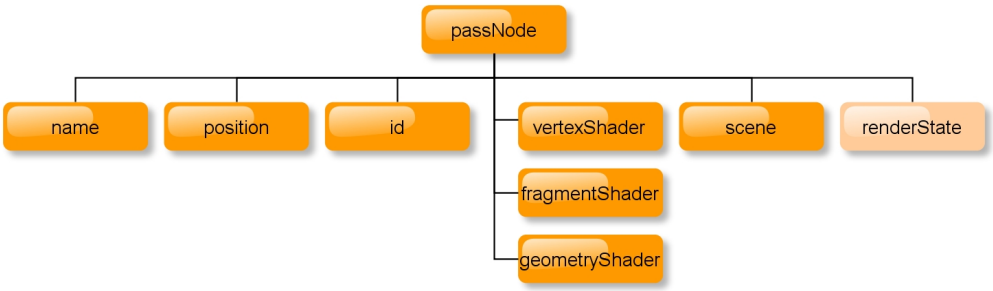

Figure 16: Document object model of the passNode element.

represent the ends of the connection and they have 3 attributes: id for identification of the node, type of the variable and variable name for the identification of a specific input or output of the node.

3.9 User Node

As mentioned in the previous chapter the user nodes are intended to be created the same way as the regular effects. After finishing the node schema, the schema and it's resources

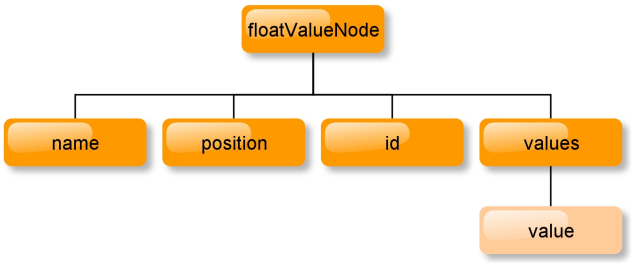

Figure 17: Document object model of the floatValueNode element.

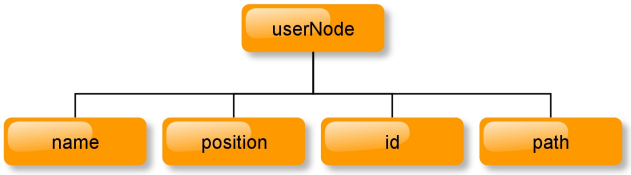

Figure 18: Document object model of the userNode element.

are to be packed in one file. For packing the EffectDesigner is using the Zip compression to reduce the size of the node files. Since the resource files can be located anywhere on the computer and they can also have the same names it is necessary to guarantee that the different files with the same names will not be overwritten in the package. This problem is solved by adding a dictionary hash map to the process. Each path of the resource is added to the dictionary as a key and the new name is generated. Once the file with the same path came on the queue the dictionary returns the appropriate name.

The file describing the user node scheme has similar structure as eff file. In addition there are three new obligatory elements in the root element. It is the name of the node, the description of the node and the links element. Links element may contain two types of elements: the inputLink and the outputLink elements. These elements are used for mapping the inputs and outputs of the user node onto the inputs and outputs of the inner pass nodes. They have several obligatory attributes. The label attribute describes the purpose of the input or output. The id attribute serves the identification of the node and the name and the type attributes serve the identification of the concrete input or output.

