

ScholarWorks@GSU

Platform Independent Real-Time X3D Shaders and their Applications in Bioinformatics Visualization

Authors	Liu, Feng
Citation	Liu, Feng (2007). Platform Independent Real-Time X3D Shaders and their Applications in Bioinformatics Visualization. Dissertation, Georgia State University. https://doi.org/10.57709/1059434
DOI	https://doi.org/10.57709/1059434
Rights	I hereby certify that, if appropriate, I have obtained and attached hereto a written permission statement from the owner(s) of each third party copyrighted matter to be included in my thesis, dissertation, or project report, allowing distribution as specified below. I certify that the version I submitted is the same as that approved by my advisory committee. I hereby grant to Georgia State University or its agents the non-exclusive license to archive and make accessible, under the conditions specified below, my thesis, dissertation, or project report in whole or in part in all forms of media, now or hereafter known. I retain all other ownership rights to the copyright of the thesis, dissertation or project report. I also retain the right to use in future works (such as articles or books) all or part of this thesis, dissertation, or project report.
Download date	2026-04-12 00:18:04
Link to Item	https://hdl.handle.net/20.500.14694/4015

PLATFORM INDEPENDENT REAL-TIME X3D SHADERS AND THEIR APPLICATIONS
IN BIOINFORMATICS VISUALIZATION

by

FENG LIU

Under the Direction of Scott Owen

ABSTRACT

Since the introduction of programmable Graphics Processing Units (GPUs) and procedural shaders, hardware vendors have each developed their own individual real-time shading language standard. None of these shading languages is fully platform independent. Although this real-time programmable shader technology could be developed into 3D application on a single system, this platform dependent limitation keeps the shader technology away from 3D Internet applications. The primary purpose of this dissertation is to design a framework for translating different shader formats to platform independent shaders and embed them into the eXtensible 3D (X3D) scene for 3D web applications. This framework includes a back-end core shader converter, which translates shaders among different shading languages with a middle XML layer. Also included is a shader library containing a basic set of shaders that developers can load and add shaders to. This framework will then be applied to some applications in Biomolecular Visualization.

We first defined a minimal set of shaders for common elements in protein molecules such as “Carbon”, “Nitrogen”, “Oxygen”, “Hydrogen”, “Phosphorus”, “Sulphur”, and “Other”, which is used for all undefined elements. Then 3D molecule data sets are converted in a pipeline from

PDB to CML to X3D. During the conversion from CML to X3D, we automatically add predefined shaders to each of the elements. At the end of this pipeline, a high quality real-time shaded molecular structure is created and ready for use on the web. Considering the usability, a set of functions for improving the user manipulation and optimizing the real-time interactive performance has been designed. A Multi-Users Shared Environment has been set up for sharing dynamic shaders on web3D application.

INDEX WORDS: web3D, Shader, GPU, X3D, VRML, Shader Language Converter, SLC, High-Level Shading Language, Cg, OpenGL shading Language, XML, 3D Molecule Structure, PDB, CML, Level of Quality, LOQ, Multi-Users Shared Environment, MUSE.

PLATFORM INDEPENDENT REAL-TIME X3D SHADERS AND THEIR
APPLICATIONS IN BIOINFORMATICS VISUALIZATION

by

FENG LIU

A Dissertation Submitted in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

in the College of Arts and Sciences

Georgia State University

Atlanta, Georgia
2005

Copyright by

Feng Z. Liu

2005

PLATFORM INDEPENDENT REAL-TIME X3D SHADERS AND THEIR
APPLICATIONS IN BIOINFORMATICS VISUALIZATION

BY
FENG LIU

Major Professor: Scott Owen
Committee: Raj Sunderraman
Irene Weber
YanQing Zhang
Ying Zhu

Electronic Version Approved:

Office of Graduate Studies
College of Art and Sciences
Georgia State University
December 2005

ACKNOWLEDGEMENTS

First of all, I would like to say “Thank you” to my adviser Dr. Owen for his many years’ supports and giving me chance to explore the cutting edge technologies. He also gave me proper direction in the field and eventually helped me find the topic for my dissertation. This is an unforgettable support for me. I also cannot forget the support from Dr. Sunderraman, a really nice graduate student advisor, with intelligent and understanding. I would also like to express my thanks to all of my committee members for your time reviewing my dissertation, discussing my dissertation topic and for the corrections and suggestions from all of you.

The process of this dissertation is not only to achieve a degree. It bring more to me is a spirit. A spirit of patience, working hard, and persistence prepared me to take on many kinds of difficulties in my life.

The supports from my friends and my family have been the most valuable. Whenever I lost and came back with them, they never give up on me. I can’t forget Yun’s encourages. “Never give-up”, that is from my parents. I’d also like to have thanks to all my sisters and brothers-in-law. Thanks for their no-hesitate-helps. Thanks to all my lovely nieces and nephews being my buddies. Han Wei, my 247 helper, not only helped me correct grammar mistakes but also my no-give-up supporter and a life long love. I’m also like to say thanks you to people who were in my story. The experiences I had with you encouraged me as well to achieve this goal. Thank you all!

-- Feng Liu in Atlanta Fall 2005

For

Mom & Dad

Sumei Zhang & Yuxiang Liu

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	VI
TABLE OF CONTENTS	VIII
LIST OF FIGURES	XIII
LIST OF TABLES	XVI
CHAPTER	1
1 INTRODUCTION	1
1.1 Contribution	3
1.2 Content organization	4
2 BACKGROUND	6
2.1 Graphics Processing Units (GPUs)	6
2.2 High-level Shading Languages	11
2.2.1 Procedural Shading	11
2.2.2 C for Graphics from NVIDIA	17
2.2.3 OpenGL shading Language	22
2.2.4 High Level Shading Language	25
2.3 X3D: The next generation of VRML97 in XML	28
2.4 The Challenge of Real-time Procedure Shaders in VRML/X3D	34
3 DESIGN AND IMPLEMENTATION	40
3.1 A Framework of Platform Independent Shaders in X3D	40
3.1.0 Initial ideas of embedding Shaders into X3D	43

3.2	Shader Language Converter Design	46
3.2.1	Why the XML middle layer?	47
3.2.2	XML Layer definition.....	50
3.2.2.1	<i>Basic Data Types</i>	50
3.2.2.2	<i>Texture Sampler Types</i>	55
3.2.2.4	<i>Semantics in Cg/HLSL vs. Built-in Attributes and Variables in GLSL</i>	58
3.2.2.5	<i>Struct Definition</i>	62
3.2.2.6	<i>Operators/Operation Definition:</i>	63
3.2.2.7	<i>Statement definition</i>	64
3.2.2.8	<i>Functions Design</i>	65
3.3	Shader language Converter Implementation.....	67
3.3.1	SLC Conversion pipeline.....	67
3.3.2	Scanner.....	69
3.3.3	Parser.....	70
3.3.3.1	<i>Data types Conversion:</i>	70
3.3.3.2	<i>Declaration Conversion:</i>	70
3.3.3.3	<i>Parameter Conversion:</i>	72
3.3.3.4	<i>Operator Conversion:</i>	75
3.3.3.5	<i>Function Conversion</i>	76
3.3.4	SLC interface	77
3.3.5	Shader comparisons between pre-converted and post-converted	78

3.4 Summary of Shader-X3D frameworks and preview of its application in Bio-informatics Visualization.....	80
4 SHADER-X3D USES IN BIOINFORMATICS APPLICATIONS.....	82
4.0 Preface.....	82
4.1 Background of structural biology and Web 3D Bioinformatics Visualization.....	85
4.1.1 3D Protein structure in different formats.....	88
4.1.1.1 <i>PDB</i>	88
4.1.1.2 <i>CML</i>	90
4.1.1.3 <i>X3D/VRML</i>	93
4.1.2 Software packages and other techniques for visualizing bioinformatics.....	103
4.2 Designs and Implementations shader-X3D for molecule presentation.....	107
4.2.1 Standard shader for minimal element set for protein presentation.....	107
4.2.1.1 <i>Design</i>	107
4.2.1.2 <i>Implementation</i>	109
4.2.1.3 <i>Performance analysis and visual effect comparisons</i>	120
4.3 Enhanced user friendly interface for molecular presentation.....	125
4.3.1 Design.....	125
4.3.2 Implementations.....	128
2.4 Future interface design for 3D structural biology.....	139
5 MULTI-USER SHARED ENVIRONMENT IN X3D/VRML WITH SHADER SUPPORTED.....	140
5.1 Concepts of Multi-user shared environment and ASEC.....	140

5.2	Implantation and Improvement analysis of MUSE with shaders	144
6	CONCLUSION.....	151
	REFERENCE:.....	153
	Appendix A. shaders examples in Cg and GLSL	158
	A1a – wood_vert.cg.....	158
	A1aa – wood.vert.....	159
	A1bb – wood.Frag	159
	A1aaa – wood.vert.Xml	160
	A1c – brick_vert.cg.....	162
	A1d – brick_frag.cg	163
	A1cc – brick.vert.....	163
	A1dd – brick.frag.....	164
	A1e – Carbon atom-shader in Cg.....	164
	A1f –Carbon atom-shader in Cg.....	165
	A1ee –Carbon atom-shader in GLSL	165
	A1ff –Carbon atom-shader in GLSL	166
	Appendix B. Partially mapping tables in for Shader Language Converter.....	166
	B1 – Data Types/Sampler types mapping table.....	166
	B2 – Operators mapping table (special case only).....	167
	B3 – modifiers/qualifiers mapping table.....	167
	B4 – Transformation Matrices mapping table	168
	B5 – Semantics in Cg or HLSL/building variables in GLSL mapping table.....	168

B5 – functions mapping table	168
Appendix C. Source Code.....	169
C1 – Partially Shader Language Converter Source Code.....	169
C2 – XSL source code for X3D + shader 2 VRML + shader conversion	175
C3b – example caffeine in VRML_Shader.....	181
C4c – specular light adjustable Interface example functions and ROUTEs.....	187
C4e – bump adjustable Interface example functions and ROUTEs	188
C5 – MUSE bump sharing Interface example functions and	190
Appendix D – Image Credits	192
Appendix E – code Credits	193
Appendix F – SLC source code (Avaliable with requirment)	193

LIST OF FIGURES

Figure 2 - 1 Plastic shader on the spherical surface.....	13
Figure 2 - 2 Graphics Pipeline	15
Figure 2 - 3 GPU – CPU Interface in modern Graphics Pipeline.....	19
Figure 2 - 4 Cg Shaders loading process	20
Figure 2 - 5 GLSL Shaders loading process	23
Figure 2 - 6 Google Hits of 3D websites	29
Figure 2 - 7 X3D/VRML Growing history	32
Figure 2 - 8 Shaders in VRML/X3D web browser on single machine.....	35
Figure 2 - 9 Different Shaders format compatibility	36
Figure 2 - 10 High-level overview of the Vertex/fragment shader.....	38
Figure 3 - 0 SLC convert shader skin	40
Figure 3 - 1 a Framework of Platform Independent Shaders in X3D.....	41
Figure 3 - 2 illustration of shader library	42
Figure 3 - 3 Shader Language Converter in X3D application	46
Figure 3 - 4 Structure of Shader Language Converter.....	47
Figure 3 - 5 Comparison of the SLC design structure with/without XML layer.....	47
Figure 3 - 6 new shading language on the SLC structure design with XML layer.....	48
Figure 3 - 7 a new shading language on the SLC design structure without XML layer....	49
Figure 3 - 8 : XML layer common data set definition	53
Figure 3 - 9 Coordinate System and Transformation for Vertex Processing.....	57
Figure 3 - 10 example 3.4 struct in XML tree structure	62

Figure 3 - 11 XML tree structure of example 3.5	63
Figure 3 - 12 : CG→XML→GLSL flow	68
Figure 3 - 13 : CG→XML→GLSL conversion Path.....	69
Figure 3 - 14 : Varying variable in GLSL relocated as parameters in Cg	73
Figure 3 - 15 : Shader Language Converter Interface.....	77
Figure 3 - 16 : Shader Comparison between pre_conversion and post_conversion	78
Figure 3 - 17 : The Comparison of with and without shaders in shader library	79
Figure 3 - 18 : Framework for Independent, real-time, interactive Shaders in X3D	80
Figure 4 - 0: Beauty of life.....	82
Figure 4 - 1 Antibodies are immune system proteins	83
Figure 4 - 2 Using light to measure an object,.....	84
Figure 4 - 3 : HIV proteases.....	86
Figure 4 - 4 : HIV protease cell	87
Figure 4 - 5 : Node ROUTE function in VRML/X3D.....	96
Figure 4 - 6 : PDB → CML & CML→ Shaded X3D Conversion.....	108
Figure 4 - 7 : ShaderAppearance Prototype in VRML	112
Figure 4 - 8 : VetexShader Prototype in VRML.....	113
Figure 4 - 9 : FragmentShader Prototype in VRML.....	113
Figure 4 - 10 ProtoDeclare for Atom - Carbon without shader	114
Figure 4 - 11 Shader Declarations for the ProtoDeclare of any atom.....	115
Figure 4 - 12 Shader Declaration for the ProtoDeclare of Atom – carbon	115
Figure 4 - 13 Shader for the Atom – carbon after converted	116

Figure 4 - 14 ShaderAppearance definations in XSL modification.....	117
Figure 4 - 15 MovieTexture tag definition modification in XSL with shader added in ..	117
Figure 4 - 16 cml2x3d_shader/ x3d_shader2_vrml_shader execute comments	118
Figure 4 - 17 A converted shader-X3D riboflavinN 3D structure in VRML	119
Figure 4 - 18 Caffeine Structure	121
Figure 4 - 19 A closer comparison of the protein structures.....	123
Figure 4 - 20 Left: partially Est30 ball-stick structure with procedural shaders.	124
Figure 4 - 21 using LOQ with shader for better performance	127
Figure 4 - 22 Multi-shader selector interfaces	128
Figure 4 - 23 Level of quality example for CH4O.....	131
Figure 4 - 24 Level of quality approach while close to model of CH4O.....	132
Figure 4 - 25 controlled specular light of shader on a atom surface.....	137
Figure 4 - 26 A controlled bump shader on a atom surface.....	138
Figure 4 - 27 resulting image of adding new atoms in predefined functions	138
Figure 4 - 28 user interface design for the structural biology.....	139
Figure 5 - 1 High-level structure of ASEC System	140
Figure 5 - 2 the slides tutorial updated information real-time sharing structure	142
Figure 5 - 3 Shader sharing of Caffeine structure in multi-user environment	147
Figure 5 - 4 Shader sharing in multi-user environment	148

LIST OF TABLES

Table 3 - 1 three shading language basic data type table 1	50
Table 3 - 2 three shading language basic data type mapping table 2	51
Table 3 - 3 Comparisons basic data type between Cg/HLSL with GLSL	52
Table 3 - 4 X3D, XML and shading languages basic data type mapping table	54
Table 3 - 5 X3D, XML and shading languages Sampler type mapping table	55
Table 3 - 6 X3D, XML and shading languages basic Qualifiers mapping table	56
Table 3 - 7 X3D, XML and shading languages Transformation matrix mapping table ...	58
Table 3 - 8 XML, Semantics and shading languages built-in variable mapping table	61
Table 3 - 9 Mix function in GLSL mapping to different languages	66
Table 4 - 1 Shader definition for basic elements in Molecular Structure	110
Table 4 - 2 Comparison molecular structure with and without procedural shaders	122

Chapter

1 Introduction

Since the introduction of programmable Graphics Processing Units (GPUs) and procedural shaders, hardware vendors have each developed their own individual real-time shading language standard. None of these shading languages is fully platform independent. Although this real-time programmable shader technology could be developed into 3D application on a single system, this platform dependent limitation keeps the shader technology away from 3D Internet applications.

The primary purpose of this dissertation is to design a framework for translating different shader formats to platform independent shaders and embed them into the eXtensible 3D (X3D) scene for 3D web applications. This framework includes a back-end core shader converter, which translates shaders among different shading languages with a middle XML layer. Also included is a shader library containing a basic set of shaders that developers can load and add shaders to. This framework will then be applied to some applications in Biomolecular Visualization.

We first defined a minimal set of shaders for common elements in protein molecules such as “Carbon”, “Nitrogen”, “Oxygen”, “Hydrogen”, “Phosphorus”, “Sulphur”, and “Other”, which is used for all undefined elements. Then 3D molecule data sets are converted in a pipeline from PDB to CML to X3D. During the conversion

from CML to X3D, we automatically add predefined shaders to each of the elements. At the end of this pipeline, a high quality real-time shaded molecular structure is created and ready for use on the web. The results show the interactivity and visualization performance comparison between molecular structures with and without shaders in X3D. Considering the usability, a set of functions for improving the user manipulation and optimizing the real-time interactive performance has been designed.

One of the reasons that make these visualizations possible is the revolutionary improvement of the GPU enabling user defined programmable shaders to produce high quality visualizations in real-time. Current 3D web applications in VRML do not produce high quality images due to the fast but low quality rendering algorithm used for specular highlights. The introduction of programmable GPUs and the addition of procedural shaders to X3D provide us with new techniques to develop real-time web based scientific visualization environments. Additionally, this technique can be used not only for online scientific research, but also in online filming, gaming, e-commerce, communication, simulation, and online architectural demonstrations etc.

Three steps were taken for completing this dissertation. The First step was to research the current status of on web based 3D applications; analyze shading languages from different vendors; determine the compatibility problems for sharing shaders via the web and propose a solution. A shader can be converted from one shading language into another shading language another format for better compatibility with the hardware before it communicates with the graphics API. An XML middle layer based shader

language converter (SLC) was partially developed to prove the concept of the shader's convertibility. An analysis of where the SLC could be added into the X3D publishing process was performed.

The second step was to use our framework in a biomolecular application. We researched the current status of web based 3D biomolecular visualizations and existing 3D data formats; analyzed the best way to introduce the shader plus X3D framework to biomolecular research; defined Phong shaders for a typically used minimal set of atoms in proteins; loaded the predefined shader for each atom during the conversion from the popular 3D molecule format, PDB (Protein Data Bank), to CML (Chemical Markup Language), to the X3D format. At the end of the conversion, comparisons are given on the visual and interactivity performance of different sized biomolecular data sets with and without the shader. For improved usability, we designed a user-friendly interface for manipulating molecules. A set of basic functions was implemented for this interface. In the last step, we added shaders into a multi-user shared environment (MUSE). We performed an analysis of the benefits of using shaders in a MUSE and an analysis of the potential shader use in MUSE applications.

1.1 Contribution

The main contribution of this dissertation is a framework design for sharing platform independent shaders via the Internet and applying this framework to Bioinformatics and chemistry research. Additionally, we give a perspective application

analysis of this framework for other industries. Core parts of this framework are, a concept of exchangeable shaders and a partially developed shader language converter for converting shader formats among different shading languages by use of a middle XML layer, which proved the proposed exchangeable shader idea. To introduce the independent shader framework in the biomolecular application in X3D, a set of shaders was defined for a small set of common atoms found in proteins. These atom shaders are implemented automatically when 3D molecular data in CML is converted to the X3D molecule structure. We also developed a set of functions for manipulating the shaders between the scenes and the dynamic shader shared between the scenes in the MUSE system.

1.2 Content organization

The content of this dissertation is organized in four sections. In section 1 (chapters 2 and 3), we discuss the platform independent real-time shader framework, which includes background and a literature review about GPUs, shader languages, X3D and embedding shaders into X3D. It also includes explanations of the problems inherent in using shaders in web 3D applications via the Internet. We proposed the idea of converting shaders into different shading languages for solving this platform dependent problem. We then implement a shader language converter to prove our proposal and build a framework for platform independent real-time shaders, which allows shaders wider use in practical applications.

In the section 2 (Chapter 4), we demonstrate the framework in section one by applying it to biomolecular visualization, for example, a 3D molecule structure with shaders displayed on the web. We define shaders for the common atoms found in proteins. Then a pipeline is set-up to convert the Protein Data Bank (PDB) files to Chemical Markup Language (CML) files, then from CML file to X3D files with shader support. We also have visual and performance comparison analysis of the test results. We show the necessity of applying shaders in bioinformatics / chemistry applications and point out the benefits. For improving usability, we designed an interface for better manipulation of the shaders in the X3D Scene. A set of basic functions was implemented.

In the section 3, (Chapter 5) we demonstrate how to share shader information in a MUSE system with two scenarios. One is a collaborating work for molecular visualization; the other is a shared bump painting in a 3D gallery.

The fourth section contains conclusion (Chapter 6), references, appendixes and source code as supplementary information for sections 1, 2 and 3.

2 Background

We will begin our discussion with a review of recent literatures related to the development of GPUs and the current popular programmable shading languages, continuing with 3D Web Graphics and previous works of embedding shaders into X3D. We then analyzed and revealed the difficulty for sharing shaders on the Internet.

In chapter 3, the design and implementation of a framework for sharing the platform independent shader in X3D will be discussed.

2.1 Graphics Processing Units (GPUs)

A Graphics Processing Unit (GPU) is a microprocessor located on a graphics card (or graphics accelerator) used in a personal computer, workstation or game console. Modern GPUs implement a number of graphics primitive operations in a way that make running them much faster than drawing directly to the screen with the host CPU. They are efficient at manipulating and displaying computer graphics, and their highly parallel structure makes them more effective than typical CPUs for a range of complex algorithms.

GPUs when first introduced used monolithic graphics chips of the late 1970's and 1980's. They optimized two-dimensional imaging like that found in video games and animations by quickly compositing several images together. Some GPUs at that time were able to run several operations in a display list and use direct memory access (DMA)

to reduce the load on the host processor with no shape drawing support. It eventually became possible to move drawing support onto the same chip as a regular frame buffer controller such as (Video Graphics Array) VGA, first introduced by IBM. [Wikipedia 2005]

In the late 1980s and early 1990s, high-speed, general-purpose microprocessors were popular for implementing high-end GPUs; graphics boards for PCs and workstations using digital signal processor chips to implement fast drawing functions. However, they were very expensive. In 1991, S3 Graphics introduced the first single-chip 2D accelerator. By 1995, every major PC graphics chip manufacturer had added 2D acceleration support to their chips. At the same time, fixed-function Windows accelerators had surpassed expensive general-purpose graphics coprocessors in terms of Windows performance, and coprocessors faded away from the PC market.

Modern GPUs use most of their transistors to do calculations related to 3D computer graphics. The GPU now exceeds the CPU in the number of transistors present in each microchip. For example, Intel used 55 million transistors in its 2.4 GHz Pentium IV; NVIDIA used over 125 million transistors in the GeForce FX GPU and the latest NVIDIA 7800 GXT has 302 million transistors. NVIDIA introduced the term of “GPU” with its new meaning in the late 1990’s when the term VGA controller was insufficient for describing it accurately. This is because the older styles of GPU, VGA, are really CPU’s responsible for updating all pixel buffers. For the Modern GPU, the pixel updates have been designed into the GPU. Since the major time consuming

operation of Computer Graphics applications is that they involve a large amount of matrix and vector computations, the parallel architecture of new generation GPUs with matrix and vector operations pipelined speeds up these computations. Engineers and scientists have increasingly studied using GPUs for general-purpose computation.

Looking back at the history of the modern GPU development, it can be divided into about five generations. In the pre-GPU stage, before NVIDIA announced the modern GPU in 1998, some graphics companies like SGI and Evans & Sutherland, developed graphics systems like we mentioned earlier. Those graphics systems introduced many graphics concepts like vertex transformation and texture mapping, which are still used widely today. However, those graphics systems were expensive.

The first generation GPU, up to 1998, includes; NVIDIA TNT2, ATI's Rage and 3Dfx's Voodoo3. These GPUs can update the pixel buffer independent from the CPU. They are capable of rasterizing pre-transformed triangles and applying one or two textures. However, they lack the ability to transform vertices of 3D objects; this means vertex transformations have to happen on the CPU. The set of operations at this stage is very limited and only used for math operation to combine textures to compute the final pixel color.

In the second generation stage, from 1999 to 2000, the GPU took the task of 3D object transformation and lighting (T&L), from the CPU. Both OpenGL and DirectX7 support hardware vertex transformation. This is a very important step, since before this point only high-end workstations could achieve fast vertex transformation. The math

operation set extended to cube map texture etc. and the staged GPU's are more configurable. GPUs like NVidia GeForce 256, GeForce2 MAX, ATI Radeon 7500, etc were all developed during this time.

In the third generation, in 2001, some GPUs like GeForce 3, GeForce 4Ti, Microsoft Xbox and ATI's 8500 etc., provided vertex programmability. These GPUs allow the application to specify a sequence of instructions for processing vertices. DirectX8, OpenGL extensions, and ARB_vertex_program exposed vertex-level programmability to applications. They also provide limited pixel level configuration, but no hardware support at this time.

In the fourth generation in late 2002 and 2003, NVIDIA's GeForce FX and ATI's Radeon 9700 appeared on the market with vertex and fragment level programmability. This key stage completed the process of offloading complex vertex transformations and pixel-shading operations from CPU to GPU. The DirectX and various OpenGL extensions exposed both vertex and fragment processors programmability.

Recently, since the power of the programmable GPU has been realized, people keep thinking of ways to reduce the CPU's workload as much as possible. Research resulted in similar ideas with parallel computing on Dual-CPU's. NVidia announced the "NVidia SLI" technology, a custom "X2" motherboard designed to accept a pair of graphics components. Both graphics cards reside on the motherboard. Accordingly, an updated version of motherboard with two graphic card slots is needed. Both of the cards will have to talk between themselves via a connector. For speeding up the data transfer,

the Peripheral Component Interconnect (PCI) express bus was specifically selected for the SLI technology. All PCI Express based GeForce 6800 Ultra, GeForce 6800 GT, or Quadro FX 3400 boards support SLI technology. At the same time ATI also developed The CrossFire platform that gave a multiple graphics processor platform. It combines the power of ATI's Radeon® Xpress chipsets for Intel and AMD processors, a standard Radeon® graphics processor and a Radeon® CrossFire Edition graphics card to bring massive performance and image quality. Although their names are different, the core idea is the same. The processors need to communicate via a connector between them as well. When tasks come they are distributed to both cards and the image is composited together after computation on each individual card. These dual-GPUs/ multi-GPUs structures are very powerful in heavy-duty graphics computation. One important potential application is computer games where they will give much faster and higher quality images.

The reason we give an overview of the GPUs development is to show how fast the development and generation changes are for modern GPUs. About every nine months a new generation graphics card is released. This is good news for graphics application development and possible uses for general-purpose computation as well. However, did the hardware vendors use the same standard during the development of their GPU hardware and software? Some times it is true that “Comparison is a good reason to speed up developments”. The comparisons between different vendors did push the GPUs’ development. However, a platform dependent application has limitations for sharing resources via the Internet.

The current most popular GPU manufacturers are NVIDIA Corporation, ATI Technologies, 3DLabs, Matrox, XGI Technology Inc and Intel. The use of a high-level language interface to communicate with the GPU has almost completely replaced an assembly language interface. ATI is pushing and testing all their GPUs with High Level Shading Language (HLSL), a language designed by Microsoft that is limited to the Microsoft Windows operating system. Nvidia designed its C for Graphics (Cg) shading language for their hardware, and claimed it as a platform independent shader language. 3DLab is pushing the OpenGL Shading Language (GLSL), which has been incorporated as part of OpenGL 2.0. GLSL has the potential to eventually become a major standard shading language. In the following section we will introduce the history of procedural shader language development and focus on the Cg, HLSL, and GLSL. We also give a comparison among these three shading languages. [NVIDIA 2005][3DLab 2005][ATI 2005][OpenGL 2005]

2.2 High-level Shading Languages

2.2.1 Procedural Shading

Rob Cook and Ken Perlin are credited with first introducing shader languages for developing shading calculations. Rob Cook's paper "Shade Trees", at SIGGRAPH 84, [Cook 1984] and Ken Perlin's paper "An Image Synthesizer", at SIGGRAPH 85, [Perlin 1985], both target on offline rendering system. A shader is a program that describes the output of light sources and how this light is attenuated by surfaces and volumes. The

programming language for describing the shader is called the shading language. Rob Cook and Pat Hanrahan first used a shading language in the RenderMan specification. [Apodaca 1999] RenderMan became the most widely used rendering system for high-end special effects. A major reason for RenderMan's success is that compared to earlier types of model shading, the shader and shading language in RenderMan is not limited to a single shading equation like Phong shading. Another book about RenderMan was published in 1992 [Upstill, S. 1992]

When we talked about real-time computer graphics, it concerned vertices, triangles and pixels. The programmable shading languages gave programmers more flexibility for designing shader and surface effects by distinguishing what quantities they compute and at what point they are invoked in the rendering pipeline. In RenderMan, Shaders have been categorized as Surface shaders, Displacement shaders, Light shaders, Image shaders and Volume shaders. The Programmable Shading language gives programmers basic types to use for manipulation of points, vectors, color geometry functions, access to the geometry state at the point being shaded, including position, normal, surface and the amount of incoming light. Parameters supplied to the shader are specified in the declaration of the shader or alternatively attached to the geometry itself. The output of the shader program will be the resulting color, opacity and possibly the surface normal and/or position on a particular point. The Programmable shading language gives programmers more control over the processing and appearance of these graphics primitives for designing the shader and surface effects. Following is an example for a simple plastic like material effect. [Apodaca 1999]

A shader program in RenderMan plastic.sl by Pixar is shown in Example 2.1.

```
Surface plastic(float Ka = 1;
                float Kd = .5;
                float Ks = .5;
                float roughness = .1;
                color specularcolor = 1;)
{
    normal Nf = faceforward (normalize(N),I);
    Oi = Os;
    Ci = Os * ( Cs * (Ka*ambient() + Kd*diffuse(Nf)) +
              specularcolor * Ks*specular(Nf,-normalize(I),roughness));
}
```

Example 2.1 Renderman plastic shader code fragment

By adjusting different diffuse colors, the plastic ball surface effect is shown in Figure 2.1

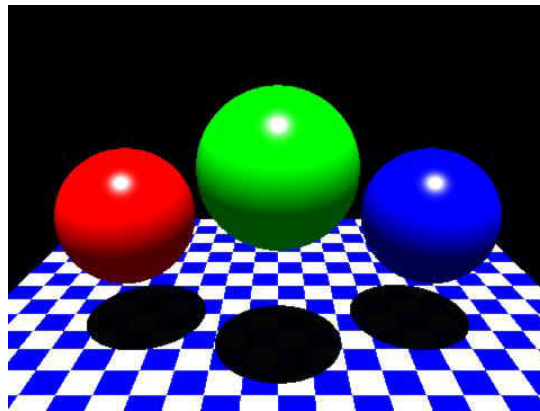


Figure 2 - 1 Plastic shader on the spherical surface

As we can see, Shaders and the shading language give developers and users a very realistic picture. However, by the time RenderMan was first introduced, a real-time interactive scene with shading was still impossible due to the amount of memory and fast processing needed.

Pixar's Photo realistic RenderMan [Pixar 2000] renderer was an implementation of the scan line rendering algorithm REYES (Renders Everything You Ever Saw)

architecture. REYES was developed with the goal of creating a rendering system for motion picture special effects. Cook first introduced REYES “The REYES Image Rendering Architecture” at SIGGRAPH 87 [Cook 1987]. This architecture was aimed at avoiding various flaws and constraint limitations, which they felt other algorithms like polygon z-buffer, polygon scan line and ray tracing, had. Examples of these flaws are Vast Visual Complexity, and speed and memory limitations. The resulting design brings a revolutionary new work in flexible shading and stochastic anti-aliasing and allowed renderers to produce photorealistic images.

The REYES algorithm is a geometry pipeline, and the shading procedure is done before the visibility (hiding) test. One of the advantages of this order is that it made displacement shading possible. However, the biggest disadvantage of shading before hiding is that if the scene is huge, with a large depth complexity, large numbers of geometry need to be shaded and then hidden by other objects closer to viewer. This drastically increased the rendering time and memory space needed.

An article in 1995, from Chapel Hill, the first interactive “Real-Time Programmable Shading” introduced a parallel architecture rendering pipeline, PixelFlow, which could achieve a real-time 30 frames/second interactive rendering rate. This work was contributed by Marc Olano [Olano and Lastra 1998]. As researchers work on software algorithms to improve the rendering rate, architecture designers are also working hard on their revolutionary graphics board designs.

A few years ago, all the transformation and rasterization algorithms had to rely on the CPU to produce rendered images. Over time, programmers learned to access the hardware provided graphics functionality through standard 3D programming interfaces, such as OpenGL or Direct3D. In the beginning, such expensive graphic cards were only available on expensive UNIX workstations. However, by Moore's Law, the hardware price reduced so much that low cost PC and game users could benefit as well. In 1998, a SGI Infinite Reality Server rendering 13 million triangles per second cost \$100,000, and by 2004, an NVIDIA GeForce 6800 Ultra rendering up to 600 million triangles per second, costs only \$500.

Graphics Pipeline

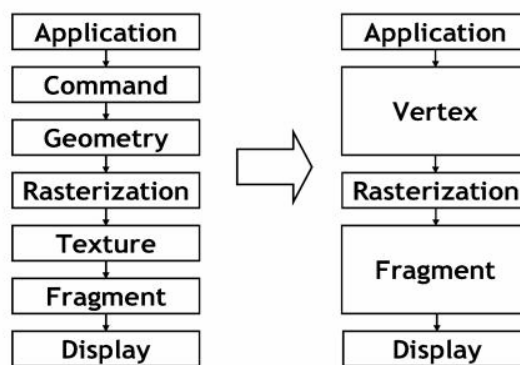


Figure 2 - 2 Graphics Pipeline

Figure 2.2 shows the traditional pipeline converted into the modern Graphics Pipeline with programmable shader, Vertex and Fragment shader. In this graph, vertex processors have replaced the geometry transformation and lighting and the fragment

processors have replaced the texture mapping processes. The real-time programmable shaders (vertex & fragment shader) on the new generation of graphics board replaced the fixed pipeline of the old boards. At SIGGRAPH 2001, a group from NVIDIA presented a paper on "A User-Programmable Vertex Engine". [Lindholm, et.al. 2001]. The paper discussed the assembling language instructions necessary to program their board (the NVIDIA GeForce 3). Because of the difficulty of programming with assembly language a higher-level language interface was needed. A paper was presented at SIGGRAPH 2001 from Pat Hanrahan's group on "A Real-Time Procedural Shading System for Programmable Graphics Hardware" [Proudfoot, et.al 2001]. Their shading language was a high level shading language based on the RenderMan shading language.

In 2002, NVIDIA came out with an improved version of their programmable graphics board that was programmable in the high-level language. Soon after, both ATI and 3D Labs came out with graphics boards with programmable shading capability. Each of these three companies came out with a high-level program language for programming their boards. NVIDIA developed Cg (C for graphics), which was fully supported on NVIDIA cards on both OpenGL and Direct3D API. Microsoft with ATI developed HLSL (High Level Shading Language), which was limited to the Windows platform. 3D Labs developed the OpenGL Shading Language (GLSL) front-end compiler according to GLSL from SGI, which was released as part of OpenGL 2.0 on Sep. 2004. The GLSL claims to be the standard shading language as the core part of OpenGL 2.0. More tests still need to be done. [OpenGL 2005]

By SIGGRAPH 2004, there were a few more shading languages announced. For example, Waterloo's SH Shading Library and Brook, which is a system for general-purpose computation on programmable graphics hardware. Brook extends C to include simple data-parallel constructs, enabling the use of the GPU as a streaming computer on graphics processors [Buck 2004].

In the following section, we will have a closer look at the three popular shading languages, Cg from NVIDIA, OpenGL shading language from 3DLabs, and High Level Shading Language from Microsoft.

2.2.2 C for Graphics from NVIDIA

June 2002, a C-like programming language Cg (C for Graphics) was developed and first released for the programmable graphical processing units from NVIDIA. During the 2002 SIGGRAPH conference, NVIDIA presented its Cg Toolkit version 1.0.1 with great success.

The Cg language is based on both the syntax and the philosophy of C [Kernighan and Ritchie 1988]. In particular, Cg is hardware oriented language and intended to be general-purpose (as much as is possible on graphics hardware). As in C, most data types and operators have an obvious mapping to hardware operations, so that it is easy to write high-performance code. Cg includes a variety of new features designed to efficiently support the unique architectural characteristics of programmable GPUs. Cg also adopts a few features from C++ [Stroustrup 2000] and Java [Joy et al. 2000], with an intention to be a language for small programs. Cg is most commonly used for implementing shading

algorithms, but Cg is not an application-specific shading language in the sense that the RenderMan shading language [Hanrahan and Lawson 1990] or the Stanford real-time shading language (RTSL) [Proudfoot et al. 2001] are. For example, Cg omits high-level shading-specific facilities such as built-in support for separate surface and light shaders. It also omits specialized data types for colors and points, but supports general-purpose user-defined compound data types such as structs and arrays.

Each shader would be called upon several times during the rendering process, i.e., generally speaking a vertex shader will be called once for each of the vertices on the scene and the fragment shader (or pixel shader) will be called once for every pixel in the screen. In the same way that a shader for RenderMan has to be called by a structure of a scene in a RIB file, a Cg shader must be called by a structure of a scene in a main program that draws to the screen. What we can see is that all the input information is the properties of vertices, such as position and normal of each vertex and light source. Additionally, the fixed matrixes also are provided as input for transformation from object coordinate to world coordinate etc. Those matrixes come from the main program that will load the shader. The output of the vertex shader will be the color and location of each vertex in object coordinate or world space coordinate etc. Those values will be the input of the fragment process on the pipeline.

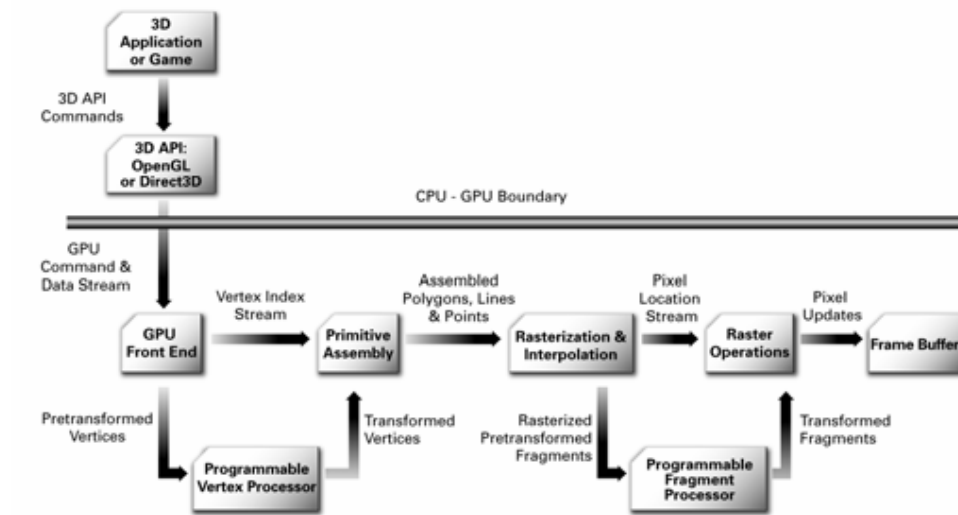


Figure 2 - 3 GPU – CPU Interface in modern Graphics Pipeline

Figure 2.3, which is borrowed from the Cg Tutorial book [Fernando, R., and Kilgard 2003], shows the GPU – CPU Interface in modern Graphics Pipeline. A 3D application program which contains a vertex shader and a fragment shader compiles and executes on both the CPU and GPU. The main program compiles and executes on the CPU and communicates with the graphics API (either OpenGL or Direct3D). The shaders are translated into assembly code. The main program calls the Cg run-time library, provided by NVIDIA and sits between the application and underlying graphics API, to execute the assembly code on the GPU. The calculation results from GPU are sent to the frame buffer.

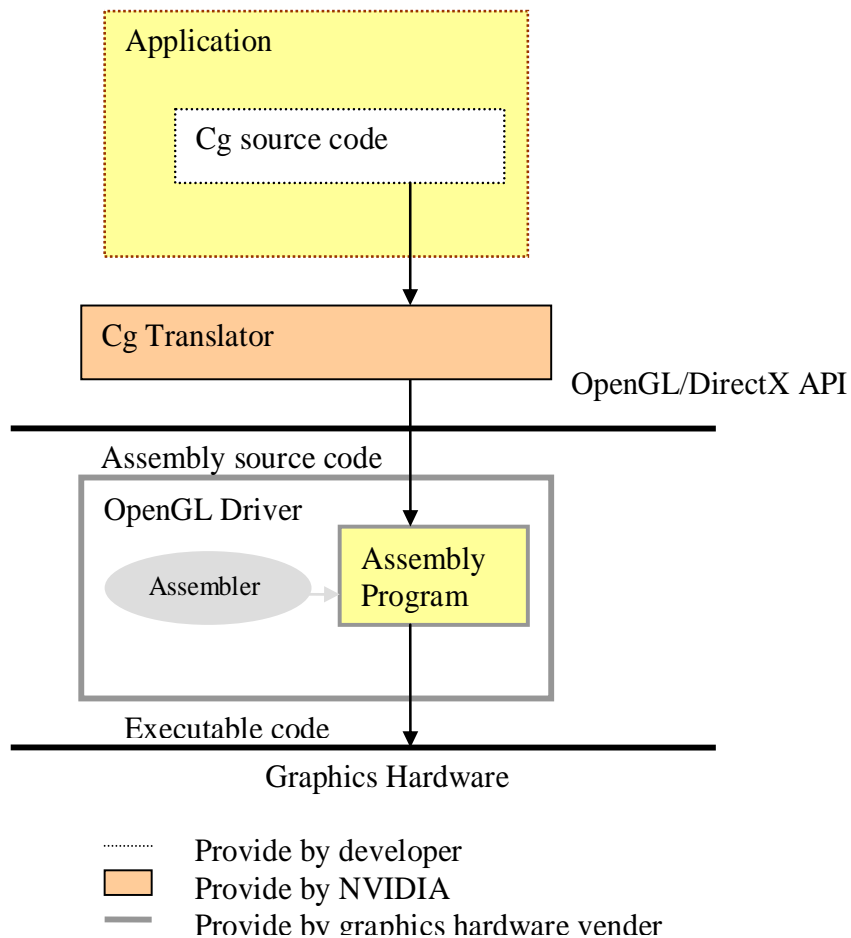


Figure 2 - 4 Cg Shaders loading process

Let's have a closer look at how a Cg shader is loaded into the executable environment. Figure 2.4 [Rost 2004] illustrates how does a Cg shader is loaded and handled by the execution environment. Compared to GLSL, Cg is designed as a source code to source code translator. The Cg compiler is really a translator that is outside of the OpenGL or Direct3D API compared to a GLSL driver that is in the OpenGL driver. The Cg program really compiles to assembly code vertex shader and fragment shaders in OpenGL or Direct3D. Advantage of Cg is that the translation can be done offline. But, the assembly code has to be parsed and assembled at execution time.

For a better understand of the Cg shading language, let's look at a simple example of a vertex shader for brick shown as example 2.2.

```
// Brick-vert.cg vertex shader of brick in Cg

const float3 LightPosition = float3(0.0, 0.0, 4.0);
const float specularContribution = 0.3;
const float diffuseContribution = 0.7;

void main(
    out float LightIntensity,
    out float2 MCposition,
    in float4 gl_Normal : NORMAL,
    float4 gl_Vertex : POSITION,

    out float4 gl_Position : POSITION,

    //out float4 Color0 : COLOR0,
    uniform float4x4 ModelView,
    uniform float4x4 gl_ModelViewProjectionMatrix,
    uniform float4x4 ModelViewIT
){
    float4 ecPosition = mul(ModelView , gl_Vertex);
    float3 tnorm = normalize(mul(ModelViewIT , gl_Normal).xyz);
    float3 lightVec = normalize(LightPosition - ecPosition.xyz);
    float3 reflectVec = reflect(-lightVec, tnorm);
    float3 viewVec = normalize((float3)(-ecPosition));
    float spec = max(dot(reflectVec, viewVec), 0.0);
    spec = pow(spec, 16.0);
    LightIntensity = diffuseContribution * max(dot(lightVec, tnorm), 0.0) +
        specularContribution * spec;
    MCposition = (float2)(gl_Vertex);
    gl_Position = mul(gl_ModelViewProjectionMatrix , gl_Vertex);
}
```

Example 2.2: Vertex Shader of Brick in Cg language

Look at the parameter area in the main function, there is few parameters with the modifiers “in”, “out”. In the end of some parameter, there is a “:” with some capital letters which are called semantics in Cg. What are these decorations for? The modifier “in”, “out” indicates this parameter is a connector of main application with the vertex shader or from the vertex shader out to the fragment processor. The semantics tells which graphics register this parameter will flows into. The semanticed variable is a connector between shader programs to graphics API.

2.2.3 OpenGL shading Language

Three years ago, SGI started the definition of the OpenGL Shading Language, which claimed to be a powerful high-level, hardware independent shading language, the OpenGL Shading Language (GLSL). This language enables direct compilation of C-like programs to graphics hardware machine code, creating enormous opportunities for compiler and graphics architectural innovation and bringing real-time realistic rendering a step closer to reality. [OpenGL 2005]

OpenGLSL has been designed to allow application programmers to define processing at those programmable points of the graphics pipeline. Two shaders in the pipeline are vertex shaders and fragment shaders. Both are independently compilable units separated from the main program. Like the Cg shading language, the OpenGL Shading Language is based on ANSI C with extended vector and matrix data types which make it more concise for the typical operations carried out in 3D graphics. Before the OpenGLSL can be supported by the OpenGL standard, it does have to have a set of extensions that are supported by a number of graphics card vendors.

If there is an OpenGL application, we should disable the original rendering fixed pipeline function and add the programmable shader into the rendering pipeline. Built-in access to the existing OpenGL state keeps the API entry points the same as developer are used to. Another benefit from GLSL is that shader compiles during runtime. The major difference GLSL has with other formats of shading languages is that it allows GLSL to

be compiled inside of OpenGL rather than compiled with individual hardware vendors. This could bring optimal performance on the graphics hardware. [Rost 2004]

OpenGLSL is getting closer to becoming the cross-platform standard for all shading languages. The GLSL is included as a subset of the release of OpenGL2.0 on Sep. 9th 2004. This also depends on if GLSL will become so dominant so that every hardware vendor likes to develop the back end compiler for it.

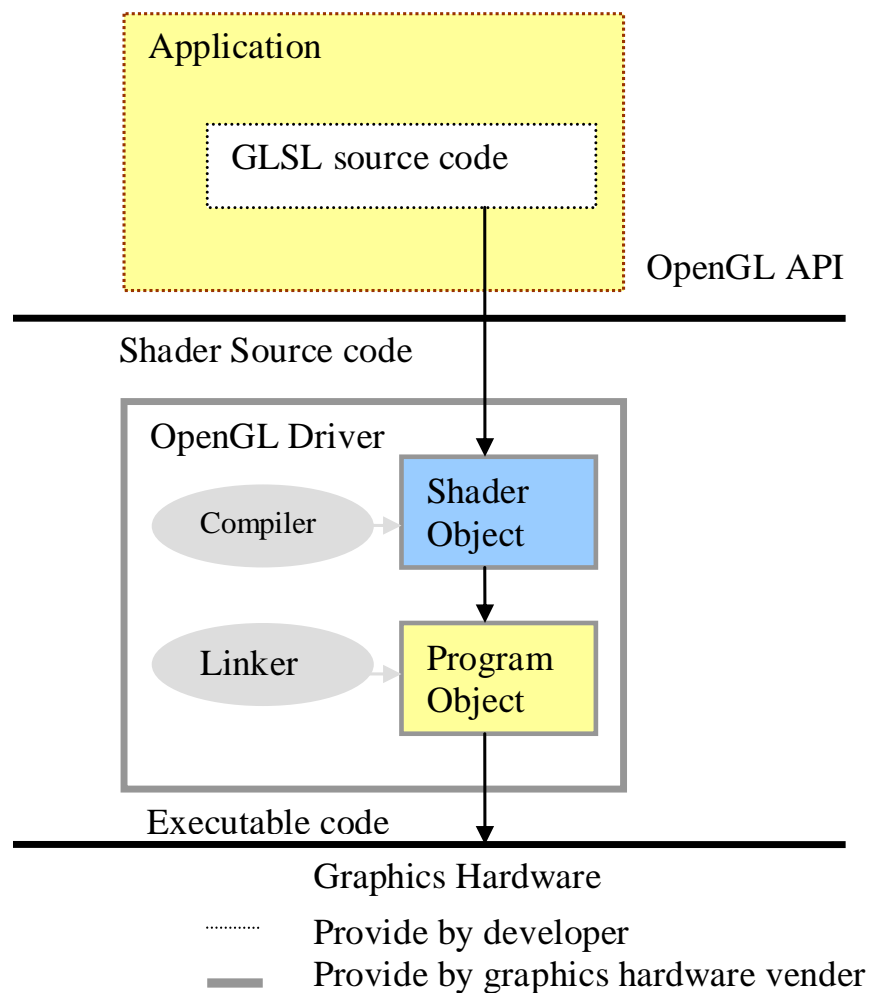


Figure 2 - 5 GLSL Shaders loading process

Let's look at how the GLSL shader is loaded by the main program and executed on the GPU. Figure 2.5 illustrates how the OpenGL shaders are handled in the execution environment of OpenGL. Applications communicate with OpenGL functions by calling functions in the OpenGL API. An OpenGL function "glCreatShaderObjectARB" allows a shader data structure to be created within the OpenGL driver. Then a character string is sent to the OpenGL API to indicate the name of the shader program. 3DLabs released their GLSL front-end compiler. This front-end compiler tokenizes the shader string and produces a binary, high-level representation of the language. The back end of the compiler has to be implemented individually by different hardware vendors. This should be typically packaged inside of the hardware driver. Of course, hardware support for GLSL depends on each of the hardware vendor's willingness to implement the compiler for GLSL.

```
// GLSL
const vec3 LightPosition=vec3(0.0, 0.0, 4.0);
const float specularContribution = 0.3;
const float diffuseContribution = 0.7;//0.7 = 1.0 - specularContribution

varying float LightIntensity;
varying vec2 MCposition;

void main(void)
{
    vec4 ecPosition = gl_ModelViewMatrix * gl_Vertex;
    vec3 tnorm      = normalize(gl_NormalMatrix * gl_Normal);
    vec3 lightVec   = normalize(LightPosition - vec3 (ecPosition));
    vec3 reflectVec = reflect(-lightVec, tnorm);
    vec3 viewVec    = normalize(vec3 (-ecPosition));
    float spec      = max(dot(reflectVec, viewVec), 0.0);
    spec            = pow(spec, 16.0);
    LightIntensity  = diffuseContribution * max(dot(lightVec, tnorm), 0.0)
    + specularContribution * spec;
    MCposition      = vec2 (gl_Vertex);
    gl_Position     = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Example 2.3 A vertex shader for brick in OpenGL shading language.

Next, example 2.3 shows the OpenGLSL shader. One of the differences between Cg and OpenGLSL is that the connector which glues the shader with the underlying graphics APIs. In GLSL, a set of built-in variables does the gluing job. They have prefix of “gl_” for example, “gl_ModelViewMatrix” is a uniform matrix passed in from main program; and “gl_Vertex” is a variable for the position of each vertex. These variables are needed during vertex shader is executed. Compare to Cg shader, In GLSL the matrix variables for transformation were predefined and can be used directly in shader program. On the other hand, Cg shader takes care of the connection by some developer defined variables binding with a set of Semantics, i.e. POSITION, COLOR etc. and the uniform matrixes which passed in from main program.

On the other hand, in Cg, the “gl_” are not predefined. In GLSL, the inputs of the vertex shader are vertex information such as normal, color, texture coordinate and so on. Data flows into vertex shaders via attribute built-in/user-defined variable, uniform built-in/user-defined variable, and built-in/user-defined varying variable, as well as special vertex shader output variables will pass the information to be rasterized and then to the fragment shader on the next step of the rendering pipeline.

More details of the OpenGL Shading Language functionality and data type's built-in/user defined variable and functions will be discussed in detail in chapter 3.

2.2.4 High Level Shading Language

High Level Shading Language (HLSL) is subset of DirectX 9 that allows shader developers to write programmable shaders on the Windows platform. HLSL also has all

of the usual advantages of a high level language such as easy code reuse, improved readability and the presence of an optimizing compiler. However, this language is limited by the platform dependency on Windows platforms.

Example 2.4 is a vertex shader for a wood shader written in HLSL. This code is from [Peeper and Mitchell 2002].