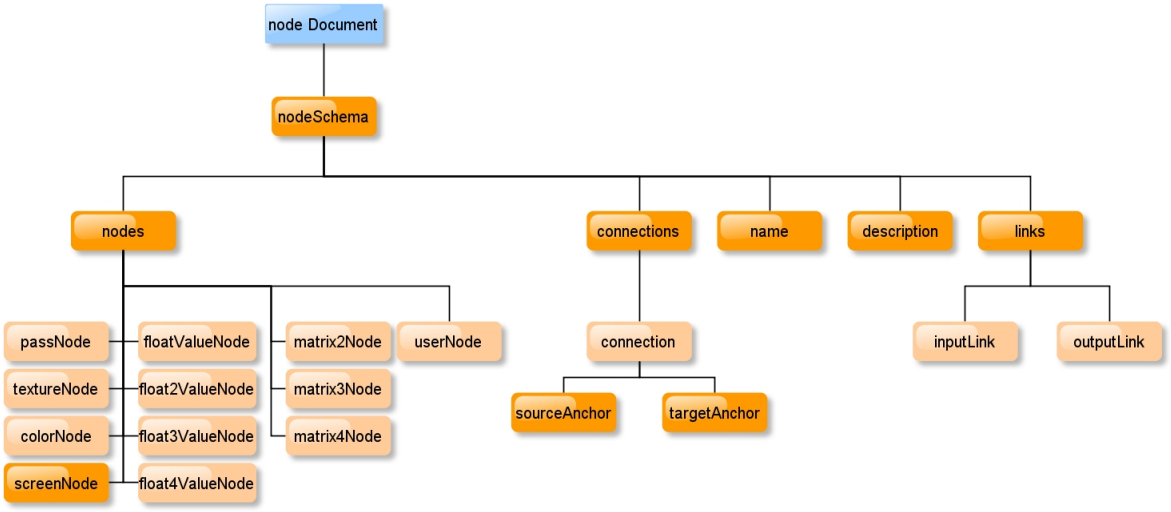

Figure 19: Document object model of Node Document.

4 Testing

EffectDesigner was tested on both single effects and more complex effects. As mentioned in the Analysis and Design chapter some basic fragment operations should be provided as nodes. Since these operations could be easily realized by shaders it is profitable to create them as User Nodes.

The first useful operation is an addition function. The addition is a two operand function so the inputs of such a node should be two sampler2D variables. In the fragment shader the fragments from the both samplers are taken and the addition is applied. Since the GLSL is working with the saturated arithmetic all of the values greater then 1.0 have to be trimmed. The behaviour of such a function can be seen in Fig. 20.

Figure 20: The behaviour of the addition function.

The second useful operation is a multiplication function. The multiplication is, as the addition function, a two operand function. The two fragments retrieved from the two input sampler2D variables are multiplied. The behaviour of such a function can be seen in Fig. 21.

Figure 21: The behaviour of the multiplication function.

The inversion, in opposite to the previous two operations, is a one operand function. The source color is replaced with the complement color. This is realized by a subtraction of an RGB value from the unit vector. The behaviour of such a function can be seen in Fig. 22.

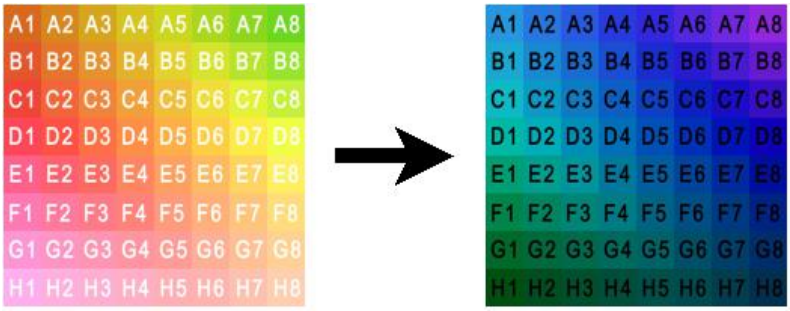

Figure 22: The behaviour of the inversion function.

The blur is a simple function. It takes values from the surroundings of the fragment and the result is a weighted average of these values. The behaviour of such a function can be seen in Fig. 23.

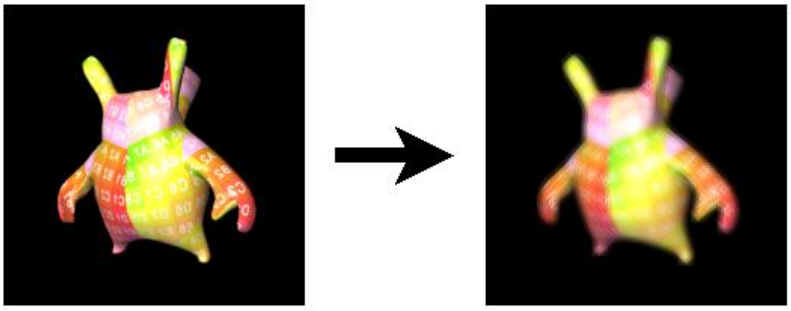

Figure 23: The behaviour of the blur function.