```
// HLSL shader of wood
float4x4 view_proj_matrix;
float4x4 texture_matrix0;
struct VS_OUTPUT
{
    float4 Pos : POSITION;
    float3 Pshade : TEXCOORD0;
};
VS_OUTPUT main (float4 vPosition : POSITION)
{
    VS_OUTPUT Out = (VS_OUTPUT) 0;
    // Transform position to clip space
    Out.Pos = mul (view_proj_matrix, vPosition);
    // Transform Pshade
    Out.Pshade = mul (texture_matrix0, vPosition);
    return Out;
}
```

Example 2.4 A vertex shader for wood in High-Level shading language.

In Example 2.4, the first two lines declare a pair of 4×4 matrices called `view_proj_matrix` and `texture_matrix0`. Following these global-scope matrices, a structure is declared. This `VS_OUTPUT` structure has two members: a `float4` called `Pos` and a `float3` called `Pshade`. The main function for this shader takes a single `float4` input parameter and returns a `VS_OUTPUT` structure. The `float4` input `vPosition` is the only input to the shader while the returned `VS_OUTPUT` struct defines this vertex shader's output.

Looking at the code of the main function, you'll see that a function called "mul" is used to multiply the input `vPosition` vector by the `view_proj_matrix` matrix. This

function is commonly used in vertex shaders to perform vector-matrix multiplication. Following the transformation of the input position `vPosition` to clip space, `vPosition` is multiplied by another matrix called `texture_matrix0` to generate a 3D texture coordinate. The results of both of these transformations have been written to members of a `VS_OUTPUT` structure, which is returned. Those values output from the vertex shader are interpolated across the rasterized polygon and are available as inputs to the pixel shader. In this case, `Pos` is an output from vertex shader and an input for the fragment shader. `Pshade` is passed from the vertex to the pixel shader via an interpolator.

We discussed each of the shading languages and gave examples for each of them. Interestingly, the public release of the NVIDIA Cg system was done concurrently with the design and development of similar systems by 3Dlabs OpenGLSL, and Microsoft HLSL. There has been significant cross-pollination of ideas between the different efforts and they are very similar in structure [Rost 2004]. We will discuss detail of these similarities and differences between these systems throughout chapter 3 Design for the shader converter. In our design, we will try to keep the advantage of those three languages and keep the design and implementation as simple as possible.

The OpenGL 2.0 specification, which includes the OpenGL Shading Language specification, was released as the new OpenGL standard. This cleared the way for graphics vendors to ship OpenGL drivers that support the industry's first open standard, high-level shading language. However there are still many programmers that are not familiar with OpenGLSL, instead, they are used to Cg structure. At the same time, the

older version of graphics card, which only supports Cg or HLSL, will be out-of-date for the new standard shading language. So, how to avoid these cards from becoming trash? Our design of a shader converter will be one of the answers. The converter proved the concept of converting the shaders from one language format to another shading language format. And the converted shader would be able to load from the main program, execute and display.

With the rapidly developing speed of the GPUs and shaders, some work has been done on embedding shaders into web 3D applications. In next two sections, we are going to briefly introduce the history of the most popular web 3D application formats VRML/X3D, the prior works on embedding shaders into X3D and the problem of sharing shaders via the Internet.

2.3 X3D: The next generation of VRML97 in XML

VRML (Virtual Reality Modeling Language) was first introduced in 1995 and its increased development and wide use over the past decade has made it the industry standard for 3D web. X3D is the next generation of VRML in XML. An Internet survey [googleHit 2004] of various web3D technologies showed the usage of VRML/X3D is more widely used for web 3D. This is one of the reasons VRML/X3D has been chosen as part of this dissertation research work.

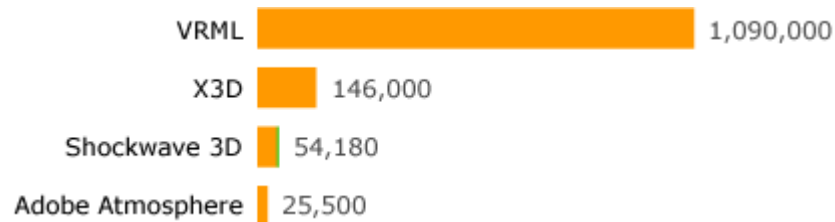


Figure 2 - 6 Google Hits of 3D websites

Tim Berners-Lee first used the term Virtual Reality Markup Language (VRML) when he discussed a need for a 3-D Web standard during a European Web conference in 1994. Soon afterward, an active group of artists and engineers formed a mailing list called `www-vrml`. They changed the name of the standard to Virtual Reality Modeling Language to emphasize the connection to graphics. The result of their efforts was to produce the VRML1.0 specification [web3D 2005]. As a basis for this specification, they used a subset of the Open Inventor file format from Silicon Graphics created in 1992. [Open Inventor 2005]. The VRML1.0 standard was implemented in several VRML browsers, but it only allowed you to create static virtual worlds. This limitation reduced the possibility of its widespread use. Clearly, the language needed a robust extension to add animation and interactivity. The VRML 2.0 standard was developed by the year 1997 and was adopted as the International Standard ISO/IEC 14772-1:1997. Since then it is referred to as VRML97. A book “VRML2.0 Sourcebook” gives an overview of VRML97 and how to use it [Ames. et.al 1997].

In the following paragraph, we will give few examples in VRML and X3D format to show how to create a VRML/X3D file and the tools or plug-ins required for displaying VRML/X3D files.

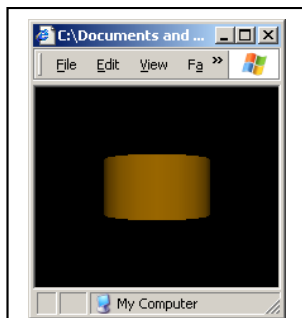
For creating a virtual world with VRML, a developer will need to have a VRML browser and a text editor. Like building a house or office building, the VRML file serves as a blueprint for the virtual world. Developers create this virtual world by specifying and organizing the structure of the VRML world in a scene graph schema describing how to build shapes, where to put them, and what color or texture their surfaces should have. Displaying this blueprint in the VRML browser will be much more attractive than the traditional 2D printout, because what the user will see is a 3D interactive virtual reality world displayed via the Internet. An early VRML browser developed by SGI was Cosmo Player. Cosmo World was an interface for creating VRML worlds which turned out to be a very useful API, and was further developed and made available from many companies including Cosmo Software, SGI (Silicon Graphics), and Platinum. Unfortunately SGI stopped supporting its PC version. You can still download a copy of the player for IRIX. [SGI 2005]

Other successful VRML browsers are Blaxxun Contact by www.blaxxun.com, Cortona from ParallelGraphics, Flux from MediaMachines.com and Octaga from Octaga.com which are commercial browsers. Some open sources VRML browsers are FreeWRL for Linux/Mac, and the Xj3D active project from web 3D consortium that was developed in Java. Most active browsers can be found at the web3D consortium website [web3D 2005].

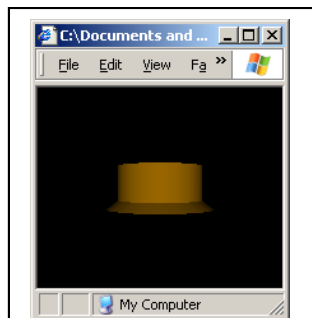
```

#VRML V2.0 utf8
Group{
  children[
    # draw first object
    Shape{
      appearance DEF Brown Appearance {
        material Material {
          diffuseColor 0.6 0.4 0.0
        }
      }
      geometry Cylinder {
        height 2.0
        radius 2.0
      }
    }
    # draw second object
    Transform { # Transfer 2nd shape
      translation 0.0 2.0 0.0
      children Shape{
        appearance USE Brown
        geometry Cone {
          height 2.0
          bottomRadius 2.5
        }
      }
    }
  ]
}

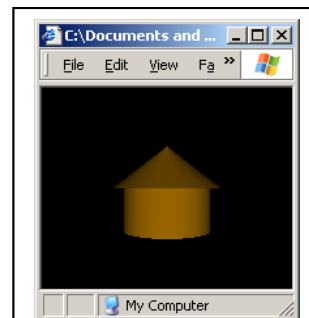
```



Draw first shape



Draw second shape

Transfer 2nd shape

Example 2.5 simple shapes built in VRML step by step

For a better understanding of the VRML language structure, example 2.5 presents each step for building a simple shape in VRML.

Besides the basic structure, VRML also supports animation by defining a route, texture mapping and other extensible functions by creating new nodes for the world.

VRML is an easily mastered script language. Detailed explanations about the route and extensible node design will be given in chapter 4.

As the Internet rapidly developed in the late 1990s, the platform independent eXtensible Markup Language (XML) was adopted by the W3C [W3C 2005]. XML gives structural rules for developers to extend systems by defining their elements in a flexible way. An XML document is structured with elements and attributes. A developer defined Document Type Definition (DTD) provides agreements for what elements and attributes are valid to be used in developer owned programs. XML has become the standard language format for transferring data on the Internet. X3D was developed as VRML defined in the XML format, i.e., the next generation of VRML97. By 1999, San Ramon from the web3D consortium initiated the process to define X3D. It was decided that X3D would be developed in XML. In Figure 2.7[web3D 2005], we can see the developmental history of VRML/X3D from 1997 to 2002.

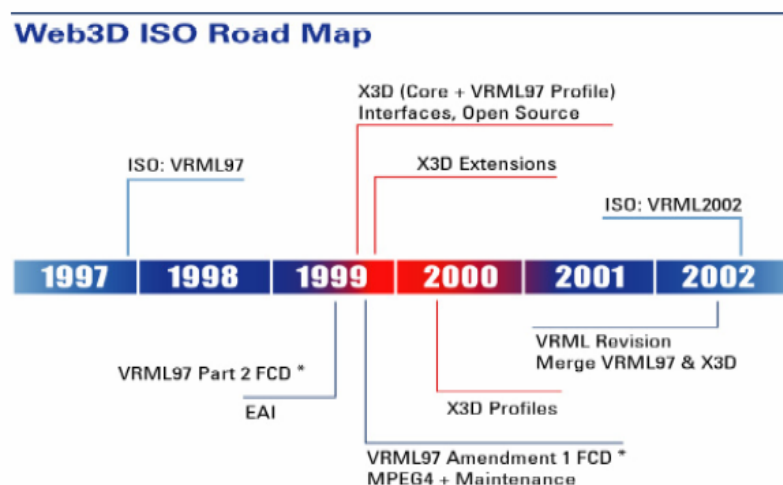


Figure 2 - 7 X3D/VRML Growing history


By SIGGRAPH 2004, X3D immersed as the standard web3D format on the Internet [web3D 2005]. Hopefully soon Internet browsers will be embedded with X3D plug-ins as standard and users will be able to view X3D scenes without the additional download each time they load a web3D scene.

Next, lets see how the same shape in example 2.6 built in X3D with detail commented.

```

<X3D> # header ...#
  <Scene>
    <Group>
      <Shape> # draw 1st shape
        <Cylinder height='2.0' radius='2.0' />
        <Appearance DEF='Brown' />
        <Material diffuseColor='0.6 0.4 0.0' />
      </Appearance>
    </Shape>
    <Transform center='0.0 0.0 0.0' translation='0.0 2.0 0.0' /> # transform 2nd shape
      <Shape> # draw 2nd shape
        <Cone height='2.0' bottomRadius='2.5' />
        <Appearance USE='Brown' />
      </Shape>
    </Transform>
  </Group>
</Scene>
</X3D>

```



Draw 1st shape draw 2nd shape transform 2nd shape

* This scene is displayed in Flux 1.1 browser from mediamachines.com.

Example 2.6 A step by step example of a simple X3D scene

The format differences between VRML and X3D can be shown by the following example. In VRML you define a viewpoint like this:

```
DEF MyView viewpoint {position 0 0 1}
```

In X3D, the same viewpoint is defined as a tag with few attributes:

```
<viewpoint DEF='MyView' position='0 0 1' />
```

Example 2.6 a simple viewpoint definition in both VRML and X3D

An X3D program can be implemented by using XML parsers. The XML format allows X3D to become an interchangeable web 3D format on the Internet. There are tools for generating X3D scenes, like Xena from IBM [Xena 2005]. The Xj3D group also worked on this, [Xj3d CVS 2005] their CVS parser was available on the web3D website. Mediamachines released their X3D browser Flux [Flux 2005], which allows the X3D world to be visualized in web browser.

For creating X3D scenes, developers can either use tool kits like Xena from IBM or export the X3D format from 3D software applications like AutoCAD, Blender, or H-Anim. Also, 3D Studio Max and Maya support the exporting to VRML format. Updated information can be viewed on the web3D web page [web3D 2005].

2.4 The Challenge of Real-time Procedure Shaders in VRML/X3D

The reason VRML/X3D does not produce an acceptable realistic rendering is because VRML/X3D browsers calculate lighting using the Gouraud shading algorithm.

The advantage of the Gouraud shading algorithm is that its rendering time is much less than other shading algorithms, such as Phong shading. The Phong shading algorithm gives a more realistic visual effect for specular highlights. When programmable shaders became available in 2002, developers started working on ways to bring this new technique into VRML/X3D. By SIGGRAPH 2003, some significant success was gained and the possibility of using embedded shaders in the VRML/X3D was explored. [De Carvalho 2003][Parisi 2003]

Figure 2.8 illustrates a single machine browser showing a Phong model shader and a reflection environment mapping shader [De Carvalho et al. 2004], respectively.

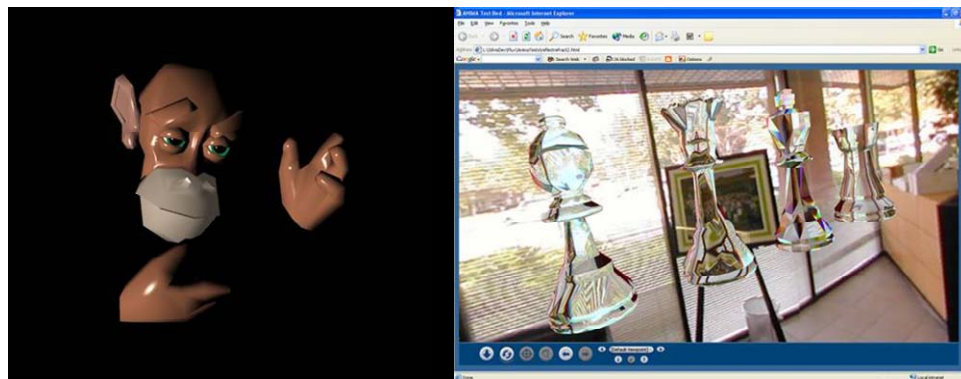


Figure 2 - 8 Shaders in VRML/X3D web browser on single machine

The most important feature of X3D is that it is a 3D web language for 3D applications on the Internet. The previous examples however, were limited to a single PC and no sharing over the Internet was demonstrated. So the question we are faced with is whether or not it is possible to move shaders to the Internet and share real-time, interactive, and realistic environments.

As we have previously discussed, the different shading languages from different vendors have yet to achieve platform independence at this time. This is a serious problem if we want to share data over the Internet that includes shaded models and/or environments. The compatibility of these shaders among each other is illustrated in Figure 2.9. The label “Y” in the graph means fully compatible and “N” means not fully compatible.

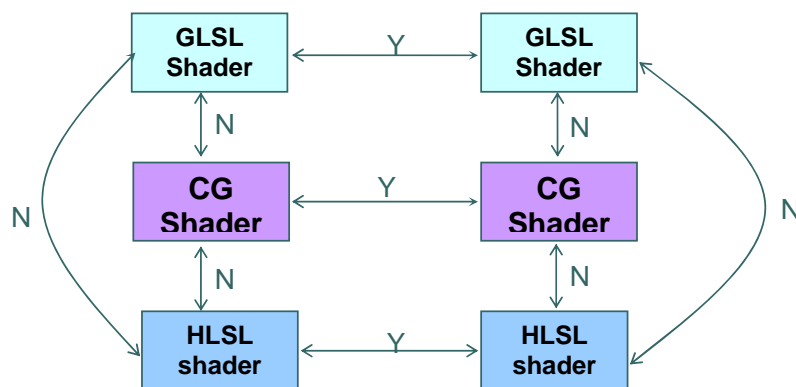


Figure 2 - 9 Different Shaders format compatibility

During SIGGRAPH 2004, a Bird of Feather (BOF) meeting of the X3D shader group presented a solution of solving the non-compatible hardware problem. The solution was to provide multiple formats of the same shader. When the X3D file arrived at the local machine the browser would determine which type of code it should be compiled as.

There are some inherent problems with this proposal: first, for one shader effect to be used on the web, a programmer has to learn three shading languages, duplicate the shader in three formats and have three different machine platforms to test whether the shader is running correctly before it can be published. This proposal is not a very

practical solution. However, it proved the equivalent functionality between Cg, GLSL and HLSL. This provided an assumption for our converter design. The assumption was that as long as we can convert the shader between the different formats, display it on different graphics cards and platforms it will be correctly handled by the VRML/X3D browsers plug-in. At this time that only Bitmanagement Contact version 6.2 VRML/X3D [Bitmanagement 2005] web browsers plug-in, correctly supports Cg and HLSL shaders in DirectX9.0. It is believed that more vendors will support shader functions, enabling a better future for 3D web applications. Other evidence that supports our assumption is that in March 2005, 3DLab announced a shader tool called “ShaderGen” [ShaderGen 2005]. This tool aims to show that programmable hardware can be used to obtain the same rendering results as the OpenGL fixed function path. According to 3DLab the utility *“allows you to set the parameters for fixed function rendering through a convenient GUI. With the click of a mouse, ShaderGen creates an OpenGL shader that will produce the same result as the fixed function state you've set.”* [3DLab 2005] What we surmise from this description of the new tool is shader writing in OpenGL shading language is equivalent to fix functions in the graphics pipeline. This is exactly the same purpose as Cg was pushed out. *“It replaced the part of the fixed functions in the graphics pipeline”*.

In this dissertation, we assume that all the different formats of shaders have equivalent functionality in different language formats. We propose a solution with a XML middle layer converter for making the shader platform independent and sharable via the Internet. The shader converter reads different formats of shaders written by shader developers in arbitrary shading language (currently, Cg, GLSL, and HLSL) and converts

them to the XML format before the X3D world is published. After the X3D world was downloaded, the shader is converted to the corresponding client machine supported shader format by the converter and displays the equivalent shading result on the client's machine.

Imagine the graphics pipeline as a product producing assembly line. If one of the pieces of the assembly line breaks down, a functionally equivalent piece needs to be put back in its place for the machine to operate properly. The same is true for the graphics pipeline. Let's look closely at the graphics pipeline with different shader formats. We try to find the differences and similarities among different formats of shaders. All of the vertex and pixel shaders are located in the same positions in the pipeline. Vertex shaders are located between loading vertices from the main application and outputting corresponding transformed object/world coordinate vertices back to the main application. Fragment shaders are located between loading and textureing colors and drawing with rasterizing colors and textures down to each pixel. Figure 2.10 shows a high-level overview of different format shaders that are located almost same spot in the graphics pipeline and contain both vertex and fragment shaders.

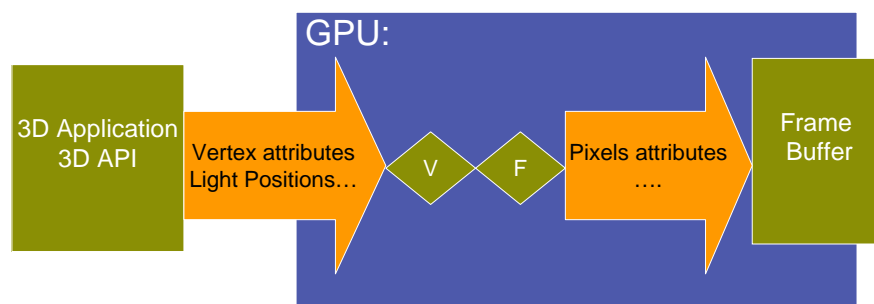


Figure 2 - 10 High-level overview of the Vertex/fragment shader in the graphics pipeline

In section 2.2.2 – 2.2.4, we had a close look at the detailed structure of how each of the shader formats communicates with the graphics API. We also know that both GLSL and Cg or HLSL vertex/pixel shaders are located at the same position in the graphics pipeline except for the connector with a little different gluing method (Cg/HLSL Semantic uses an “in”, “out” modifier and GLSL uses built-in variables). This means they still have an equivalent functionality as part of the programmable graphic pipeline. So if we can convert their skin to the different corresponding platform supported format, the different formats of shader will be shareable in web3D applications. We assume X3D/VRML can load different individual shaders successfully.

3 Design and Implementation

3.1 A Framework of Platform Independent Shaders in X3D

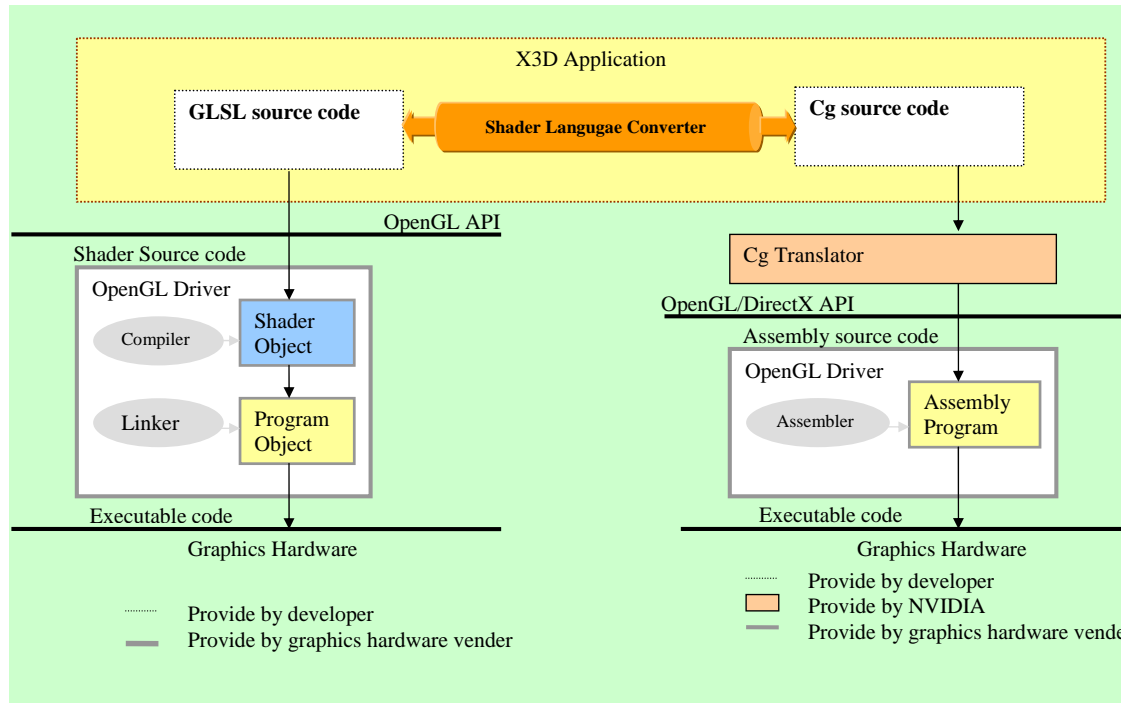


Figure 3 - 0 SLC convert shader skin before loaded to API

Let's go back and look at how the GLSL and CG /HLSL shaders have been loaded into the main application in Figure 2.4 and Figure 2.5. Once the Cg shader has been loaded, it will be translated to assembly code. On the other hand, when the GLSL shader is loaded, it will compile until it gets connected with the OpenGL API without the need to be translated into assembly code. Clearly they went to different pipelines. This means we can not control them once it loaded and parsed to the Graphics API. At the same time, we realized there is so much similarity in syntax and functions etc. among these three languages that we decided do some work before it reaches the API. That's

means we will do a conversion between the languages with a Shader Language Converter (SLC) shown in Figure 3.0. Thus the SLC functions as a shader translator among different languages, therefore making the shaders platform independent.

The next step was to design a framework for fitting the independent platform real-time shaders into the X3D scene. This framework includes a back-end core shader language converter (SLC), which translates shaders between different shading languages, and a library containing a basic set of shaders. In chapter 4, this framework will be demonstrated with an example of a shaded X3D scientific visualization of protein structures using data gathered from x-ray crystallography. This section will focus on the shader converter design and a brief introduction of the shader library. Figure 3.1 shows the converter framework.

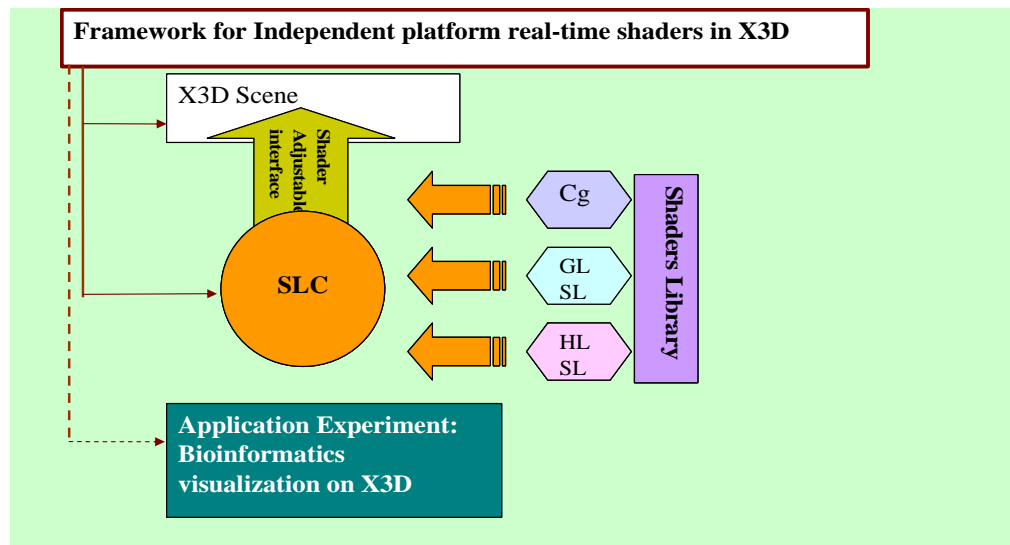


Figure 3 - 1 a Framework of Platform Independent Shaders in X3D

The problem of multiple shader formats being gradually released on the market by different vendors appears to be one of the most important issue for shaders being applied in X3D on the web. The shader language converter translates shaders between different shading languages. It is a core for the function of sharing independent shaders on the Internet. The shader language converter takes one of the most important roles of this entire framework. In section 3.2, we are going to discuss the detail design and implementation of this converter.

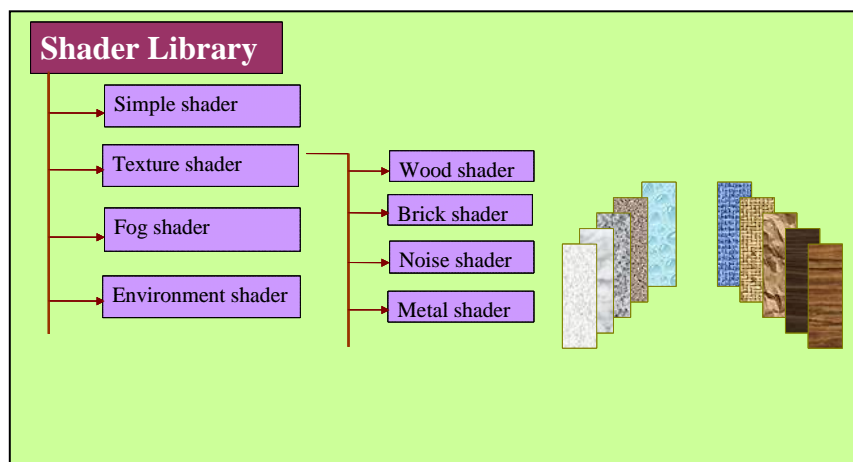


Figure 3 - 2 illustration of shader library

We collected all of the shaders used during the SLC implementation and bioinformatics visualization tests in chapter 4. We then set up a library with this basic set of shaders for different materials, like wood, brick, noise bump texture, and Phong shading and Blinn-Phong shaders for plastic materials. The Phong shader is used for defining different elements in the protein structures.

The purpose of design this library is to give shader users a chance to share their platform independent shaders and give them space to communicate both shader design

and bug reports in the SLC. The more shader developers use the SLC, the more useful shaders will be tested, the more bugs found and fixed, the more shaders added to the library. When the SLC has finished its testing period, there will be a much larger set of useful shaders added to the library by developers and the more general user can use them more efficiently.

This shader library contains some successfully converted shaders from different languages. All shader examples used in this chapter or later chapters are all from the shader library. Please refer to the appendix A for shader examples.

Beside the shader converter and shader library, another important part of this framework is a shader supported VRML/X3D browser needed for displaying the shaders. At this time, the only successful shader supported X3D/VRML browser is Bitmanagement BS contact version 6.2. The only support shader format at this time is in Cg/HLSL in DirectX9.0. [Bitmanagement 2005]

3.1.0 Initial ideas of embedding Shaders into X3D

Initially, our design for embedding shaders into X3D was trying to define a new set of nodes for the shaders used in X3D. These shader nodes simply contain a set of shaders that are connect to the X3D/VRML file by calling shader names with a few parameters and function names. A connection interface contains parameters that can be defined by the user. The Shader could also be external to the VRML/X3D main program. The benefit of this design is that it simplifies the interface for both the developers and

users. The conversion among different shading languages is handled by SLC and hidden from users.

The usability of the final user interface for implementing shaders in X3D takes a key role in the design. A good, convenient and consistent interface will give us a simple, friendly test-bed and possibly add to the longevity of the application.

The basic idea for the initial design is shown in the following simple example of an X3D file:

```
X3DShaderNode : X3DNode
{
SFNode [in,out] metadata NULL X3DMetadataObject]
.....}
```

```
//X3D Application e.g.: A attribute of a sphere surface
Appearance {
  Shader = "wood3";
  parameter1 = value1;
  Parameter2 = value2;
  Parameter3= value3;
  .....}
```

Example 3.1 Illustration of initial design idea

The advantage of this design was that it has a simple user interface. The disadvantage was that for complex shaders, especially for the shaders that need to write the texture out dynamically for texture mapping it might cost more time for updating the shader file. In that case, we have to define an interface for the browser to dynamically send events in/out with the intermediate output.

Due to the lack of source code for the VRML plug-in, we are limited in our design of the connector interface to a very basic level. At the same time, the X3D shader group is working on standardizing the interface for shader embedding into X3D. They also pointed out the multi-shading language non-exchangeable problem in the web3D symposium Oct. 2004[web3D-Shader-Group 2005]. We switched our direction to focus more on designing the shader language converter to solve the shaders' platform dependent problem. We designed an XML middle layer Shading Language Converter (SLC). We believe the only task needed for matching the XML layer to the X3D shader nodes once the X3D shader group defines their standards is to match our XML nodes to the new X3D's standard definitions.

This converter was called the "Universal Converter" in the SIGGRAPH 2004, Web Graphics presentation "Universal Converter for platform independent shader in X3D" [Liu 2004]. Later we changed the name to "Shader Language Converter" (SLC) as suggested by Dr. Owen and Dr. Don Brutzman.

3.2 Shader Language Converter Design

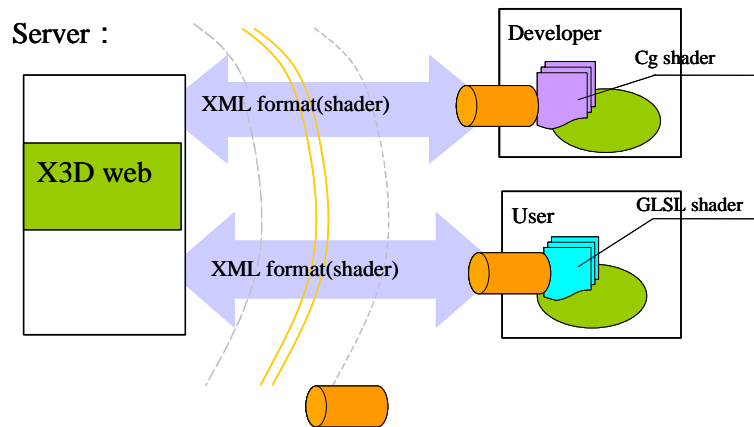


Figure 3 - 3 Shader Language Converter in X3D application

The X3D application published and shared via the web. For example, developer “A” is working on a windows platform and develops his shader in HLSL, which is converted by the SLC into the XML format before it is published to the X3D format. User “B” working on a Linux operating system with a 3DLab wildcat graphics card which best supports GLSL, receives this file and after the SLC is notified that the hardware supports GLSL, converts this XML formatted shader to the GLSL 3DLab card supported GLSL format. The equivalent shading result should be achieved.

The basic structure of the SLC is to first read any of the natural shading languages (Cg/GLSL/HLSL), and then convert it to the XML format before the X3D files are published. The SLC on the client side browser converts the XML format shader to the client’s machine compatible shader format. The structure is shown in Figure 3.4.

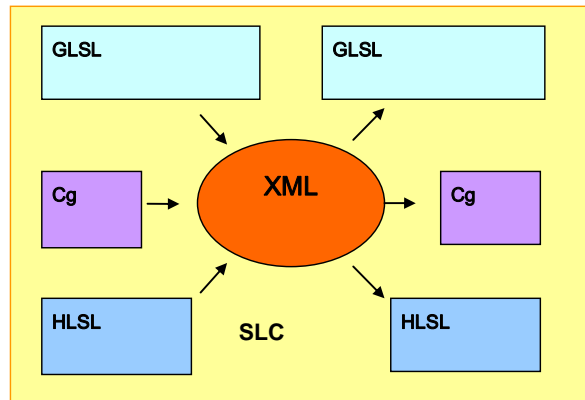


Figure 3 - 4 Structure of Shader Language Converter

3.2.1 Why the XML middle layer?

People might ask, why bother use the middle XML layer, why not write a wrapper and wrap the code like Cg shader up and output it in GLSL / HLSL format. When one calculates how many conversions will need to be written, we see that for the current three shading languages, 12 conversions are required. This is illustrated in Figure 3.5. On the other hand, if there is a XML layer, the conversions required are 3 shading language formats for input and 3 formats for output. So the total number of conversions is only 6.

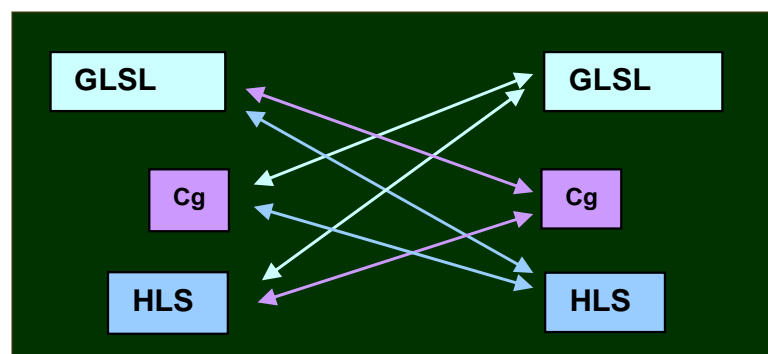


Figure 3 - 5 Comparison of the SLC design structure with/without XML layer

There might be new shading languages with special platform support appearing on the market, in that case we have to add these new shading language conversions to the SCL structure so that all existing clients can compile and show these new shaders. At the same time this new platform will be able to display equivalent shaders from other formats. If one new shading language appears on the market and needs to be added into the conversion framework, the only conversions needed to be added into the XML layer structure is from the new shader format to the XML middle layer and from XML middle layer to the new Language format. The total extra conversions needed for each new added language will be 2, shown in Figure 3.6. However, adding a new language in the structure without XML layer, the conversion will be from each existing shading language to the new language and from new language to each of the old language for a total of six conversions, shown on Figure 3.7.

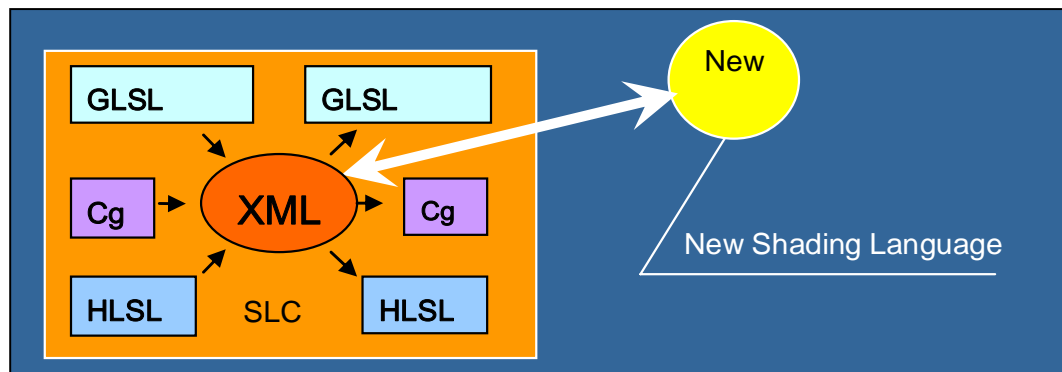


Figure 3 - 6 new shading language on the SCL structure design with XML layer

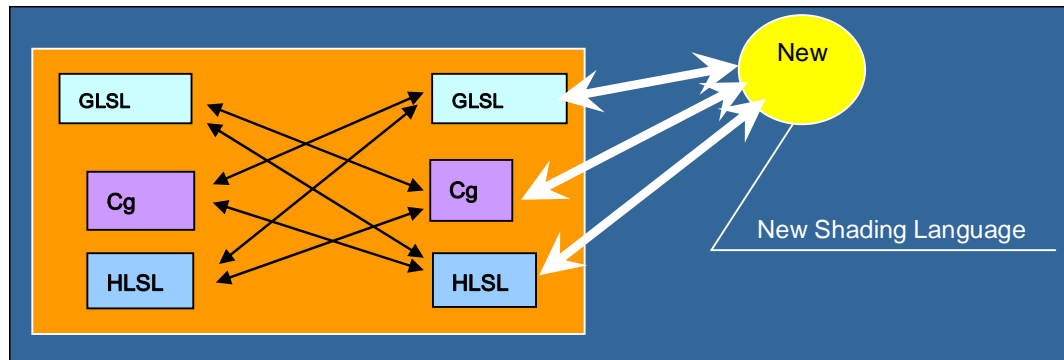


Figure 3 - 7 a new shading language on the SLC design structure without XML layer

Overall, using XML as a middle layer design has several advantages. They are,

1. For every new shading language added to the converter, a constant number of conversions (2, 2, and 2...) need to be added in SLC with XML middle layer. On the other hand without the XML middle layer the number of conversions increases linearly (6, 8, and 10...) with each new language.
2. The XML middle layer format is easily embedded into X3D standard because X3D is designed as an XML format.
3. The XML middle layer has potential for implementation of validating shader checks for scene graph structure.

Some disadvantages of the XML layer are,

1. The possible loss of some unique function supported by one of the languages.

2. The XML structure might add more lines of code to the shader program because of the XML validation and checking structure.

For long-term design, we choose the XML structure with the XML middle layer. Before designing our converter, let's first determine the common structures in the existing three shading languages and then decide whether to take the common parts of all three languages to setup the middle XML layer structures and functions.

3.2. 2 XML Layer definition

3.2.2.1 Basic Data Types

Let's look at the following Table 3.1 of the data types in Cg, GLSL, and HLSL. Actually, data types vary for various profiles, what is show here is latest version profile for Cg: vp30 (vertex profile version 3.0) and fp30 (fragment profile version 3.0).

Table 3 - 1 three shading language basic data type table 1

Cg	HLSL	GLSL
bool	Bool	Bool
bool[]	Bool[]	bool[]
int	Int	int
half		
float	float	float
float[]	float[]	float[]
double	double	
double[]	double[]	
		float
		float[]
float2	float2	
float2[]	float2[]	
float3	float3	
float3[]	float3[]	
float4	float4	
float4[]	float4[]	
		vec2
		vec2[]

		vec3
		vec3[]
		vec4
		vec4[]
float3x3	float3x3	
float3x3[]	float3x3[]	
float4x4	float4x4	
float4x4[]	float4x4[]	
		mat3
		mat3[]
		mat4
		mat4[]

The following table shows data type mapping.

Table 3 - 2 three shading language basic data type mapping table 2

Cg	HLSL	GLSL
bool	bool	bool
bool[]	bool[]	bool[]
int	int	int
half	float	float
float	float	float
float[]	float[]	float[]
float[]	int[]	int[]
float	int	int
double	double	float
double[]	double[]	float[]
double	double	float
double[]	double[]	float[]
float2	float2	vec2
float2[]	float2[]	vec2[]
float3	float3	vec3
float3[]	float3[]	vec3[]
float4	float4	vec4
float4[]	float4[]	vec4[]
float3x3	float3x3	mat3
float3x3[]	float3x3[]	mat3[]
float4x4	float4x4	mat4
float4x4[]	float4x4[]	mat4[]

The data types of Cg and HLSL are almost identical. (Refer to Table 3.3). The following three cases will be the comparison between Cg/HLSL with GLSL

Table 3 - 3 Comparisons basic data type between Cg/HLSL with GLSL

Cg/HLSL	GLSL
float	float

Case (1)

Another example:

Cg/HLSL	GLSL
float2	vec2
float2[]	vec2[]
float4x4	Mat4
float4x4[]	Mat4[]

Case (2)

Further more:

Cg/HLSL	GLSL
float2	ivec2
float2[]	bvec2[]
half	float

Case (3)

We compared CG, GLSL, and HLSL language formats. Next we showed how the XML middle layer data types have been defined. Three cases in Table 3.3 show that the data types can easily find their match because they are either exactly the same data type with the same name or exactly the same data type with a different name. However, there are some data types that are hard or impossible to find exact matches. For example: for data type “half” in Cg, there is no exact match in other languages. The data type “half” is a new data type defined in Cg, and does not exist in C or C++. It holds a half-precision floating-point value (typically 16-bit). This is more efficient in both storage and performance than the standard 32-bit precision floating-point value. However, for sharing this data type via the web, we have to give up this benefit for now. This is a tradeoff for making all client machines able to compile the shader smoothly and sharable via the Internet. Figure 3.8 shows our strategy for deciding what data types should be eventually listed in XML middle layer.

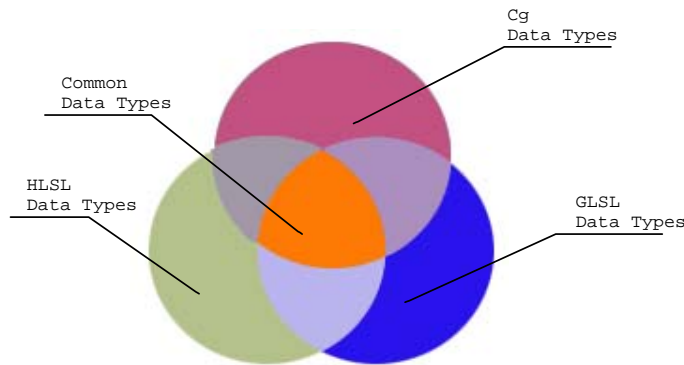


Figure 3 - 8 : XML layer common data set definition

The final data type set chosen for XML is the common set of Cg, HLSL, and GLSL. The ones that couldn't find matching data types will be matched with its looser data type. For example, there is no matching data type for "half" in GLSL, so we use "float" instead in the XML data set. During the SIGGRAPH 2004 presentation, some suggestions were given for enlarging the data type set to contain every single data type in each of the languages. A problem with this idea is for each new data type added into one of the shading languages; we need to update our whole data matching set for six directions, instead of just matching the new data type to an existing common data type set. This strategy is not strict in the situation where there is no existing data type to match within the common set, even with a looser condition. For sharing this data we have to add this new data type to all mapping tables and add the part of new matching into the code manually.

Table 3.4 shows the matching data types from Cg, HLSL, and GLSL to XML, and X3D proposed data types.

Table 3 - 4 X3D, XML and shading languages basic data type mapping table

X3D	XML	Cg	HLSL	GLSL
SFBool	Bool	bool	bool	bool
MFBool	bool[]	bool[]	bool[]	bool[]
MFInt32	Float[]	float[]	int[]	int[]
SFInt32	Float	float	int	int
SFFloat	Float	float	float	float
MFFloat	Float[]	float[]	float[]	float[]
SFDouble	double	double	double	float
MFDouble	double[]	double[]	double[]	float[]
SFTime	double	double	double	float
MFTime	double[]	double[]	double[]	float[]
SFNode	Node fields	Node fields	Node fields	Node fields
MFNode	Node fields	Node fields	Node fields	Node fields
SFVec2f	Float2	float2	float2	vec2
MFFVec2f	Float2[]	float2[]	float2[]	vec2[]
SFVec3f	Float3	float3	float3	vec3
MFFVec3f	Float3[]	float3[]	float3[]	vec3[]
SFVec4f	Float4	float4	float4	vec4
MFFVec4f	Float4[]	float4[]	float4[]	vec4[]
SFVec3d	Float3	float3	float3	float3
MFFVec3d	Float3[]	float3[]	float3[]	float3[]
SFVec4d	Float4	float4	float4	float4
MFFVec4d	Float4[]	float4[]	float4[]	float4[]
SFRotation	Float4	float4	float4	vec4
MFFRotation	Float4[]	float4[]	float4[]	vec4[]
MFCColor	Float4[]	float4[]	float4[]	vec4[]
SFColor	Float4	float4	float4	vec4
SFImage	int[]	int[]	int[]	int[]
MFFImage	int[]	int[]	int[]	int[]
SFString	Not supported	Not supported	Not supported	Not supported
MFFString	Not supported	Not supported	Not supported	Not supported
SFMatrix3f	Float3x3	float3x3	float3x3	mat3
MFFMatrix3f	Float3x3[]	float3x3[]	float3x3[]	mat3[]
SFMatrix4f	Float4x4	float4x4	float4x4	mat4
MFFMatrix4f	Float4x4[]	float4x4[]	float4x4[]	mat4[]

In Table 3.4, the first column is the proposed data type by the X3D shader group. It has been submitted for approval but its definition has not been finalized at this time. Since we do not have a standard at this time, we will define our data types for XML layer

now. The only job that will need to be done once the new standard X3D data types have been approved is mapping all the data types from XML to them.

3.2.2.2 Texture Sampler Types

A comparison of sampler type objects is done for Cg, HLSL and GLSL. The sampler type “sampler2DShadow” is specified in GLSL, which similarly done in Cg and HLSL by the sampler type “sample2D”. However in Cg the sampler type “samplerRECT” for video temporary support, has no matching sampler type. We will use sample2D to represent this two dimension multi-frames data. The texture sampler types in XML are listed in the first column of Table 3.5. Its matching types in Cg/HLSL/GLSL are listed in the remaining columns.

Table 3 - 5 X3D, XML and shading languages Sampler type mapping table

XML	Cg	HLSL	GLSL
sampler1D	sampler1D	sampler1D	sampler1D
sampler2D	sampler2D	sampler2D	sampler2D
sampler3D	sampler3D	sampler3D	sampler3D
samplerCUBE	samplerCUBE	samplerCUBE	samplerCube
samplerRECT	samplerRECT	samplerRECT	No-support
sampler1D	sampler1D	sampler1D	sampler1DShadow
Sampler2D	Sampler2D	Sampler2D	sampler2DShadow

3.2.2.3 Qualifiers / Modifiers

The meaning of “uniform” in Cg is not same as it is in Renderman. In Renderman, the uniform modifier indicates those values that are constants over the surface. In Cg, a “uniform” qualifier variable obtains its initial value externally, for example from the main program. In Cg, all variables are changeable unless its qualifier is

“const”. In GLSL, “uniform” qualifier variables are changed at most once per primitive vertex and they pass user defined states from the application to both shaders. The “attribute” qualifier variable is typically changed per vertex. Because there is no attribute qualifier in Cg, both “attribute” and “uniform” are mapped as uniform. In HLSL, vertex and pixel shaders have two types of input data, “varying” and “uniform”. The varying input is the data that is unique to each execution of a shader. For a vertex shader, the varying data (i.e. position, normal, etc.) comes from the vertex streams. The uniform data (i.e. material color, world transform, etc.) is constant for multiple executions of a shader.

Another important qualifier in GLSL is “varying”. The “varying” variable passes information from the vertex shader to the fragment shader. In Cg, this task is done to some variables with the modifiers, “IN”, “OUT” or “INOUT”. We will give an example to explain how to convert the “IN/OUT” with “varying” in the later implementation section.

We choose “const”, “uniform”, “IN/OUT” as the middle XML layer qualifier.

Please refer following mapping table 3.6.

Table 3 - 6 X3D, XML and shading languages basic Qualifiers mapping table

XML	Cg	HLSL	GLSL
const	Const	uniform	uniform
uniform	Uniform	varying	attribute
In, out, inout	In, out, inout	In, out, inout	varying

3.2.2.3 Transformation Matrixes Parameters in Cg/HLSL vs. Build in Matrixes in GLSL

The conventional arrangement of transformations used to position process vertex positions. The Transformation Matrices in both languages Cg and GLSL are doing the same task. Let's look at the Figure 3.9 borrowed from Cg tutorial book as following. This graph illustrates

In GLSL, there is a set of built-in uniform transformation matrices that are responsible for vertex transformation. For example, "gl_ModelViewMatrix" takes care of the modeling and viewing transformation. If object vertices in the object space are multiplied with "gl_ModelViewMatrix", the coordinate of the object result in Clip space. It is same situation for

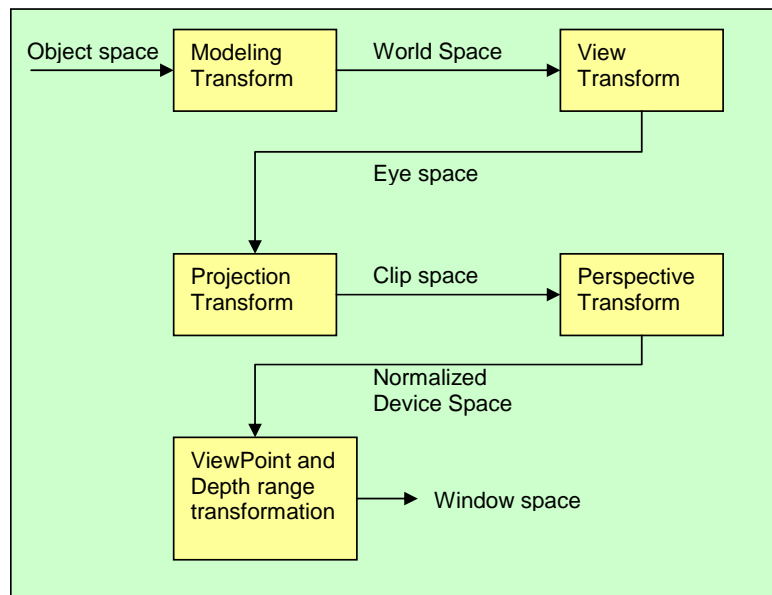


Figure 3 - 9 Coordinate System and Transformation for Vertex Processing

other "gl_" built in matrix operations. Since the matrices in Cg don't have fixed names, they can be user defined, as long as it is named the same as it is in the main program, where the matrices were sent out, it will have no compile errors. For X3D application, we

assume we are loading the loading different shader from one main program. So, we simply use the same name as the matrices in GLSL. The following Table 3.7 gives a mapping between Cg, XML and GLSL built-in uniform matrices. XML transformation matrices named the same as the ones in GLSL.

Table 3 - 7 X3D, XML and shading languages Transformation matrix mapping table

XML	Cg	HLSL	GLSL
gl_ModelViewMatrix	gl_ModelViewMatrix	gl_ModelViewMatrix	gl_ModelViewMatrix
gl_NormalMatrix	gl_NormalMatrix	gl_NormalMatrix	gl_NormalMatrix
gl_ModelViewProjectionMatrix	gl_ModelViewProjectionMatrix	gl_ModelViewProjectionMatrix	gl_ModelViewProjectionMatrix

For this set of transformation matrices mapping, the program works with a condition, which is the transformation matrices are named the same as it is in the mapping table. When users use this converter, they have to clean their code with the same name as the ones we specified in table 3.7 and change their name back to the ones used in main program once after the conversion. The reason we can assume they are the same in the X3D program is that we had an assumption that these converted shader will be used in one X3D program, so the Transformation Matrix from the same X3D main program should stay the same.

3.2.2.4 Semantics in Cg/HLSL vs. Built-in Attributes and Variables in GLSL - Shader program and Main program connector design

Let's look at a simple example first in both the Cg and OpenGL shading Language:

```

struct output {
    float4 position    : POSITION;
    float4 color       : COLOR;
};

```

Example 3.2 Cg semantic binding example

In Example 3.2, “POSITION” and “COLOR” are both called semantics. Semantics are in a sense, the connector that binds a Cg program to the rest of the graphics pipeline. “POSITION” and “COLOR” indicates the hardware resource that the respective member feeds when the shader program returns its output. “POSITION” is the clip-space position for the transformed vertex. “COLOR” means “primary color” in OpenGL or “diffuse vertex color” in Direct3D. In Cg profile vp20 based on OpenGL for example, let’s look at the following out vertex to application connector:

```

#pragma bind appin.Position = HPOS
#pragma bind appin.Position = COL0

// define inputs from application
struct output {
    float4 HPosition;
    float4 Color0;
};

```

Example 3.3 Cg vp 20 binding example

In Example 3.3, we can see output has two members: HPosition and Color0. #pragma binds are used to specify register locations for the variables. In this case, Homogeneous position information resides in the hardware register HPOS and vertex color information reside in the register COL0. This earlier version of binding method already has been replaced by introduced the Semantics concept. We will focus on the

comparison between Semantics format connector in Cg and the binding method of GLSL shader with its application.

Since OpenGL shading Language is the core part of the OpenGL library, a set of attribute and built-in variables have been defined as part of OpenGL shading language for vertex shader and fragment shader respectively. According to Cg's semantics, in GLSL, these attributes and built-in variables take the connectors' task and talk to the OpenGL API. For example, a vertex shader can access the standard attribute using the following attributes.

In GLSL, there are special output variables for output in the Vertex shader. For example, "gl_Position" is write position in clip space. So these attribute and variables really do the same thing that the semantics does in Cg, i. e., function as a connector between the shader program and main application.

The same situation happens to the fragment shader program. A fragment shader can read in varying variables and the fragment shaders special output variables listed as follows. Most of the varying names in fragment processor are similar to the attributes in vertex shader. However, there is no conflict, because the scope of vertex attributes is within the shader. The detail mappings are shown in Table 3.8. Because the "gl_XXX" has been defined as part of glstate, we choose "gl_XXX" as the name in the XML middle layer to minimize some possible conversions.

```

// vertex Attributes

attribute vec4 gl_Color;
attribute vec4 gl_Vertex;
attribute vec4 gl_Normal;
attribute vec4 gl_SecondColor;
attribute vec4 gl_MultiTexCoord0;
attribute vec4 gl_MultiTexCoord1;
.....
attribute vec4 gl_MultiTexCoord7;
attribute vec4 gl_FogCoord;

// vertex build-in variables

vec4 gl_Position;
vec4 gl_ClipVertex;
float gl_PointSize;

// fragment Attributes

varying vec4 gl_Color;
varying vec4 gl_SecondColor;
varying vec4 gl_MultiTexCoord0;
varying vec4 gl_MultiTexCoord1;
.....
varying vec4 gl_MultiTexCoord7;
varying vec4 gl_FogFragCoord;

// fragment special output variables in GLSL

varying vec4 gl_FragColor;
varying vec4 gl_FragDepth;

```

Vertex / Fragment shader attributes and built-in variables list in GLSL

In the Table 3.8, the Semantics, which are widely used in all existing profiles, are listed in the second column in the following table. Accordingly the matching XML layer, Cg, HLSL and GLSL attribute names are listed in the first, third and fourth columns:

Table 3 - 8 XML, Semantics and shading languages built-in variable mapping table

XML	Cg	HLSL	GLSL
gl_Position	POSITION	POSITION	gl_Position
gl_Vertex	POSITION	POSITION	gl_Vertex
gl_Normal	NORMAL	NORMAL	gl_Normal

gl_Color	COLOR	COLOR	gl_Color
gl_SecondColor	COLOR0	COLOR0	gl_SecondColor
gl_MultiTexCoord0	TEXCOORD0	TEXCOORD0	gl_MultiTexCoord0
gl_MultiTexCoord1	TEXCOORD1	TEXCOORD1	gl_MultiTexCoord1
.....
gl_MultiTexCoord6	TEXCOORD6	TEXCOORD6	gl_MultiTexCoord6
gl_MultiTexCoord7	TEXCOORD7	TEXCOORD7	gl_MultiTexCoord7

3.2.2.5: Struct Definition

We define our XML layer as native XML format of tree structure. In Example 3.4, since all the three shading languages are coming from C, this struct structure in all three languages is the same.

```

struct A
{
    float m;
    float4X4 n;
}

```

Example 3.4 simple struct in Cg/HLSL/GLSL

Its tree structure in XML is shown in Figure 3.10:

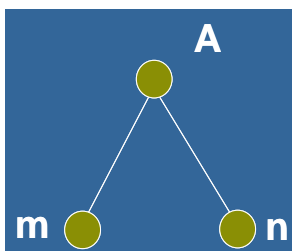


Figure 3 - 10 example 3.4 struct in XML tree structure

Its XML format structure is shown as:

```

<struct value = "A">
  <struct_member type = "float" name = "m"/>
  < struct_member type = "float4X4 " name = "n"/>
</struct>

```

Example 3.4 (a) example 3.4 struct in XML format

3.2.2.6 Operators/Operation Definition:

Since all shading language design was based on C, most mathematic operations are the same. We are not going to give details about each operation here; instead, we will talk briefly about the structure of the operators and operands in the XML tree structure. For separate operators with other user-defined variables, we reserve “op_” as the key word standing for operation. Please refer to the appendix operator for a detailed operation map.

Again, operator and operation also will also show as tree structure. For a simple operation example in Cg as following:

```
A = B + C;
```

Example 3.5 simple expression with operation in Cg/HLSL/GLSL

Its tree structure is shown as:

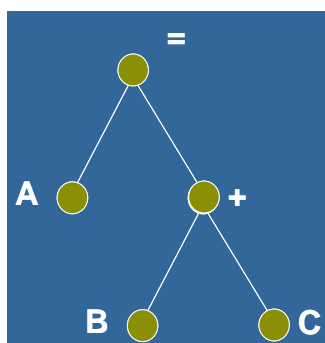


Figure 3 - 11 XML tree structure of example 3.5

Its XML format structure is shown as Example 3.5 (a):

```

<oper_assignment>
  <parameter1 name = "A">
    <parameter2 >
      <oper_unary_plus>
        <parameter1 name = "B">
          <parameter2 name = "C">
        </ oper_unary_plus >
      </parameter2>
    </oper_assignment>

```

Example 3.5 (a) example 3.5 in XML format

There is a special operator in shading languages called swizzle. What this operator does is select vectors in multi-dimension arrays. For example, a position vertex defined as “position1 (20.0, 10.0, 0.0)”. We could assign a variable float pos_x = pos.x; or a two dimension float vector name float2 position_xy = position.zz.. We should give the swizzle property a definition in XML layer. Fortunately, this swizzle does exist in all three shading languages and they function equivalently. We keep the swizzle name and define it as an operator in XML layer.

3.2.2.7: Statement definition

As in C, the shading language supports flow control with “if...else”, “for”, “while/ do.... while” conditions. In the following is an example of the “if” statement. We used keywords “if”, “else” as tag names in XML and “condition” and “body” are represented as “()” and “{ }” respectively. The following examples show a condition expression in Cg/HLSL/GLSL and XML formats accordingly.

```

if (A)
{   B ;   }
else
{   C ;   }

```

Example 3.6 condition statement in Cg

```

<if>
  <condition> A </condition>
  <body> B </body>
</if>
<else>
  <condition> A </condition>
  <body> C </body>
</else>

```

Example 3.6 (a) example 3.6 in XML tree structure

The “if...else” condition statement is shown as example 3.6 (a) in XML format.

Since Cg and GLSL have similar structures for the “if” condition expression in example 3.6. We defined a set of tag elements named as keywords of the statement. i.e. “if” and “else”. Example 3.6 (a) shows the XML tree structure if example 3.6.

3.2.2.8 Functions Design

In this section I will give comparisons of built-in functions exist in all shading languages and define one for the XML layer function set. The format of the function will be included inside of a pair of <function> </function> tags. Functions include mathematical functions, geometric functions, texture map functions etc. Most of the functions are similar and easy to find matches for and we will ignore the obvious ones here. Two situations for non-matching functions were found. The first one is that they have defined exactly the same function with a different name in one of the languages. For

this situation, we picked one of the function names as the XML middle layer function. An example is the function “frac(x)” in Cg returns the fractional part of value x. The function “fract(x)” in GLSL does exactly the same task. In this case, we selected “fract(x)” as the function name in the XML middle layer. The second case is that some of the functions only exist in one of the languages. For example, in GLSL, function Mix(a, b, z) does a mixing of the values a and b by percentages of z and 1-z, respectively. However, there is no similar function in Cg. So, how should we handle this if it is needed to mix two values in XML? So that after it is converted to Cg, it does the same job? One way for realizing this function is write an equivalent expression shown in example 3.7 (a). The result from the expression in example 3.7 (a) is the same as the Mix() from example 3.7. In this example, the corresponding expression from Table 3.9 will be used for processing the same function of Mix () in GLSL. Please refer more detail of functions mapping in Appendix B5

```
Mix(a, b, z)
```

Example 3.7 Mix function in GLSL

```
aColor*(1.0 - w)+ bColor*(w);
```

Example 3.7 (a) equivalent expression of mix function in example 3.7

Use this function as following:

Table 3 - 9 Mix () function in GLSL mapping to different languages

XML	<code>color = aColor*(1.0 - w)+ bColor*(w);</code>
GLSL	<code>color = mix(aColor, BColor, w)</code>
Cg	<code>color = aColor*(1.0 - w)+ bColor*(w);</code>
HLSL	<code>color = aColor*(1.0 - w)+ bColor*(w);</code>

There are few other functions, e.g. `lit()` in Cg, etc. please refer to the appendix of function mapping table for details.

Another special case is for matrix/vertex multiplication. In Cg, a function `mul(m, n)`, where at least one of (m, n) has to be matrix. In GLSL, all the vertex/ matrix multiplications are simply done by an operator `“*”`. We choose `mul(m, n)`, at least one of (m, n) have to be matrix, the way Cg represent for minimize conversion..

Summary of the design rules for defining middle XML layer function.

(1) If map exist for similar functions in shading languages use one of them as the name of XML function name.

(2) Choose common expression implementation for the unique functions only supported by one language.

3.3 Shader language Converter Implementation

3.3.1 SLC Conversion pipeline

Each of three languages, CG, HLSL, GLSL, are located in exchangeable position with other ones in the SLC structure and conversion process. CG and HLSL are more similar to each other. We are going to show the conversion between Cg and GLSL. For

simplification, the conversion pipeline from Cg to XML to GLSL is shown in the Figure 3.12. This pipeline is composed of a scanner, a parser, and XML DOM components.

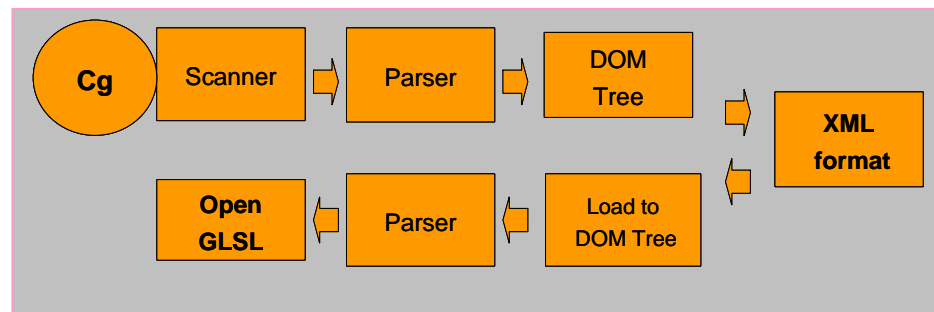


Figure 3 - 12 : CG→XML→GLSL flow

Shaders written in the Cg format are first read in by the scanner and then sent to the parser to be passed to a DOM tree. This DOM tree can be a simple output like the XML format. This output will be the version used when a shader is published to the web. On the client side, if the user needs a GLSL shader format, the XML format shader is downloaded from the Internet and loaded into the DOM tree, then the parser outputs it to the GLSL shader format. The converted GLSL shader is sent to the GPU. The equivalent shader effect will appear in the web browser.

In Figure 3.13, the conversion of CG→XML→GLSL is shown as highlighted with a dashed line path in the entire SLC structure. Since the SLC is a symmetric structure, we developed the conversion between CG, XML, and GLSL to prove the concept of our proposed idea. The remaining implementation should be very similar.

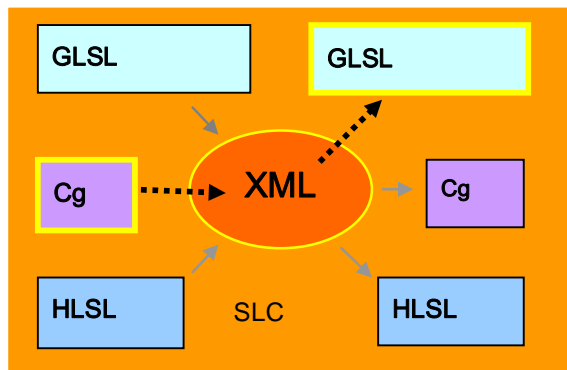


Figure 3 - 13 : CG→XML→GLSL conversion Path

3.3.2 Scanner

The Scanner's function is to reformat and simplify input code, in this case, Cg code. Every developer friendly programming language is designed with maximum flexibility for the developer. Very often, developer's code that is written in a slightly different format will still work correctly, as long as it doesn't have any syntax errors. A very simple example is for the empty line in a code. Most languages have no compile error for this. We try to optimize the XML layer by cleaning up "dirty" code. "Dirty" here means, for example to define a variable a, I write "float A = /n b;" code written in two lines. This does not effect compilation, however it cause difficulty for later conversion to the tree structure. So, cleaning up code is an important part of Scanners job.

Simplifying is the second job of the scanner. XML is a binary tree structure and this means the output will only allow up to two children. For example, a statement $A = B + C + D$ has to be simplified as: $A = (B + C) + D$. This simplification makes it easier for later parsing. Another simplification that needs to be done is to rename the uniform

matrix parameters as its defined name in the XML layer, which is the same as built-in matrix names in GLSL.

3.3.3 Parser

The Parser contains both conversion and parsing processes. After conversion, The Microsoft XML DOM component has been employed to parse the final output. We are going to focus on the conversion processes. The conversion process includes Data type conversion, declaration conversion, parameter conversion, operator conversion, and function conversion.

3.3.3.1 Data types Conversion:

Data type conversions includes basic data types conversions and sampler types conversions. This process is straight forward and based on the mapping definition of the XML layer (table 3.3 and table 3.4). Conversion from CG to XML is based on the data type mapping table of Cg2XML. Conversion from XML to GLSL is based on the data type mapping table of XML2GLSL. Please refer to the mapping tables in section 3.2.2 and source code in the Appendix C1.

3.3.3.2 Declaration Conversion:

Let's look at two declaration examples 3.9 and example 3.10. The first example is a 4 floating point elements vector "position1" declaration with a value assigned. The value is a result of function return. The function is the multiplication of two pre-defined variables. The second example is also a 4 floating point vector named "oPosition". This

vector is an output position from the vertex shader, which is indicated by the modifier “out”. Its semantics is POSITION. These two declarations cover most cases we need to consider for declaration conversions.

```
float4 position1 = mul(matrix, vPosition);
```

Example 3.8 declaration conversion example 1

```
out float4 oPosition : POSITION
```

Example 3.9 declaration conversion example 2

This declaration conversion basically includes data type conversions, qualifier/modifier conversions, assignment/operation conversions and semantics conversions as well. Each of these conversions is based on the mapping tables 3.4 - 3.9 and the XML tree structure. The outputs of the XML format of these two examples above are shown as follows:

```
<declaration>
  <modifier></modifier>
  <qualifier></qualifier>
  <type>float4</type>
  <op_assign>
    <para>pos</para>
    <func>
      <para>mul</para>
      <parameters>
        <para> gl_matrix </para> // build-in variable
        <para> vPosition </para>
      </parameters>
    </func>
  </op_assign>
</declaration>
```

Example 3.9 (a) XML format output of Example 3.9

```

<declaration>
  <modifier>out</modifier>
  <qualifier></qualifier>
  <type>float4</type>
  <para>gl_Position</para> // oPosition in Cg
  <semantic>POSITION</semantic>
</declaration>

```

Example 3.10 (a) XML format output of Example 3.10

When the conversion reaches the second part of the path, from XML2GLSL, all modifier/qualifier and data types are converted according to the mapping table of XML2GLSL. For “op_mul”, originally the mul() function, will output as operation “*” . The Output in GLSL is as follows:

```

Vec4 position1 = matrix * vPosition;

```

Example 3.9 (a) GLSL format output of Example 3.9 (a)

For the second example, the output position from the vertex shader will be converted to the built-in variable in GLSL as “gl_position”. So there is no declaration needed other than variable “gl_position”. In the next section, we are going to discuss parameter conversion.

3.3.3.3 Parameter Conversion:

During parameter conversion, we have few decisions to make. They are the modifiers decision (in/out modifier variables in Cg with varying variable in GLSL),

semantic decision, and relocations decision. We give the explanation with an Example 3.11. First let's compare two equivalent code fragments in both Cg and GLSL.

<pre>// GLSL const scale = 2.0; varying position; varying LightPos; void main(void) {....}</pre>	<pre>// Cg const scale = 2.0; void main(out position; out LightPos;) {.....}</pre>
--	---

Example 3.11 Parameters Conversion

By observing both examples in 3.11 and the knowledge that all GLSL main function voids any argument or parameter, where should the Cg out parameters go? We realize there are list of varying variables located ahead of main function in the GLSL shader. Those variables are equivalent to the out parameters in Cg. Figure 3.14 illustrates that the varying variables of GLSL are relocated as the parameters of Cg with the equivalent code shown in Example 3.11.

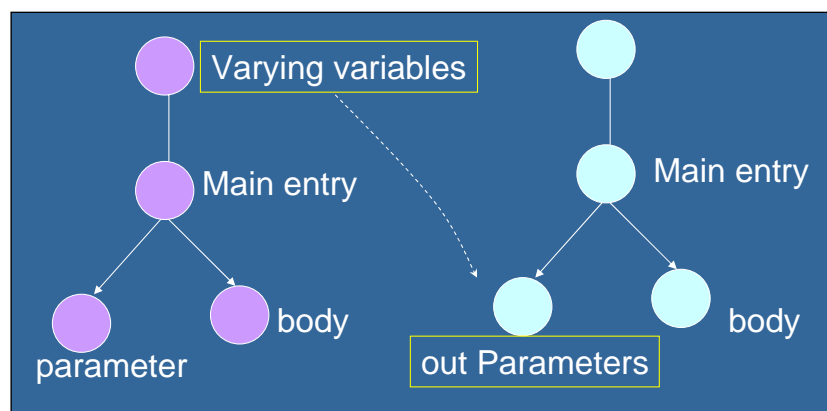


Figure 3 - 14 : Varying variable in GLSL relocated as parameters in Cg

As we know the parameter of the main function in the vertex shader of Cg is not only the “out” parameter, but there are some “in” parameters and uniform parameters as well. All “in” parameters are the input from main programs. They are vertex attributes, vertex positions and normals. Those attributes are defined as built-in variables in GLSL as `gl_Vertex` and `gl_Normal` individually. Other types of uniform parameters are transformation matrices like `modelViewMatrix`, these matrixes are defined as built-in parameters and can be access from calling the name of certain matrices anytime in GLSL. That transformation matrix conversion is shown next.

The following two code fragments in Example 3.12 contain uniform transformation matrices in Cg and built-in transformation matrices in GLSL. The uniform float4x4 matrix name as “`gl_ModelViewProjectMatrix`” is displayed as the built-in variable `gl_ModelViewProjectMatrix` in GLSL, after the process of two conversions. The reason we show matrices with the same name in both formats is because that the uniform matrix in Cg does not have a fixed name. As long as the matrix name is the same as the uniform matrix name of main program the shader in Cg will work. For simplification, we used the GLSL built-in matrix name set as both Cg matrix name set and XML layer name set. All uniform matrices in the parameters section of Cg have been renamed according to the built-in matrix names in GLSL. For the converter to successful convert, the user has to pay close attention to renaming their uniform transformation matrices before and after the conversion.

<pre>//Cg const scale = 2.0; void main(uniform float4x4 gl_ModelViewProjectMatrix; uniform float4x4 Matrix2; uniform float4x4 Matrix3; ...){... ...}</pre>	<pre>// GLSL const scale = 2.0; void main(void) { Pos = gl_ModelViewProjectMatrix*gl_vertex; }</pre>
--	--

Example 3.12 Transformation Matrix Conversion