The edge detection takes values from the surroundings of the fragment. If one of the surrounding values differs too much from the average the edge is detected. The shader puts a given color to the places where the edge has been detected. The behaviour of such a function can be seen in Fig. 24.

All of the mentioned functions were successfully implemented and packed as user nodes. This means that they can be used in following effect.

One of the effects which was tried in EffectDesigner is the silhouette highlighting effect. The scheme of this effect can be seen in Fig. 25 and the result can be seen in Fig. 26. At the first the model is rendered using the phong shading algorithm with texture mapping. Next, the model is rendered once more, but this time with a flat color shading. In this case, the background is rendered white and the model in black color. Result of this pass is, in fact, a mask which is further used as base for edge detection. Then the edge is blurred for better visual impression. In the end the blurred edge is multiplied with textured model, which was rendered in the first pass. The result is sent on the screen.

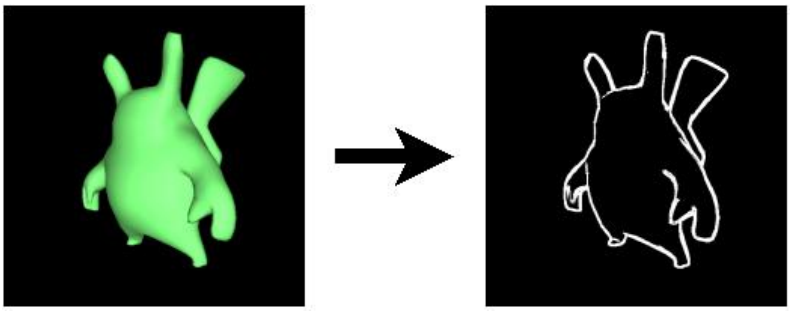

Figure 24: The behaviour of the edge detect function.