```
<declaration>
  <modifier>in</modifier>
  <qualifier>uniform</qualifier>
  <type>float4x4</type>
  <para>gl_ModelViewProjectionMatrix</para>
</declaration>
```

Example 3.12 (a) Transformation Matrix output in XML format

Example 3.12 (a) shows variable “gl_ModelViewProjectionMatrix” as an output in XML format. Meanwhile, another decision can be made during the matching of the semantic name of the variables in Cg and the built-in attributes in GLSL, in this conversion, the mapping Table 3.8 used.

3.3.3.4 Operator Conversion:

Most operators are the same for all languages. We defined the middle layer XML operation with “op_” with each name of the operators. A few special cases are explained here. The first one is, operator * in GLSL also matches the mul () function in Cg. This means, during the conversion from GLSL → XML → Cg, whenever an operator has been read in XML, we have to check both of the children of the two parameters (the two

tokens of the mul function). If one of the data types of the children is a vector and the other one is a type of matrix, the output will be a function mul (parameter1, parameter2), otherwise it will be (parameter1 * parameter2), a multiplication function. Converting from the other direction Cg \rightarrow XML \rightarrow GLSL would be straight forward because of the function of “mul ()” is already defined as tag of “op_mul” in XML layer. Once “op_mul” read for processing XML \rightarrow GLSL. It is simply converted to operation “*” as output in GLSL.

3.3.3.5 Function Conversion

Function conversions include built-in functions conversion, texture functions conversion and user-defined functions conversion. For built-in function, the main task is mapping functions from Cg2XML and XML2GLSL. Texture function conversions are also based on the function mapping table. For those unable to find mapping even with a looser scope, we can't support them at this time. This is a limitation, but they are a very small percentage of the total number of functions. Function conversion includes Return type conversion, Function name conversion, Parameters conversion, and Body conversion, which includes Statements conversion, Assignments conversion, and Declarations conversion. Since functions build from data, operations, assignment, statements, etc. We talked about each of the parts above individually. When the function conversion comes, it is automatically completed with all conversion parts done. The only extra points we need to mention are the return type and how to determine if it is a

function. Determining function by the function tag in XML structure and return type will be the type attribute after function name.

In the next section, we are going to give some demonstration pictures of the SLC and example shaders in Shader Library.

3.3.4 SLC interface

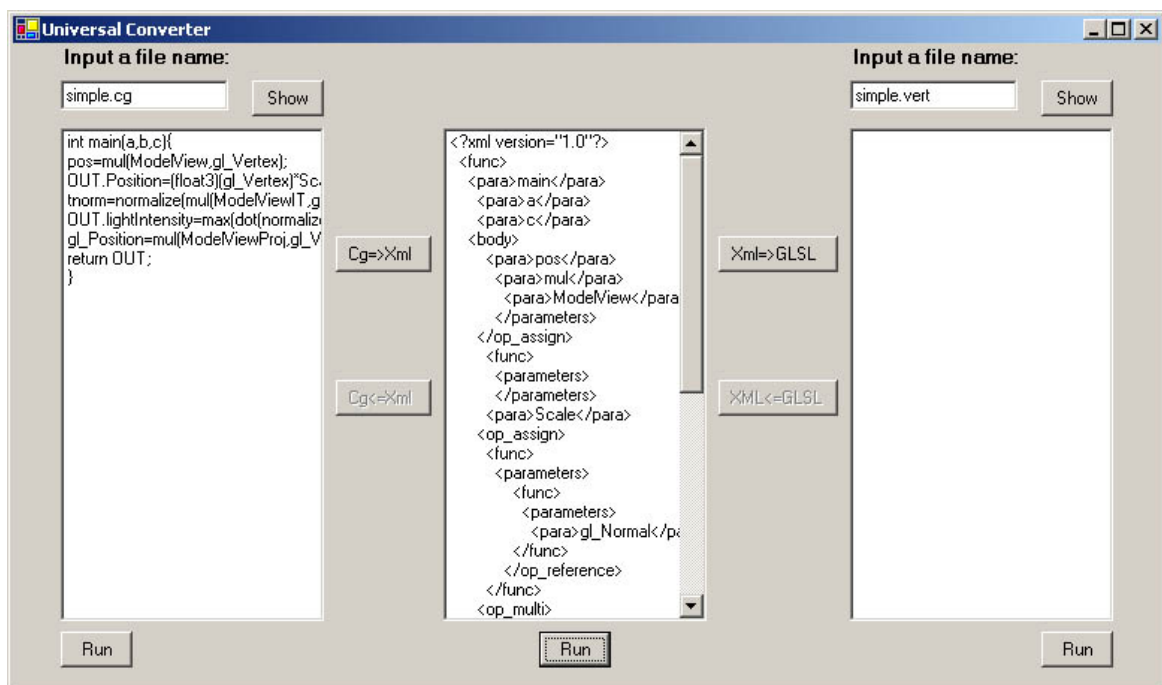


Figure 3 - 15 : Shader Language Converter Interface

On the left part of the interface shown in Figure 3.15, the user can input their shaders in Cg. By clicking on the button “Cg => Xml”, the XML format of the shader is displayed in the middle section. Continue by clicking on the button “Xml=>GLSL”, the shader in GLSL will be available in the right hand section. For running demo, simply click the “Run” buttons.

3.3.5 Shader comparisons between pre-converted and post-converted

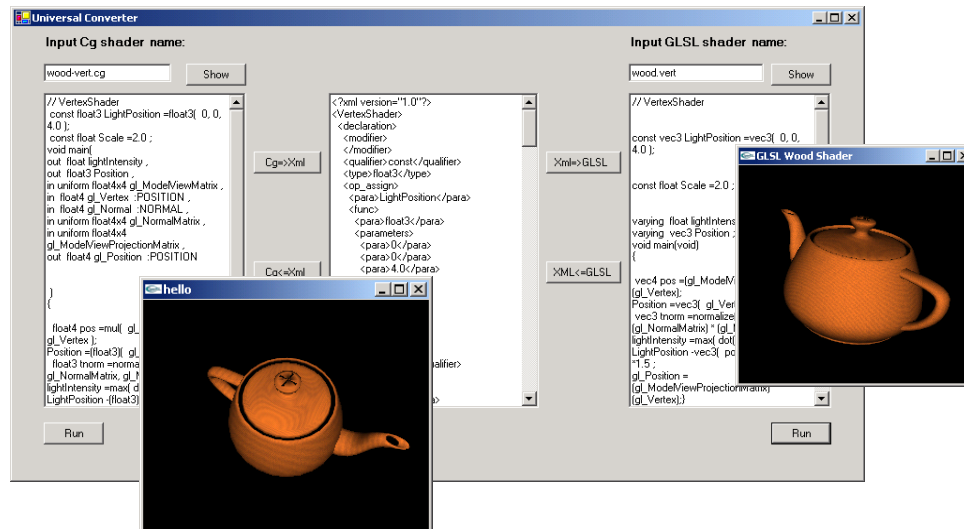


Figure 3 - 16 : Shader Comparison between pre_conversion and post_conversion

This example shows same shader in different shading languages showing same shading effect. In this case, it is a wood shader. First of all, a wood shader is loaded into the Cg source area. Then, we show its result by clicking “run” button. The Cg main program is applying this wood shader on a teapot. When the executable file successfully loads this wood shader, a wood teapot window appears in the lower left corner. By clicking Cg=>Xml and Xml=>GLSL button, a wood shader in GLSL format is produced. A main program loads the GLSL wood shader after clicking the “run” button in the GLSL area. Then, we can see a same wood shader applied on the teapot. Adjusting the parameters allows further testing of the shaders. Some of the parameters adjusted in one format can affect the result in another format

3.3.6 More sample shaders in shader library

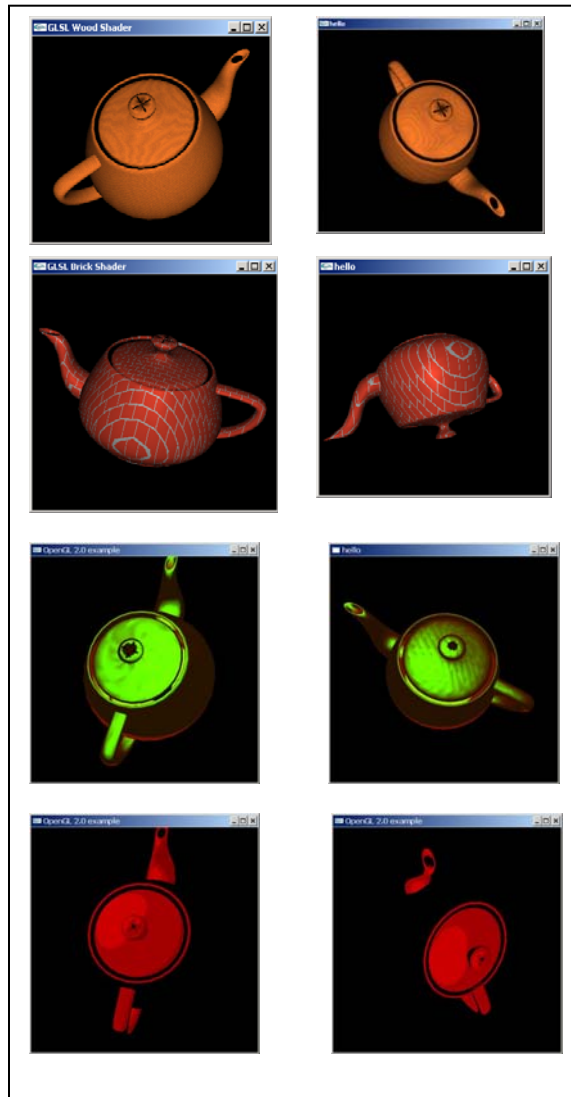


Figure 3 - 17 : The Comparison of shaders in shader library between pre_conversion and post_conversion

Figure 3.17 shows some of the shaders listed in the shader library. The images on the left are the results of different shaders in Cg format applied to the teapot. The images

on the right are the results of shaders in GLSL format applied. Source codes for these shaders are available the Appendix B.

3.4 Summary of Shader-X3D frameworks and preview of its application in Bio-informatics Visualization.

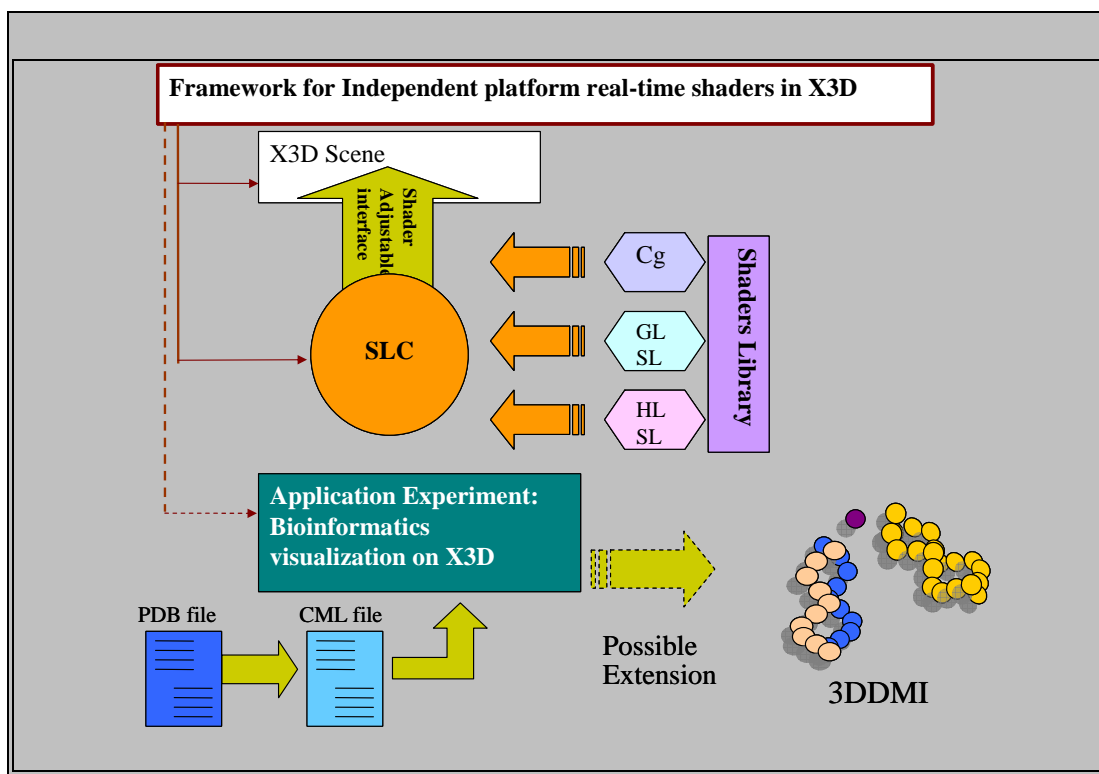


Figure 3 - 18 : Framework for Independent, real-time, interactive Shaders in X3D

So far, we introduced the different parts of Shader-X 3D frameworks. We spent time on the SLC design and implementation. We successfully converted a few example shaders. It proved the idea of our proposal that the skin conversion of the different language shaders can make the shading independent from platform. We gave a brief introduction of the shader library. Now, let's overview the framework again and

introduce a pipeline for this framework applied to Bioinformatics 3D Molecular structures on the web.

We discussed the upper half of the Figure 3.18. Let's look at the remaining bottom half of the picture. We are going to translate a popular 3D format of molecule structures. Protein Data Bank (PDB file format) to a XML based Chemistry Model Language (CML format). Then shaders will be added during the translation from CML to X3D/VRML web 3D format. After that our shader-X3D framework can be applied to the biomolecular structure in an online 3D application. In the coming chapter we will discuss the process of PDB to CML to Shader-X3D conversions. This is an important step for preparing realistic 3D bioinformatics application for sharing online. The future direction for this MDB¹ supported project might involve taking advantage of hardware, like head mounted displays, gloves and haptic devices to realize direct 3D manipulation (3DDMI) for multi-user environment and 3D bioinformatics visualization online.

¹ Molecular Basis of Disease

4 Shader-X3D uses in Bioinformatics Applications

4.0 Preface

“Through the lens of a microscope or the shaft of a telescope exists a universe of life and beauty that is unknown too many. Hidden from sight because of the human eyes restricted ability, atoms, crystals, grains of pollen, sear comets, live and die”

- *From Heaven & Earth by Roucoux K.*

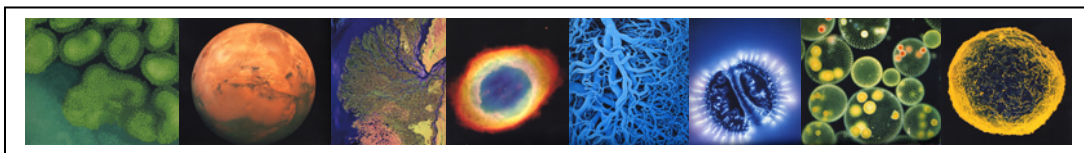


Figure 4 - 0: Beauty of life – *Picture from Heaven & Earth*

Observed though both microscopes and telescopes, man discovered and understood more about the universe. We are living on a huge ball and didn't even realize the ground was curved. This globe we call earth is just one cell in the universe. Looking out the window we see cars, buildings, and many living organisms like humans, animals and plants. It is a very complex ball. What exists on other balls in the universe? Are they as complex as this one? Do they resemble the worlds of science fiction like “Star Wars”? Are the functions and rules that organize them in some kind of balance? Scientists are trying to land on each of them to find the answer. Zoom into a living cell and you can see the similarities between those cellular environments and the bodies that make up the universe. Looking at Figure 4.0, can you tell which is star and which is cell? We can't help asking what's inside of the cell of human body. Are they as complex as the bodies of the universe?

Zooming in to human cells, they are built from proteins. Proteins are the body's worker molecules. "A protein called alpha-keratin forms hair and fingernails, it is also a major component of feathers and wools etc. Muscle proteins called actin and myosin, enable all muscular movement – from blinking to breathing to roller blading. Receptor protein studs the outside of your cell and transmits signals to partner proteins on the inside of the cell. The hemoglobin protein carries oxygen in your blood to everywhere in your body and antibodies are proteins that help defend your body against foreign invaders, such as bacteria and viruses." [NIH2005]. Proteins are built with different shaped small molecules called amino acids. Twenty different shapes and sizes of amino acids are used to build the different structures and sequences in proteins. Most proteins contain 50 to 5000 amino acids hooked end to end in different combinations. They twist and fold finding a balance when forming each of the proteins. Because proteins are built in three dimensional spaces, studying the 3D structure of those shapes will help biologist discover the secrets of those proteins which cause human diseases and behaviors. Figure 4.1 shows a set of sample of proteins.

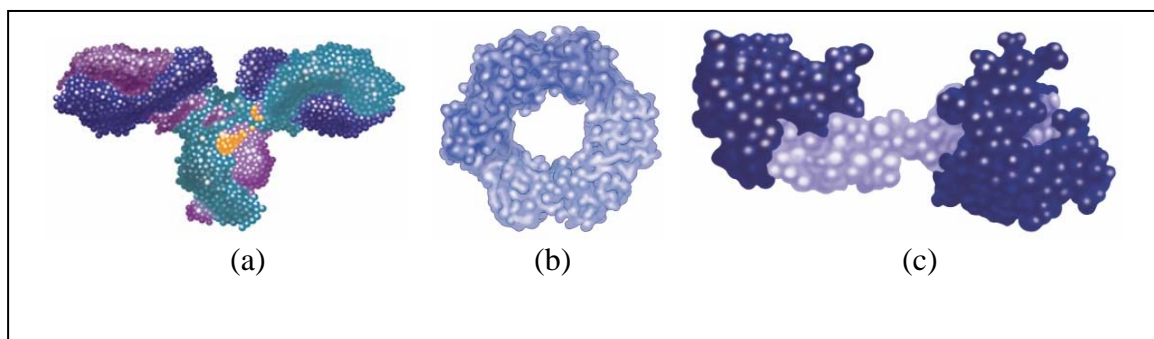


Figure 4 - 1 (a) Antibodies are immune system proteins that rid the body of foreign material, including bacteria and viruses. The two arms of the Y-shaped antibody bind to a foreign molecule. The stem of the antibody sends signals to recruit other members of the immune system. (b) Some proteins latch onto and regulate the

activity of our genetic material, DNA. Some of these proteins are donut shaped, enabling them to form a complete ring around the DNA. Shown here is DNA polymerase III, which cinches around DNA and moves along the strands as it copies the genetic material. (c) Troponin C triggers muscle contraction by changing shape. The protein grabs calcium in each of its "fists," then "punches" other proteins to initiate the contraction.

Using microscopes, we are able to study the cell level component of objects. current technologies that allow us to study molecular structures are X-Ray crystallography and nuclear magnetic resonance (NMR). The X-Ray technique was first used in 1959 at Cambridge University by John Kendrew to determine the structure of myoglobin. X-Ray Crystallography uses wavelengths of 0.5 to 1.5 angstroms, to measure the distance between atoms. One angstrom is one ten-billionth of a meter, that's 10 million times smaller than the ".", used to complete this sentence. Visible light with a wavelength 4,000 to 7,000 angstroms is used in ordinary light microscopes. Figure 4.2 shows how big a water molecule is by comparison to objects that we can see with the naked eye.

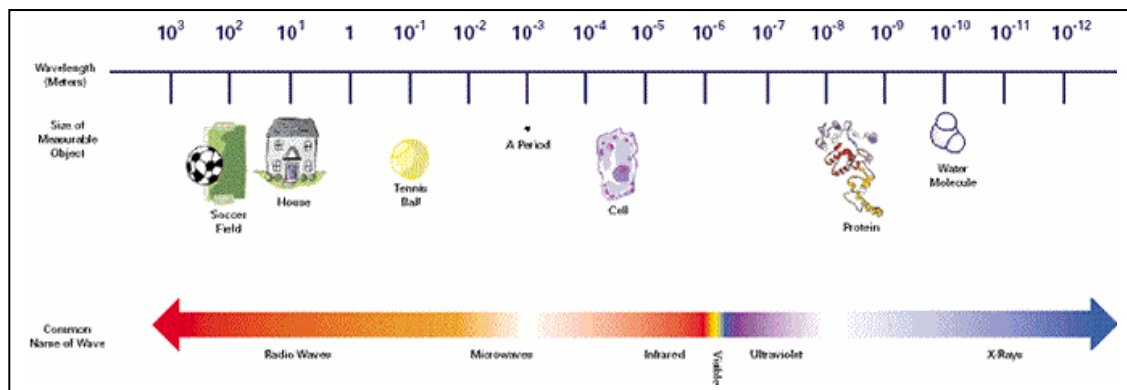


Figure 4 - 2 : Using light to measure an object, the wavelength of the light needs to be similar to the size of the object. X-rays, with wavelengths of approximately 0.5 to 1.5 angstroms, can measure the distance between atoms. Visible light, with a wavelength of 4,000 to 7,000 angstroms, is used in ordinary light microscopes because it can measure objects the size of cellular components. [NIH 2005]

Beyond microscope, complex arrangement of atoms within molecules has been displayed on a naked eye seeable screen by collecting molecule data, complicated computation, and computer modeling techniques.

- *Art marries Science from "the structure of life"*

What we will focus on is taking advantage of computers to show atoms bigger and universes smaller so that they can be seen with the human eye, hence to help understand the beauty of life. With the help of computer graphics, we will be lead beyond the limited scope of our sight and our feet. Beyond the computer screen, technologies are trying to bring the 3D model of the molecule to life to touch, to feel, to study them.

4.1 Background of structural biology and Web 3D Bioinformatics Visualization

Amino acids are comprised of atoms with different structural bonding configurations between them. Proteins are built from 20 different amino acids. A protein consists of a chain with of length of more than 100 amino acids. This means the number of possible proteins is 20^{100} . Functions of proteins determined by its structure can be divided as the following four levels: primary structure is the sequence of the chain; secondary structure is the spatial arrangement, its screw-shaped structure, unfolded; tertiary structure shows the 3D conformation of the chain; quaternary structure is how the chains, which proteins are composed of are related.

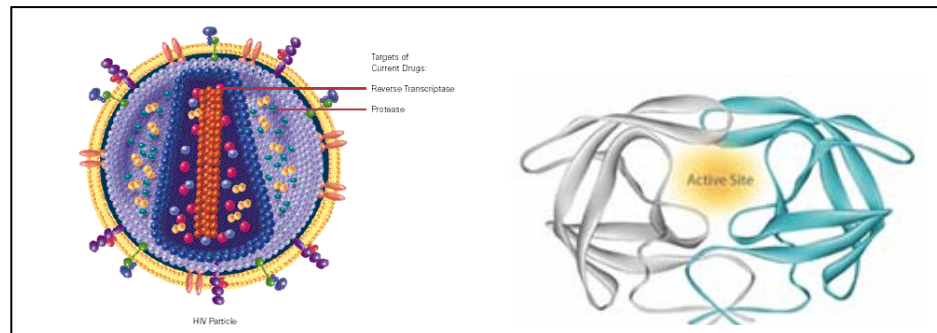


Figure 4 - 3 : HIV protease² is a symmetrical molecule with two equal halves and an active site³ near its center

Let's look at some research on Human immunodeficiency virus (HIV) structure based drug design to see how structural biology helps. In 1981, doctors discovered HIV or Aids, a virus that attacks the human immune system; it is the big killer for humans. Although no cure for protecting the immune system from HIV has been discovered, structured modeling in biology played a key role during the development of drugs for fighting this virus. The X-Ray Crystallography structure of HIV protease¹ was determined by 1989, shown in Figure 4.3. Since then scientists have been interested in blocking the enzyme away from the virus. HIV protease is a symmetrical molecule with two equal halves and an active site² near the center. [NIH 2005]

Figure 4.4 shows a single blood cell named T-lymphocyte infected with HIV. It is scanned with electric micrograph, magnified 24000 times. The virus particle is shown as red dot in the picture. By attacking the T-cell, HIV reduces the efficiency of the whole immune system.

² Protease: An enzyme that cleaves peptide bonds that link amino acids in protein molecules.

³ Active site: the active site of an enzyme is the binding area where has an acceleration of the chemical reaction rate.

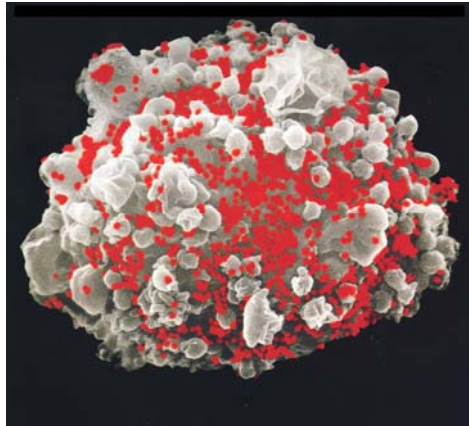


Figure 4 - 4 : HIV protease cell

The Molecules of HIV, an article written by Dan Stowell give us a story of “HIV life-cycle” [Stowell 2005] and an excellent animation of the HIV life-cycle is available at [HIV-Animation 2005]

After one or more virus particles must enter the body. These particles will be borne in fluid such as blood or semen. “The first step of the HIV life cycle is binding to the cell membrane, followed by membrane fusion, to get the virus particle's contents into the host cell. Then follows reverse transcription of the HIV's genome from RNA into DNA, and its integration into the host genome. Last step is producing new virus. Once integrated the virus can lie low in human cells, or can begin the production of new viral RNA and proteins, turning the cell into a HIV factory. This production is followed by assembly, budding, and maturation, in which the new HIV particles are packaged up and sent out to infect new cells.” [Stowell 2005]

With the help of computers researchers can finally see their target molecules structure. By feeding the structural information into a computer-modeling program, they

can spin a model of protease around and zoom in/out on particular atoms. Computer visualization has become one of most important tools to determine the type of molecule and location of the structure, which has been attacked or blocked away from the virus. The process of retrieving molecular data from the chemistry lab as input for the computer and design from computer lab back to biology lab saves a lot of time and money. It has good potential to be used on drug design.

Beyond drug design: “As its root, structural biology can teach us about the fundamental nature of biological molecules”

- *The structure of life*

Beyond visualization, humans gain knowledge through touch. Structural physical modeling with computers can also help with the manufacturing of a touchable model for biologist intuitive design [Mike bailey 2005]. One potential idea is that a physical model can dynamically simulate the real protein structure. However at this time, there is no existing technology with this ability. The best we can do is to simulate with the computer as much as possible. Virtual/Augment reality would be one of the best choices.

In the following section the 3D data format for molecules and the way to visualize them will be introduce. It will also introduce some 3D bioinformatics software for structural chemistry and biology.

4.1.1 3D Protein structure in different formats

4.1.1.1 PDB

The Protein Data Bank (PDB) [PDB 2005] is a public database that can be downloaded in PDB file format with the extension *.pdb. It contains 3D coordinates of

atoms of 3D biological macromolecular data. The 3D structure files, the *.pdb format, is widely supported by various 3D visualization and/or modelling applications. In the Biology Department at Georgia State University the Arp/Warp software package for bioinformatic research has been used. One of output format of Arp/Warp could be the PDB. A PDB file containing the 3D structure of proteins could be successfully represented in VRML, especially the coordinates of each atom, or say the center of the atom. But PDB does not contain the information of how the atoms are connected, which is very important. The connection between atoms is called “bond” or “electron bond”.

To determine where the bond should form and how long the bond length should be Olive Krown developed a formula for calculating this in his thesis in spring 2003 [Krone 2003].

According to Oliver, due to some hydrogen atom missed in the PDB dataset, it is not ideal to use the electron charge density for calculating the bonds between atoms. PDB gives the coordinate position of most of the atoms. Sizes of different atoms and the connection length between them are already known so by using the following formula, the single bonds between atoms A and B will be determined. The double bond is determined by shortening the single bond by 0.21% [Krone 2003].

$$D(A - B) = r_A + r_B - 0.08 * |x_A - x_B|$$

Atomic spacing = sum of both atoms radius – 0.08 (different of both Electron Affinity values)

Comparing this value with the standard connection length, we can determine if a possibility exists for a connection between those particular atoms. Next step Oliver took was to calculate the angles between the atoms to select the possible connections.

There were two steps for calculating the molecular structure. One for storing atoms for later connection computation and another is the actual calculation. A container named ResidueContainer for checking the amino acid by pushing each of the atoms in and comparing with the standard amino acids. The calculations of bonds between atoms run until the last line of PDB file. At the same time bonds are build, the output of the conversion was in CML – Chemistry Markup Language.

4.1.1.2 CML

Much research has been done on how to represent biomolecular structures. The Chemical Markup Language (CML) [CML 2005] is a data format for storing chemical data in eXtensible Markup Language (XML). As a new markup language defined in XML, CML defined its own elements and attributes rules especially for chemistry research. It contains numeric and string data in scalar, array, matrix or tabular form of molecular information.

The chemical Markup language (CML) was developed by P. Murray-Rust and H. S. Rzepa and published in 1999 [CML 2005]. Molecule data stores reaction data and data of crystallography. There are two profiles of CML at this time: coreCML and fullCML. coreCML is a subset of the full version and is mainly for representing small molecules. The structure of a document is given more firmly in coreCML rather than fullCML

[Kroen 2003]. In the following paragraph, a CML document will be shown. There is some source available on CML site for converting PDB2CML [Zara 1999]

```

<molecule>
  <atomArray>
    <atom id="a_1">
      <float builtin="x3" units="A">1.0303</float>
      <float builtin="y3" units="A">0.8847</float>
      <float builtin="z3" units="A">0.9763</float>
      <string builtin="elementType">C</string>
    </atom>
      :
    <atom id="a_9">
      <float builtin="x3" units="A">3.7056</float>
      <float builtin="y3" units="A">2.1820</float>
      <float builtin="z3" units="A">2.1139</float>
      <string builtin="elementType">H</string>
    </atom>
  </atomArray>
  <bondArray>
    <bond id="b_1">
      <string builtin="atomRef">a_1</string>
      <string builtin="atomRef">a_2</string>
      <string builtin="order">1</string>
    </bond>
    <bond id="b_2">
      <string builtin="atomRef">a_1</string>
      <string builtin="atomRef">a_4</string>
      <string builtin="order">1</string>
    </bond>
      :
  </bondArray>
</molecule>

```

Example 4.1 An example of molecule CML structure

An example of a typical CML file is shown in Example 4.1. The file starts with root element “molecule”, with all other information as children inside of the molecule element. First, all atoms in the element atomic array are listed. Each atom element possesses the same structure. The attribute ID identifies an atom clearly. An atom contains float elements, which describe the exact coordinate positions in the molecule,

and a string element, which specifies the chemical type of element. After the listing of the atoms the connections between the atoms are specified.

The connections are described by one bond for each element. Each bond element is exactly characterized by the clear attribute ID as well. The Bond element contains three string elements, where the first two elements are atom's IDs and ID attribute value, between which atoms the connection exists. Last string of the element indicates the type of the connection.

CML is not able to be visualized as 3D structure. After PDB to CML conversion, we'd like to visualize high quality imaging molecule structures in 3D scene, especially on the Web. Oliver did a conversion from CML to X3D. The extensibility which X3D inherited from XML allows a new set of prototypes to be defined for chemistry elements and attributes. These prototypes were used for converting from CML to X3D. Poly also did a conversion from CML to X3D/VRML [Polys 2003]. These conversions are an important step to bring the 3D molecular dataset to be visible. However, duo to the low quality shading algorithm in VRML/X3D, we can interact with the 3D scene, but not with very realistic images. Adding shaders into the 3D structure will rapidly improve the realism. It brings more immersive environment for biology researches. We will discuss our design and implementation of how to automatically add shader in during the conversion of CML to X3D in section 4.2.

```

<molecule convention="MDLMol" id="nicotine" title="CAFFEINE">
  <formula>C8 H10 N4 O2</formula>
  <atomArray>
    <atom id="caffeine_karne_a_1" convention="mol">
      <float builtin="x3" units="A">-2.8709</float>
      <float builtin="y3" units="A">-1.0499</float>
      <float builtin="z3" units="A">0.1718</float>
      <string builtin="elementType">C</string>
    </atom>
    .
    .

    <atom id="caffeine_karne_a_24" convention="mol">

      <float builtin="x3" units="A">-2.0682</float>
      <float builtin="y3" units="A">-3.5218</float>
      <float builtin="z3" units="A">1.1381</float>
      <string builtin="elementType">H</string>
    </atom>
  </atomArray>
  <bondArray>
    <bond id="caffeine_karne_b_1" convention="mol">
      <string builtin="atomRef">caffeine_karne_a_1</string>
      <string builtin="atomRef">caffeine_karne_a_2</string>
      <string builtin="order" convention="MDL">1</string>
    </bond>
    .
    .

    <bond id="caffeine_karne_b_25" convention="mol">
      <string builtin="atomRef">caffeine_karne_a_14</string>
      <string builtin="atomRef">caffeine_karne_a_24</string>
      <string builtin="order" convention="MDL">1</string>
    </bond>
  </bondArray>
</molecule>

```

Example 4.2 an example of caffeine CML structure from [Polys 2003]

The format of CML by Olivers is the format that Polys used to develop his XSLT for CML2X3D. Example 4.2 is a CML format of a caffeine structure example. Since Polys's code base is more accessible, our work will be based on his.

4.1.1.3 X3D/VRML

For viewing the CML or PDB file as a 3D structure over the web, the most popular format is VRML/X3D. We briefly introduced the syntax and structure of both VRML and X3D in chapter 2, and how these 3D structures are displayed in a 3D

However, only displaying a 3D scene of molecule structure is not enough for purpose of doing research on structural biology. After converting a CML file to a shader supported X3D scene, we also developed a set of powerful functions for convenient user interactions. The next few paragraph, we will talk about some fundamental components we can used to implement those functions in VRML/ X3D with shaders.

ROUTE:

Since VRML 2.0, web 3D had the ability to interact with the 3D world. This function was realized by "Event " based concept, which was taken by X3D completely from VRML. Sensors register each action from users and release whenever certain events been called. These sensors and routes allow developers to write some condition scripts for controlling certain nodes with some desired response. Let's see some detail about the communication between nodes in VRML. A scene could be as simple as two nodes; both have some attributes. When there is a communication between nodes, an event path has to be set up between them.

Some attributes of nodes are changed when events are exchanged. When the event happens, the node needs to be identified and the data fields (or attributes) between which communication is required as indicated. It is critical how the data field is to be accessed. The following four field types can receive or send data in VRML2.0/X3D.

- eventIn (inputOnly) : Over this field type a node can receive events at run-time.

- eventOut (outputOnly) : This field type makes it possible for a node to send events. These can be received and processed then from one input field of another node.
- exposedField (input / output) : Over this field type both events can be taken up and sent away, this field more like a global variable in C, it can be changed and then noticed anywhere in the program.
- field (initializeOnly) : No new values can be assigned to fields of this type at runtime. However it can be assign data by “IS” from the predefined field.

In Figure 4.5, Node A sends event out to node B by ROUTE1, 2. Two field of node B received the data. In code fragment Example 4.3, the function ROUTE of the concept illustrated in both VRML and X3D. For the indication of a ROUTE, the attribute (field) names and the node names are indicated. In X3D field attributes indicated the input and output nodes. In VRML, The node names must be specified in addition before the DEF field.

Besides the suitable field types, the data types of both fields must fit with each other. Only fields with identical data types can be interconnected with updated values. A Script can connect fields with different data types indirectly with their data conversion. We will show a few examples later in this chapter. More complex events are processed by implementing functions. These functions can be written either directly in the file as JavaScript / vrmlScript, or externally in a Java class. Those features really make this

simple script language very flexible and as powerful as other programming languages. For controlling objects in the scene, VRML/X3D also provide us with some handy sensors.

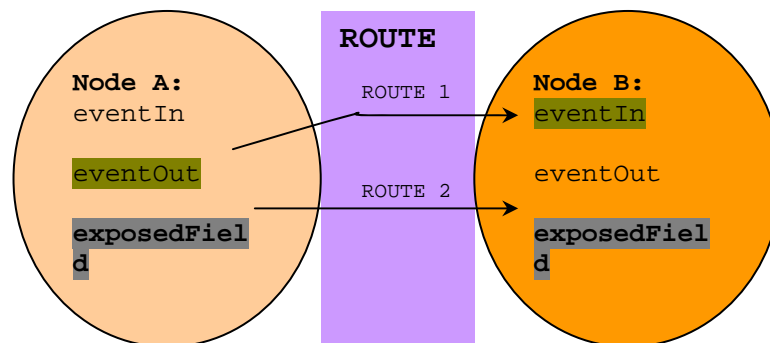


Figure 4 - 5 : Node ROUTE function in VRML/X3D

```
< ROUTE fromField="eventOut "A" fromNode="node toField="eventIn "toNode="node B"/>
# the above is in X3D and the following is in VRML
ROUTE Node.eventOut_changed TO Node.set_eventIn
```

Example 4.3 example of ROUTE format in X3D/VRML

SENSOR:

In VRML and/or X3D different sensors can be defined. The most useful sensors we used are explained in detail.

- **ProximitySensor**

With this sensor one can determine the exact position of the viewer of a scene. The data field specifies the center and the size of a region. Once the viewer enters this

region, value “TRUE” is assigned to the field “isActive”. Within the region, it returns a value of *position_changed* and *orientation_changed*. ProximitySensor needs a size, so that if the user is outside the virtual box then the ProximitySensor will NOT generate events. To avoid viewer jump out of the active region, we can always define the *size* of the ProximitySensor to be larger than the world itself.

- **TouchSensor**

This sensor makes the processing of mouse actions possible. If the mouse pointer is over an object, a value “TRUE” is assigned to the data field. If the mouse is moved from the object, the value is changed to “FALSE”. The TouchSensor has a single field which specifies if the sensor is “enabled” or not. The “isOver” event is generated with the value “TRUE” by this node when the field is “enabled” and the mouse moves from a position where it is not over a shape contained within the group to a position where it is over a shape. A value “FALSE” is generated by this event when the sensor is “enabled” and the mouse over a shape within the group. We can define a route to change a touched object’s texture and/or color. When the mouse is over a shape as a TouchSensor, then the events of *hitPoint_changed* , *hitTextCoord_changed* and *hitNormal_changed* are generated when the mouse moves. Example 4.4 shows touch sensor change color example in X3D.

In this example, a red ball is defined, which is provided with a TouchSensor . If one activates the sensor by one mouse click, an appropriate event is sent at the Script node

```

<Scene>
  <Transform DEF="trans" translation="3 0 0">
    <TouchSensor DEF="touch"/>
    <Shape>
      <Sphere radius="2"/>
      <Appearance>
        <Material diffuseColor="1 0 0"/>
      </Appearance>
    </Shape>
  </Transform>
  <Script DEF="schieben">
    <field accessType="inputOnly" name="input" type="SFBool"/>
    <field accessType="outputOnly" name="output"
      type="SFVec3f"/>
    <![CDATA[ javascript:
      function input(){ output = new SFVec3f(6, 0, 0);}]]>
  </Script>
  <ROUTE fromField="isActive" fromNode="touch"
    toField="input" toNode="schieben"/>
  <ROUTE fromField="output" fromNode="schieben"
    toField="translation" toNode="trans"/>
</Scene>

```

Example 4.4 A touch sensor example in X3D from [From Krone 2003]

- **PlaneSensor, SphereSensor, CylinderSensor**

PlaneSensor objects are movable within the X/Y level. Movement is limited by the data field min position and max position. About the field translation the exact coordinates are communicated to the object, which can be shifted. SphereSensor maps the movement to the surface of a conceptual sphere. CylinderSensor Maps the movement to the surface of a conceptual cylinder.

Example 4.5 shows an example of using a PlaneSensor to move a Sphere in a rectangular area defined by (-1,-1), (1, 1).

```
#VRML V2.0 utf8
Group {
  children [ DEF ps PlaneSensor {
    minPosition -1 -1
    maxPosition 1 1
  } DEF tr Transform {
    children Shape {geometry Sphere {}}
  } ]
}
```

```
ROUTE ps.translation_changed TO tr.set_translation
```

Example 4.5 a plane sensor move a sphere in VRML

One might ask where “translation_change” and “set_translation” from? Actually, “isActive” (Boolean), “translation_changed” (SFVec3f) and “trackPoint_changed” (3D point) are predefined variables in each of PlaneSensor. The value of “set_translation” will be assigned to the new value of “translation_change”. “translation_change” would be the new location of the object.

- **PROTO DEFINITION IN VRML AND/OR X3D**

Besides the given nodes in VRML / X3D, we can define our own object nodes. That makes sense if an object occurs several times in an easily modified form in a scene. For example, if we have a stool with a seat face and four legs. The legs have different colors, a very special stool. If a leg is described by prototypes with its color changeable, each individual leg does not have to be defined separately. This is the same thing as a virtual function in C++.

Another mechanism, with which objects in a scene can be reused, is the “DEF/USE” concept. An object is defined over its names with the keyword “DEF”, which make the object referable in further scenes. With “USE” calling this defined node, it can be used arbitrarily as often as needed. Create instance with “USE”, only the exactly node is reloaded, which was specified by DEF. During this process, no variations can be inserted. For example, the legs in the stool example can not be created in different with “DEF/USE”. So if we want to customize the object, PROTO should be used. In VRML a prototype with the node “PROTO” is defined in X3D with “ProtoDeclare”. The concept is identical in both cases but with different syntax. Before an object is used in the scene, the exact structure must be fixed. That is defined in the node “ProtoDeclare” in X3D, or “PROTO” in VRML. The further of organization prototypes in X3D are divided into two parts: “ProtoInterface” and “ProtoBody”. In the declaration section, which is defined by the node ProtoInterface, the different data fields in prototypes are specified, and as many fields can be put in as desired. They are eventIn, eventOut, exposedField and field. They are the interface when the instance object communicates with other nodes using ROUTE. Following “ProtoInterface” section, “ProtoBody” with the actual characteristics prototypes are specified. In the following example, the default values of fields from the declaration section are used. In order to specify different appearances, an initialized value should be assigned to each data field. This will produced different instances of the prototypes.

In the program Example 4.6, a C_ball is defined by a set of parameters in X3D. The first element of the declaration of C_Ball, ProtoDeclare, is indicated. In this case the

name “C_Ball”. In the following declaration three data fields are defined. The first data field is named “translation”, the field type “input/output” and the data type “SFVec3f”. The structure of the other data fields is similar. When we define different balls for different atoms, they are only data field with different input values. ProtoInterface are needed.

```

< ProtoDeclare name="C_Ball">
  <ProtoInterface>
    <field accessType="inputOutput" name="translation"
      type="SFVec3f"/>
    <field accessType="initializeOnly" name="radius"
      type="SFFloat"/>
    <field accessType="inputOutput" name="color"
      type="SFColor"/>
  </ProtoInterface>
  <ProtoBody>
    <Transform DEF="s_transform">
      <IS>
        <connect nodeField="translation"
          protoField="translation"/>
      </IS>
    <Shape>
      <Sphere DEF="sphere">
        <IS>
          <connect nodeField="radius" protoField="radius"/>
        </IS>
      </Sphere>
    <Appearance>
      <Material DEF="s_mat">
        <IS>
          <connect nodeField="diffuseColor"
            protoField="color"/>
        </IS>
      </Material>
    </Appearance>
  </Shape>
</Transform>
</ProtoBody>
</ProtoDeclare>

```

Example 4.6 Example of prototype define in X3D

This could be realized with some connecting attribute, which is a “IS”. “IS” is an element can only be used inside the prototype definition. It is the transportation who carries the input value from interface to the local variable, like data field as “field”. The object receives the information from the declaration about the data field name. The individual data fields are still initialized with the node field value when it is instanced. The exposeField data field is a global variable which can receive updated data from anywhere. An instance of C_Ball was initialized as follow, Example 4.7.

```
<ProtoInstance name="C_Ball">  
  <fieldValue name="translation" value="10 0 0"/>  
  <fieldValue name="radius" value="2"/>  
  <fieldValue name="color" value="1 0 0"/>  
</ProtoInstance>
```

Example 4.7 Example of instance of C_Ball in X3D

For visualize CML files, they can be converted to the X3D/ VRML format [Polys 2003], which can be viewed on the Internet. A set of chemical structures has been successfully converted to X3D and VRML formats and viewed on the Internet with an Extensible Stylesheet Language Transformation (XSLT) translation sheet available. This Translation sheet is what we used to implement shaders into the X3D scene.

The standard VRML shading algorithm [Gouraud 1971] can only produce relatively low quality images with smeared specular highlights, especially with low polygon count models. By defining a set of standard procedural fragment shaders for a small set of atoms and applying these shaders to molecular structure visualization in X3D, we are able to visualize higher quality 3D molecular structures over the Web and still allow real-time user interaction. In this dissertation, we will take advantage of

graphic hardware by displaying the shading of each element of molecule and computing its shading on the GPU instead of the CPU, allow real-time rendering and interactive protein structures on the World Wide Web.

4.1.2 Software packages and other techniques for visualizing bioinformatics on the Web

The PDB file format of 3D molecule structure is widely supported by various 3D visualization and/or modelling applications. It could be viewed in some of standalone package like Rasmol [Rasmol 2005]. With the internet developing so rapidly today, there are many collaborations between institutions or research centers. The web-based interactive realistic visualization will take a very important role in bioinformatic research.

Before X3D appeared, the ability to display 3D molecular structures with a 3D viewer like VRML, the PDB files were first transformed into VRML 3D scenes. There are a few tools available to convert PDB files into VRML 3D models.

A small open-sourced Perl script named `pdb2vrml.pl` [pdb2vrml-perl 2005] is available for converting PDB into VRML but only VRML 1.0 and the conversion is limited in the space-fill mode. A free Windows executable called `pdb2vrml.exe` [pdb2vrml-chem] is also available, its current version is 1.4. The program is written in ASCII C by David N. Blauch. It converts PDB into VRML 2.0 in spacefill, ballstick and wireframe mode. It only works from from Window prompt or command, and is limited to the number of atoms, it can convert, to 1337 as experiment found out (test was done on a desktop in HVL lab).

A free Unix/Linux binary, `pdb2vrml` [PDB2VRML-vrml 2005] is available from Institute für Physikalische Chemie. The authors also provide a web interface for converting PDB to VRML. [PDB2VRML-vrml 2005]

A small open source C++ library, `pdb2vrml`, was created by Vieri Di Paola [pdb2vrml-c]. It is incomplete and not well maintained since 1997. A program called MolScript; [molscript 2005] developed in ASCII C and older versions, 1.4 or earlier, were written in Fortran 77. The source code and UNIX binary are distributed under licenses. The MolAuto component of MolScript can generate VRML presentations of 3D structures. The MolScript is powering the VRML web interface at the protein data bank [PDB 2005].

There are a few online conversion services via a web interface. The ChemVis [Chem-vis 2005] project provides an online VRML file creator, allowing users to input a PDB file. The protein data bank has a default 3D viewer in VRML and also provides a more detailed VRML file rendering.

Software such as Spartan [spantan 2005], MOLDEN [molden 2004] and Molda [molda 2005] provide functions for saving 3D scenes into VRML. A cross-platform application with an easy-use graphics user interface, for converting PDB to VRML named MolVRML is in development as a platform independent, standalone program with a graphic interface in Java. This give us a hope, however, they are still far from real-time interactive realistic rendering.

A paper published by Franklin Oliver in 1999, named “*Texture-based Volume Visualization for Multiple Users on the World Wide Web*” [Engel and Ertl 1999], presented a texture-based volume visualization tool, which permits remote access to radiological data and supports multi-user environments. The application uses JAVA and VRML, and therefore platform-independent and able to use fast 3D graphics acceleration hardware of client machines. The application allows shared viewing and manipulation of three-dimensional medical datasets in a heterogeneous network. However, this application was only able to display data 28 x 128 x 64 data set at 7 fps and 256 x 256 x 128 dataset at 2 fps. Which is far away from real-time by today’s standard. In later implementations, the advantage of programmable graphics card will be used for the better performance of bioinformatics visualization.

There is much research being done for Web based molecular structure visualization. [ChemVis 2004] [Zou 1999] [Perrakis et al. 1999] [Badger 2003] Most use (VRML). However, due to the low image quality of the shading/modeling algorithm in VRML and the limited CPU rendering capability, it is difficult to render high quality images of large proteins in real-time. Some of the packages, such as Raster3D [Raster3D], can generate high quality images for PDB structures, but don't have the capability of sharing such information over the Internet. None of the implementations can achieve web based high quality visualization and real-time interaction for large protein visualization.

The introduction of programmable Graphics Processing Units (GPUs) and the addition of procedural shaders to the web3D standard X3D provide us with new

techniques to develop real-time Web based visualization. In the last section, we discussed the applications of these techniques to bioinformatics and chemistry visualization, in the next section; we will discuss our design and implementation of taking advantage of procedural shaders for bioinformatics visualization, specifically the visualization of large biomolecules. By using procedural shaders, we are able to produce higher quality visualizations with minimal performance penalty. We have developed methods to automatically convert from the standard bioinformatics PDB format to CML and then to X3D. The procedural shaders are automatically inserted during the CML to X3D conversion. This provides higher quality images and leads to future possibilities of more flexible and enhanced visualizations.

4.2 Designs and Implementations shader-X3D for molecule presentation

4.2.1 Standard shader for minimal element set for protein presentation

4.2.1.1 Design

In chapter 1, we discussed the use of programmable GPUs and how they enable the creation of real-time high quality visualization. In section 4.1 we introduced background for viewing a given 3D structure PDB file of proteins, and the need for converting the PDB to CML format and then to X3D. During CML to X3D conversion, we applied shaders to the protein atoms in order to get a real-time high quality interactive 3D visualization.

After the CML2X3D and shader conversion, the next step is to make the structure displayable via the Web. Because of the limited resource of X3D Viewer, we have to view the shaders in the VRML format in a web browser which has the BS v6.2 plug in installed, Bs v6.2 successfully supported shader VRML browser with DirectX 9.0. And therefore, a conversion from X3D to VRML is required as the last part of presentation.

As already discussed there is a Extensible Stylesheet Language Transformation (XSLT) (CML2X3D [Polys 2003]) to convert from CML to X3D. The only required modification to CML2X3D is adding the shader's transformation into the style sheet. So the insertion of shaders happens between the CML to X3D conversion as shown in Figure 4.6.

The implementation of the pipeline is done in three steps. First, convert the PDB file to CML file format; we talked about the bond calculation from PDB to CML in Oliver's thesis. However, we do not have the source or executable files. We used PDB2CML from [Zara 1999]

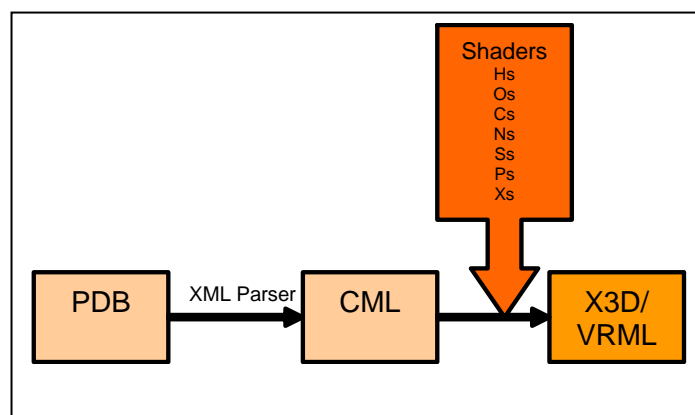


Figure 4 - 6 : PDB → CML & CML → Shaded X3D Conversion

For the size molecule translation, it does have a limitation of up to 1337 atoms on some machines. Most molecule sizes will be much larger than that. Secondly, define a set of different shaders as standard for “Carbon”, “Nitrogen”, “Oxygen”, “Hydrogen”, “Phosphorus”, “Sulphur”, and “Other”; third, embed the standard shaders set into the CML2X3D translation sheet. Since we are primarily interested in biomolecules, especially proteins and RNA/DNA, we only required the above small set of atoms, since the six defined atoms account for over 99% of all the atoms in these molecules. “Other” is for the occasional different atom, e.g., Iron in Myoglobin.

After creating the X3D file, users can download and view the 3D protein structure interactively as long as they have an X3D/VRML (currently only VRML format supported) viewer or plug in for their browser. However, there is a problem when sharing the element shaders since the shaders are platform dependent and we can't assume all users are working on the same platform (operating system or graphics card). To solve this problem, SLC is used as a tool to convert shaders into XML format before publishing and converting to the compatible shading format, which is supported by the destination platform. We showed the function of SLC in chapters 3 and 4.

For example, a shader written by a developer in HLSL may be displayed on the developer's machine nicely. However, it may not be displayed on a client's machine, because that client's machine does not support HLSL. In this case, SLC helps to translate shaders into XML format and then convert it to the shading language supported by the second client's machine, for instance, GLSL.








4.2.1.2 Implementation

Defining a set of shaders for basic elements

We define a set of different shaders as standard for 'Cs', 'Ns', 'Os', 'Hs', 'Ps', 'Ss', and 'Xs' which represents shaders for "Carbon", "Nitrogen", "Oxygen", "Hydrogen", "Phosphorus", "Sulphur", and "Other" respectively. We used a slightly modified CPK color scheme and set 'Other' to green for easier distinction from the other standard colors. (In the CPK standard, "other" is deep pink which is very similar to Oxygen which is red)

Table 4.1 shows the result of applying the fragment shader of each element with Blinn-Phong per-pixel shading on “Carbon”, “Nitrogen”, “Oxygen”, “Hydrogen”, “Phosphorus”, “Sulphur”, and “Other”.

Table 4 - 1 Shader definition for basic elements in Molecular Structure

	Os	Ns	Cs	Hs	Ss	Ps	Xs
Phong shader applied							
Surface Material	1.0, 0.0, 0.0	0.6, 0.6, 1.0	0.5, 0.5, 0.5	0.9, 0.9, 0.9	1.0, 0.8, 0.2	1.0, 0.65, 0.0	0.0, 1.0, 0.0

In this table “Surface Material” refers to the ambient and diffuse reflection coefficients. The specular reflection coefficient was set to white (1.0 1.0 1.0). The user can modify these values if they wish. We used the [Phong 1975] illumination and per pixel shading algorithm except that we used the [Blinn 1977] half angle lobe ($\text{dot}(N,H)$) rather than the Phong mirror reflection lobe ($\text{dot}(V,R)$). [Ngan et al. 2004] showed that this decreases the average fitting error by 40% because at low light angles it gives a more correct shape of the specular lobe than does the normal mirror reflection angle. Table 1: Basic set of shaders for atoms

We start by converting the PDB files to the CML format with the XML Parser, Xerces 2.0 based on PDB2CML[Zara 1999]. Then, we added pre-defined shaders as nodes (as shown in Table. 1), during the CML to X3D translation. We updated Polys’ CML2X3D by adding the shader nodes translation. The format of the defined shader

nodes is based on the web3D shader group proposal [web3D-Shader-Group 2005] and is also compatible with the VRML browser BS Contact by Bitmanagement [Bitmanagement 2005]. The shader can be a separate file loaded as a URL into the conversions with some adjustable parameters.

Let's go through an example of loading shaders in the XSLT translation sheet during the CML2X3D translation. For instance, the vertex and pixel shaders of "Carbon" defined and loaded in CML2X3D XSLT are as follows. A vertex and fragment shader example of "Carbon" in Cg is shown in example 4.8.