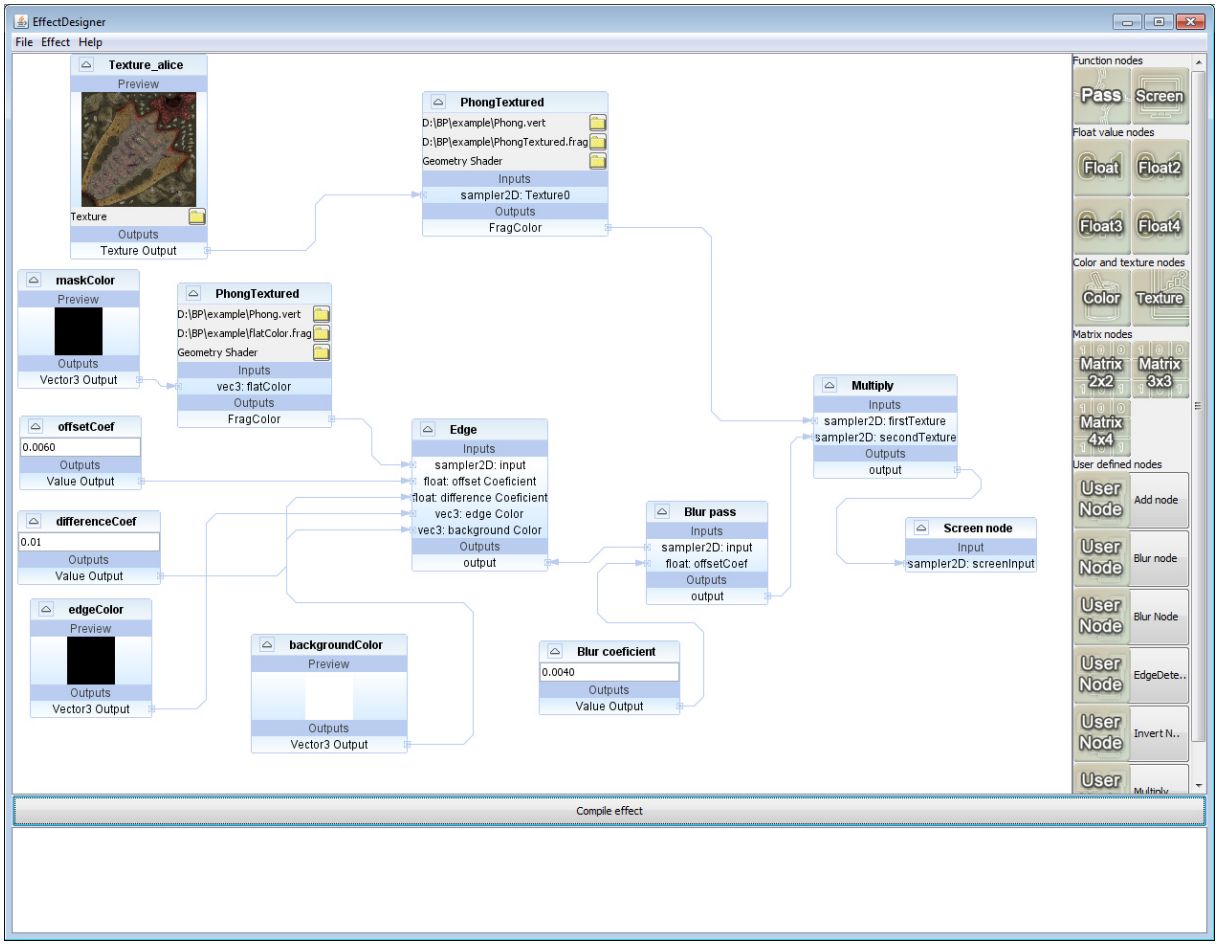

Figure 25: EffectDesigner interface and schema of the silhouette highlighting.

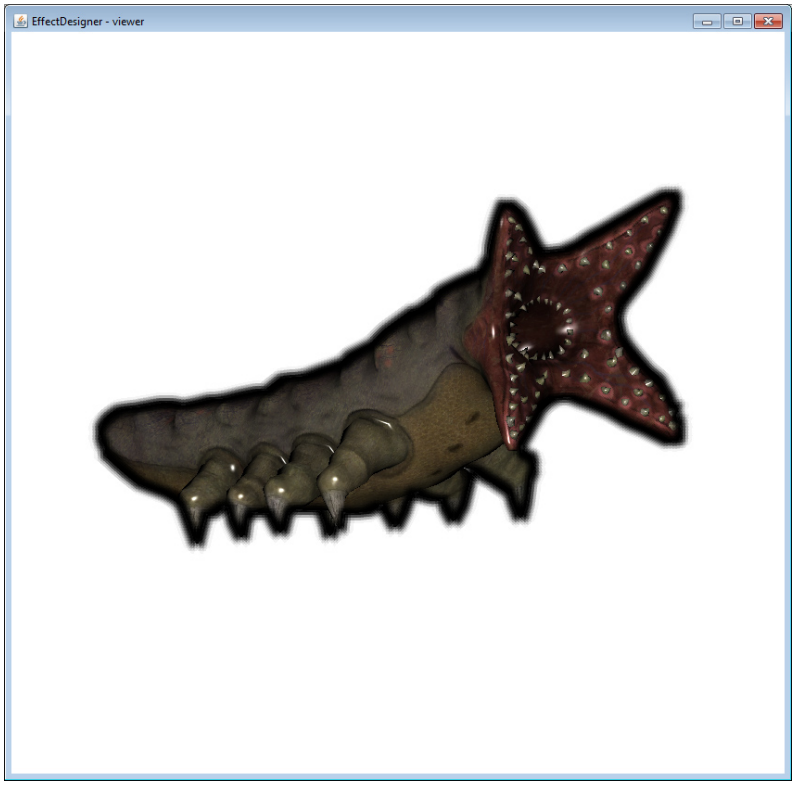

Figure 26: Result of the silhouette highlighting.

5 Conclusion

During my work on this bachelor thesis the informations about graphics, especially multi-pass rendering, was gathered. The existing solutions were elaborately examined and compared. From these processes the recommendations for features of the EffectDesigner were taken. These features are the visual development and the code generation. These features along with the assignment were taken as a base for the design of the program. The designed program was implemented using Java, JOGL, Tiger and Visual Library. The implemented solutions were tested on complex effects with good results.

Result of this bachelor thesis, the EffectDesigner, is a functional program and the features of visual development and code generation make the program competitive. Visual development concept has been achieved by using node system. The user is able to choose from various nodes. These nodes are pass nodes, nodes representing the variables or the nodes created by a user. The creation of custom nodes was achieved with the most comfortable way for user. The program is ready for extension of its functions by a system of user nodes. This solution was used for creation of some mathematic and artistic functionality such as addition, multiplication or edge detection. The generated code, which originated from unification of modifiable template and generated lines, is simple and can be used as a base for further development.

5.1 Further development

Although EffectDesigner is fully functional program there are lot of things which should be considered for future development.

EffectDesigner does not offer any way how to view and edit shader files. This feature is necessary for comfortable shader development. Hence, the first feature to be implemented into EffectDesigner interface could be text editor, which should be able to highlight GLSL syntax. Other possibility is to integrate EffectDesigner into some bigger package. For example, NetBeans OpenGL pack. OpenGL pack includes plugins for NetBeans which add syntax highlighting for GLSL shaders.

Other area which would profit from future development is the support for all types of GLSL variables. Now the EffectDesigner supports the most used and common types of variables. Uncommon types, such as, for example, sampler1D, uint, dmat4x2 are not supported.

Next thing for future development would be optimization. Right now, framebuffer objects and texture objects for rendering are created automatically for each pass. With limited amount of VRAM space this could lead to malfunction. That is why the automatic process of optimization would be useful. This process would find out framebuffer objects that can be shared and unite them.

Last direction of improvements should be pointed onto sorting of user nodes. Right now, the user nodes are sorted alphabeticaly. It is completely sufficient for small amount of items. Once there are many of the user nodes it could get confusing. User nodes should

be sorted by the type of function they provide.

Once these areas are examined and implemented EffectDesigner will become even more competitive alternative to other GLSL shader development IDEs .

Reference

[1] JIM X. Chen, CHUNYANG Chen. *Foundations of 3D Graphics Programming.* Springer-Verlag London Limited 2008, ISBN: 978-1-84800-283-8

[2] WRIGHT S. Richard Jr., LIPCHAK Benjamin, HAEMEL Nicholas *OpenGLSuperbible.* Fourth Edition, Addison-Wesley 2007, ISBN-13: 978-0-321-49882-3

[3] ŽÁRA Jiří, BENEŠ Bedřich, SOCHOR Jiří, FELKEL Petr *Moderní počítačová grafika.* 2. vydání, Computer Press Brno 2004, ISBN-10: 80-251-0454-0

[4] *The OpenGL Graphics System: A Specification Version 2.1*, © 1992-2006 Silicon Graphics Inc., Available from WWW: <`http://www.opengl.org/registry/doc/glspec21.20061201.pdf`>

[5] *The OpenGL Graphics System: A Specification Version 3.0*, © 2006-2008 The Khronos Group Inc., Available from WWW: <`http://www.opengl.org/registry/doc/glspec30.20080811.pdf`>

[6] *The OpenGL® Shading Language: Language Version 1.20*, © 2002-2006 3Dlabs Inc., Available from WWW: <`http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.20.8.pdf`>

[7] *The OpenGL® Shading Language: Language Version 4.00*, © 2008-2010 The Khronos Group Inc., Available from WWW: <`http://www.opengl.org/registry/doc/GLSLangSpec.4.00.8.clean.pdf`>

[8] *JOGL JSR-231 2.x specification*, © 2005 Sun Microsystems [cit. 2010-05-20]. Available from WWW: <`http://download.java.net/media/jogl/jogl-2.x-docs/`>

[9] *Visual Library 2.0 - Documentation*, © 2010 Oracle Corporation [cit. 2010-05-20]. Available from WWW: <`http://bits.netbeans.org/dev/javadoc/org-netbeans-api-visual/org/netbeans/api/visual/widget/doc-files/documentation.html`>

[10] EVERITT Cass, REGE Ashu, CEBENOYAN Cem *Hardware Shadow Mapping*, © 2010 NVIDIA Corporation [cit. 2010-05-20]. Available from WWW: <`http://developer.nvidia.com/attach/8456`>

A Appendix – Content of the attached CD

Content of the CD is divided into four directories. In the directory program the actual program is located with all of the libraries included. The program is being executated by EffectDesigner_32bit.bat for 32-bit version and EffectDesigner_64bit.bat for 64-bit version. Both of these files are designated for Microsoft Windows. In the directory scr the source codes of the EffectDesigner are located in the form of complete NetBeans project. In the directory text you can find the text of this bachelor thesis in pdf format. In the directory resources you can find resources such are textures, vertex shaders and fragments shaders which you can use for creating effects.