```
// Vertex Shader
void main(in float4 position : POSITION, //in object space
         in float4 position1:POSITION,
         in float4 normal   : NORMAL,   //in object space

         //mandatory parameters
         uniform float4x4 modelViewProjection,
         uniform float4x4 model,
         uniform float4x4 modelIT,
         uniform float3 viewPosition,

         //output parameters
         out float4 oPosition : POSITION, //
         out float3 oObjectPos : TEXCOORD0, //
         out float3 oNormal   : TEXCOORD1 //
        ){
float4 nposition =
float4(position.x,position.y,position.z,position.w);
oPosition = mul(modelViewProjection, nposition);
//transform the vertex position and normal into World Space:
oObjectPos = mul(model, position).xyz;
oNormal = mul(modelIT, normal).xyz;
}
```

```

// Fragment shader
void main(in float4 position : TEXCOORD0, //in world space
         in float3 normal    : TEXCOORD1, //in world space
         uniform float3 viewPosition, //mandatory in world space
         uniform float3 baseColor,
         uniform float3 lightPosition, //defined in world space
         out float4 oColor : COLOR){
float3 lightColor = float3(1.5f, 1.5f, 1.5f);
float3 P = position.xyz;
float3 N = normalize(normal);

//get lightPosition into space
lightPosition.z = -lightPosition.z;
//compute the diffuseColor value, assume lightColor is white
float3 L = normalize(lightPosition + P);
float diffuseLight = max(dot(N, L), 0);
float3 diffuse = baseColor * lightColor * diffuseLight;
float3 V = normalize(viewPosition - P);
float3 H = normalize(L + V);
float specularLight = pow(max(dot(N, H), -dot(N, H)), 256);
if(diffuseLight <= 0)
    specularLight = -specularLight;
float3 specular = lightColor * specularLight;

oColor.xyz = (diffuse+ .3*specular)/1.5;
oColor.w = 0.5;
}

```

Example 4.8 Carbon vertex and fragment shader in Cg

ShaderAppearances, vertex and fragment shader node prototype according to bitmanagement is defined in VRML as Figure 4.7, 4.8, 4.9:

```

ShaderAppearance{
  exposedField SFNode  fragmentShader  NULL
  exposedField SFNode  material        NULL
  exposedField SFNode  texture         NULL
  exposedField SFNode  textureTransform NULL
  exposedField SFBool  transparent     TRUE/FALSE
  exposedField SFNode  vertexShader    NULL
}

```

Figure 4 - 7 : ShaderAppearance Prototype in VRML

Shader Appearance node defined as support transparent function. This function is effective until the alpha value of the output colors is less than one in fragment shader, and is listed in the fragment shader code as "oColor.w = 0.5;".

```
VertexShader{
  exposedField MFString url []
  exposedField SFBool mustEvaluate FALSE
  exposedField MFString paramName []
  exposedField MFString paramType []
  # fields ...
}
```

Figure 4 - 8 : VetexShader Prototype in VRML

```
FragmentShader{
  exposedField MFString url []
  exposedField SFBool mustEvaluate
  exposedField MFString paramName []
  exposedField MFString paramType []
}
```

Figure 4 - 9 : FragmentShader Prototype in VRML

Our implementation is a two steps process. First, redesign prototype for the minimal set elements in X3D format; secondly, adding the corresponding shader / vertex shader. All elements of fragment shader are defined is a Document Type Definition (DTD) file. For smoothly showing shaders in VRML browser by Bit management, the second step uses the defined shader nodes based on bitmanagement nodes definition, shown above. The prototype of the "Carbon" nodes in CML2X3D.xsl [Polys 2003] are shown as Figure 4.10:

```

<ProtoDeclare name="Carbon">
  <field IS="atoC.translation" accessType="exposedField"
  name="position" type="Vector3Float"/>
  <field IS="atoC_mat.transparency" accessType="exposedField"
  name="MAT" type="Float" value=".6"/>
  <Transform DEF="atoC">
    <Group>
      <Transform>
        <Shape>
          <Appearance >
            <Material DEF="atoC_mat" diffuseColor="0 0 0"
            shininess="0.8" specularColor=".29 .3 .29"
            transparency="0.6"/>
          </Appearance>
          <Sphere radius=".77"/>
        </Shape>
      </Transform>
      <Transform>
        <Shape>
          <Appearance/>
          <Text string="C">
            <FontStyle size=".8"/>
          </Text>
        </Shape>
      </Transform>
    </Group>
  </Transform>
</ProtoDeclare>

```

Figure 4 - 10 ProtoDeclare for Atom - Carbon without shader

The following changes we made based on Poly's program CML2X3D. (1) added shaderAppearance , vertex and fragment shader elements based on elements defined in X3D; (2) define conversion from X3D to VRML for shader node in X3D2VRML97, which is from web3D[web3D-ToolKits 2003].

```

<ProtoDeclare name="element">
.....
  <Shape> .....
    <ShaderAppearance transparent = "TRUE">
      <VertexShader url=" element -v.cg">... </VertexShader>
      <FragmentShader url=" element -f.cg">...</FragmentShader>
      <Material ...> ... </Material>
    </ShaderAppearance>
    <Sphere radius=".6"/>
  </Shape>
.....
</ProtoDeclare>

```

Figure 4 - 11 Shader Declarations for the ProtoDeclare of any atom

For any of the atoms, its ShaderAppearance element in X3D should follow the format shown in Figure 4.11. For the element “Carbon”, the shaderAppearance section of the prototype is shown in the Figure 4.12.

```

<ShaderAppearance transparent = "TRUE">
  <VertexShader url="Carbon-v.cg"/>
  <FragmentShader url="Carbon-f.cg">
    <field accessType="inputOutput" name="baseColor"
      type="SFColor" value ="0.6 0.6 0.6"/>
    <field accessType="inputOutput" name="lightPosition"
      type="SFVec3f" value ="-10 10 -10"/>
  </FragmentShader>
</ShaderAppearance>

```

Figure 4 - 12 Shader Declaration for the ProtoDeclare of Atom – carbon

The two fields of “baseColor” and “lightPosition”, with its predefined color values, are the input to the shader uniform variables “baseColor” and “lightPosition” defined in Cg or GLSL. For our predefined shader for each of the basic elements the difference between each element is the baseColor value in table 1. The external link of “Carbon-v.cg” and “carbon-f.cg” will send data from the X3D file to shader program and the return value will be sent back to X3D and displayed on the screen. Because of the

similar structure of shaders for different atom, a “ProtoDeclare” can be used in a uniform interface and instance them with different color variable.

After designing the prototype of the shader’s in X3D, we converted the CML files to the X3D format which should be the same format as the prototype is shown in the Figure 4.13.

```
<ShaderAppearance transparent = "TRUE">
  <VertexShader url="Carbon-v.cg" />
  <FragmentShader url="Carbon-f.cg">
    <field accessType="inputOutput" name="baseColor"
      type="SFColor" value ="0.6 0.6 0.6"/>
    <field accessType="inputOutput" name="lightPosition"
      type="SFVec3f" value ="-10 10 -10"/>
  </FragmentShader>
</ShaderAppearance>
```

Figure 4 - 13 Shader for the Atom – carbon after converted

The CML file is converted to X3D with defined shaders. Because currently only VRML browsers are supporting shaders, to display the result, we have to convert the X3D file format to VRML. In order to display the shaded protein structure, we modified the X3D2VRML97 [web3D-ToolKits 2003] XSLT by adding the ShaderAppearance / VertexShader / FragmentShader nodes into the XSLT. We use VRML browser BS V6.2 [Bitmanagement 2005] because it supports shaders. The nodes defined in the new version of the XSLT for X3D2VRML are compatible with the browser and the shader format specified in the web3D shader group proposal. For example, a new node of “ShaderAppearance” with its “transparent” attribute has been added into the XSLT with its subnodes of VertexShader and Fragment Shader. After translating each of the shaders

of the elements, the protein structure can be viewed in a shader supported VRML browser on a single machine.

For example, for “shaderAppearance” elements, the node will be same name as “shaderAppearance” in VRML. Same thing works for vertex and fragment shaders. This shaderAppearance element for example can contain at lease six attributes. That means each of the attributes needs to check if their parent node is shaderAppearance. If yes, write nodes with format of shaderAppearance. If no, continue check for other nodes. Refer to Figure 4.14.

```
<xsl:when test="local-name()='ShaderAppearance' or
$nodeType='ShaderAppearance'
or $EPnodeType='ShaderAppearance' or
@nodeType='ShaderAppearance' ">
<xsl:text>appearance </xsl:text></xsl:when>
```

Figure 4 - 14 ShaderAppearance definations in XSL modification

For the child node to determine its structure, we need to check if their parent is shader appearance. An example is shown here for element node movieTexture shown in Figure 4.15:

```
<xsl:when test="(local-name()='MovieTexture' or
$nodeType='MovieTexture' or .....
and (local-name(..)='ShaderAppearance' or
$parentType='ShaderAppearance' or
$EPparentType='ShaderAppearance' or
../@nodeType='ShaderAppearance' or
local-name(..)='texture') "><xsl:text>texture
</xsl:text></xsl:when>
```

Figure 4 - 15 MovieTexture tag definition modification in XSL with shader added in

The above simply shows the steps to add nodes according to the shader defined in both X3D and movieTexture. The same will happen to all shaderAppearance children and ShaderAppearance parent node as well, i.e., vertexShader / fragmentShader, material, texture, textureTransform, transparent.

After the conversions from PDB2CML and from CML2X3D with shader, Shaded-X3D2VRML also produced by X3D2VRML.

The command of converting shader from CML→Shader-X3D→Shader-VRML is shown as:

```
saxon -o riboflavinN.xml.x3d riboflavinN.xml cml2X3d_2ccd-301.xsl
saxon -o riboflavinN.xml.x3d.wrl riboflavinN.xml.x3d
x3dToVrml97_2275.xslt
```

Figure 4 - 16 cml2x3d_shader/ x3d_shader2_vrml_shader execute comments

We are ready to show our shader 3D model of any molecule structure. For example in a single machine browser we open a single chemistry riboflavinN.wrl file, shown in Figure 4.17.

To show this 3D structure via the Internet, we have to make sure it can be display on the user's machine correctly. As we know, current existing shader formats are platform dependent. For solving this problem, in this step the Shader Language Converter will be empolysed as a tool

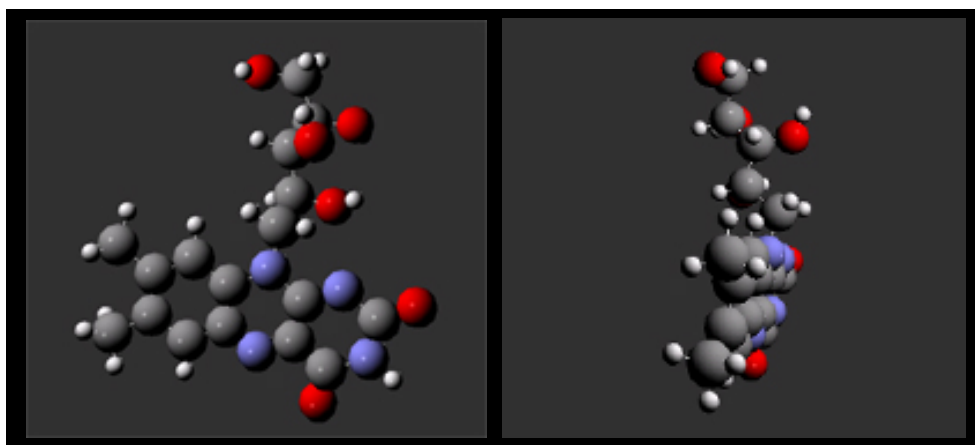


Figure 4 - 17 A converted shader-X3D riboflavinN 3D structure in VRML

```
// VertexShader
uniform mat4 modelViewProjection ;
uniform mat4 model ;
uniform mat4 modelIT ;
uniform vec3 viewPosition ;
void main(void)
{
    vec4 nposition =
    vec4( gl_vector.x, gl_vector.y ,gl_vector.z, gl_vector.w );
    gl_vector =modelViewProjection * nposition;
    gl_MultiTexCoord0.xyz =model * position.xyz ;
    gl_MultiTexCoord1.xyz =modelIT * normal.xyz ;
}
```

Example 4.9 A SLC converted vertex shader in GLSL for riboflavinN 3D in X3D

Before the Shaded 3D model is published and after it's been downloaded to the user's machine. For example, the shaders in GLSL displayed on machine that only support GLSL shader as in Example 4.9, 4.9(a).

In example 4.9 and 4.9 (a), the Shader Language Converter was used to convert shaders from its original source code, written in Cg, to the clients' machine compatible language, GLSL. It should successfully display on the client machine. Both of them passed the test in a stand alone application on a Cg and GLSL compatible platform. Due

to the shortage of the GLSL shader supported VRML/X3D web browser, we can't test the GLSL shader over the web at this moment. Currently the only browser that we are aware of that supports programmable shaders is BS Contact and it only supports HLSL/Cg shaders using DirectX 9 on Microsoft Windows. However, we assume that when the shader proposal is accepted and becomes a part of X3D that other browsers for other platforms will be developed.

The fragment shader is shown in GLSL as follows in Example 4.9(a):

```
// VertexShader -GLSL vertex shader
uniform vec3 viewPosition ;
uniform vec3 baseColor ;
uniform vec3 lightPosition ;
void main(void){
    vec3 lightColor =vec3( 1.5f, 1.5f, 1.5f );
    vec3 P =gl_MultiTexCoord0 .xyz ;
    vec3 N =normalize( gl_MultiTexCoord1.xyz );
    lightPosition.z -lightPosition .z ;
    vec3 L =normalize( lightPosition +P );
    float diffuseLight =max( dot( N, L ), 0 );
    vec3 diffuse =baseColor *lightColor *diffuseLight ;

    vec3 V =normalize( viewPosition *P );
    vec3 H =normalize( L +V );
    float specularLight =pow( max( dot( N, H ),-dot( N, H ) ), 256 );

    if(diffuseLight <=0 )
    {
        specularLight =-specularLight;
    }
    vec3 specular = lightColor * specularLight;
    gl_Color.xyz = (diffuse+ .3*specular)/1.5;
    gl_Color.w = 0.5;
}
```

Example 4.9(a) a SLC converted fragment shader in GLSL for riboflavinN 3D in X3D

4.2.1.3 Performance analysis and visual effect comparisons

Biologists and Chemists work with molecules on a daily basis. 3D virtual protein structures provide them with visual images of the molecules structure. It was our hypothesis that a higher image quality would improve their understanding and

productivity. We evaluated our X3D based biomolecular visualization in terms of user preference and program performance.

Initially, we tested the system on some simple small biomolecules. We produced a set of procedurally shaded X3D biomolecules, which were converted by the new XSLT and the corresponding X3D molecules without shaders and displayed them in IE 6.0 and FireFox 1.0 with the B.S. V6.2 VRML plug-in. Then, we had biology students manipulate the protein structures with and without the shaders. The evidence indicated the higher quality shaded molecules were superior as evidenced by the following comments made by the students:

- Large improvement in image quality;
- Better depth of view made the 3D structure easier to understand.
- Easily view and distinguish the different atoms with their uniform appearance.

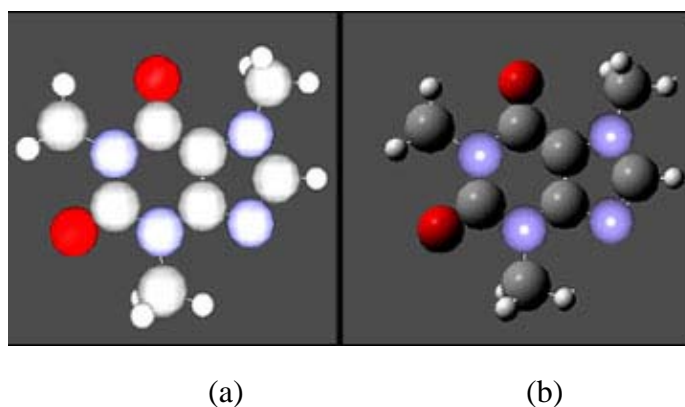
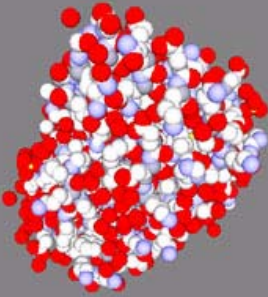
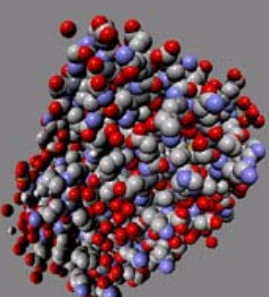
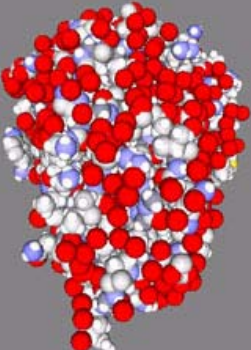
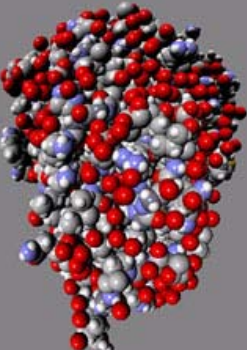
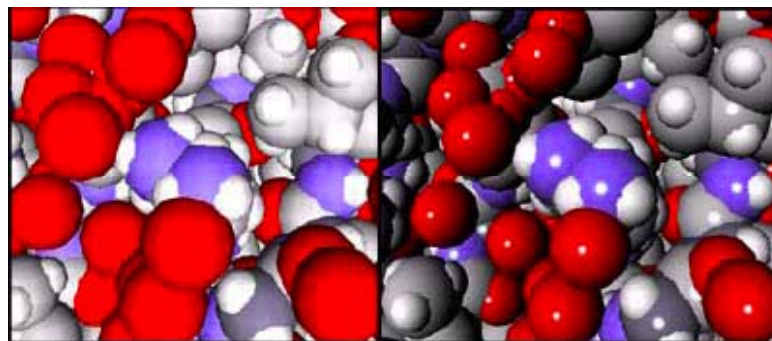


Figure 4 - 18 Caffeine Structure
(a) Solid atoms in X3D/VRML without shaders
(b) in X3D/VRML with shaders from CML X3D

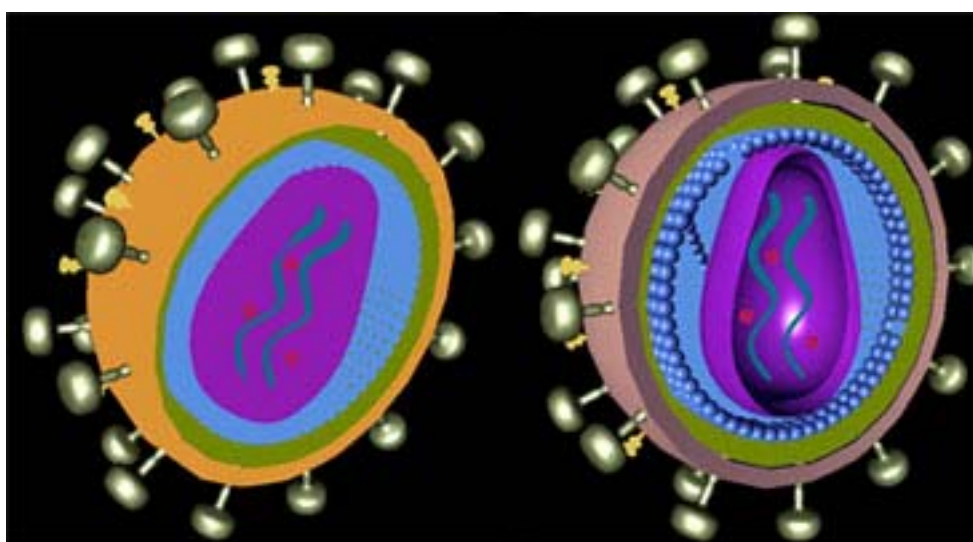
In Figure 4.18, the caffeine molecule is displayed with and without procedural shaders. Next, we applied our shaders to some larger molecular structures the following table 4.2, shows a comparison of the molecule *geobacillus stearothermophilus* carboxylesterase Est30 (1TQH) [Liu et al. 2004] and molecule HIV-1 protease (1S6G) [Tie et al. 2004] 3D structure with and without shaders. Figure 4.21 shows a closer comparison of the protein structure displayed with and without procedural shaders.

Table 4 - 2 Comparison of Est30 (1TQH) and HIV protease (1S6G). Molecular structure with and without procedural shaders

	X3D	X3D + Shaders
1TQH		
1S6G		



(a)



(b)

Figure 4 - 19 (a) a closer comparison of the protein structure with and without shaders
 (b) a comparison of the HIV structure displayed with and without shaders

Figure. 4.20 show two protein structures. On the left is the intermediate result of Est30 which contains 1327 atoms and on the right is a Non-Psychrophilic Trypsin from A Cold-Adapted Fish Species, which contains 7106 atoms. [Schroder etc. 1998]

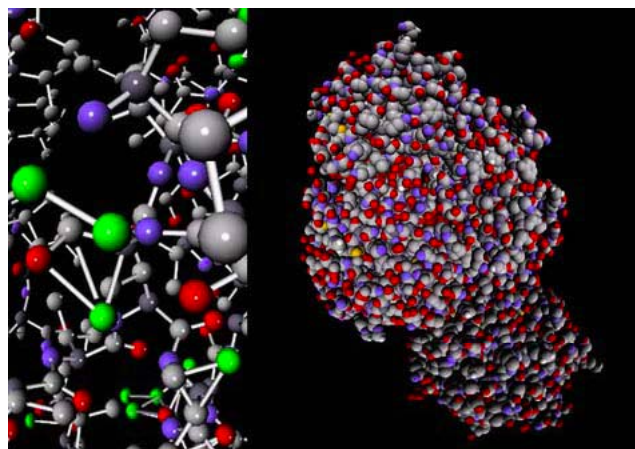


Figure 4 - 20 Left: partially Est30 ball-stick structure with procedural shaders. Right: Trypsin Molecule space-filling structure with procedural shader in X3D/VRML

We evaluated the performance of the shaded version by looking at two metrics: the change in file size caused by adding the shaders and also by the interactive performance. Adding the extra code necessary to implement the shader capability increased the VRML file size by less than three kilobytes. Considering that a small molecule, such as cortisone, has a VRML file size of 22 Kilobytes, this is not much of an increase. A very large protein, e.g., one with 7100 atoms has a VRML file size of 643 kilobytes.

The second measure of performance is in interactivity of the display, that is, the ability to move smoothly through the molecule and rotate it to see different views. We tested on two different machines. On a “low end” machine, a laptop with a 1.6 Ghz Pentium M CPU and a Nvidia 5650 (128 M bytes of graphics RAM), a protein of 1300

atoms was smoothly interactive. However, the CPU usage was 100% during movement. On this same machine the 7100 atom protein was not smooth, it was quite jerky.

The second system was a desktop with a 3 GHZ Pentium CPU and an Nvidia 6600 GT (PCI Express with 128 M bytes of graphics Ram). The 7,100 atom protein was smoothly interactive on this system, even at full screen resolution (1600 x 1200 pixels). The CPU usage was up to 60% for this molecule. Since geometric transformations in VRML are done by the CPU, it appears that the system is CPU limited and not graphics card limited, thus even the older 5650 may have been sufficient given a faster CPU.

Displaying molecule structure effectively is only one of our goals. Designing a user friendly interface for scientist to manipulate with the atoms and freely build structures is our next task.

4.3 Enhanced user friendly interface for molecular presentation

During few discussions with a researcher at the CDC (Center for Disease Control), some desired functions have been specified. For example, freedom to move atoms to a position they want. The ability to highlight a particular atom will be useful during remote collaboration. Based on these desired functions, a set of interface features have been developed for improving molecular 3D presentations.

4.3.1 Design

1. **Add level of Quality (LOQ)** with shaders into the X3D/VRML scenes. This addition reduces rendering consuming time and improves real-time interactions performance.

VRML provides a LOD function. This function defines, for example, one object with multiple resolutions. Different levels of detail will be selected as the viewer gets closer to the object in certain predefined distances. We borrow the traditional LOD idea and create a new concept LOQ, Level of Quality. It means the image quality increased while the viewer getting close to the object. The high quality images achieved by loading shader in.

Why is LOQ better for our basic molecule structures instead of subdivision technology, which dynamically create the sub-polygons for a detail area? Subdivision technology achieved a good performance for very big structures with a lot of detail in particular parts of the object, sharp curves etc. this also means that large big models will have to be treated as one single object. However, for the molecule structure, we treat each atom as a separate object; similar atoms are just copies with a transformed 3D position. We don't have very complex surfaces for a single atom. But we want it to look detailed when we approach it. We use high level of quality with a shader to represent those molecules close to us, while the rest of the molecules can be displayed in low resolution. This will give an extra dimension control of the surface quality.

Benefit: lower numbers of polygons in distant views, high resolution and detailed shading in closer views. This is exactly the goal that we need. Let's simulate the steps when trying to view detail of a particular part of a molecule. First, we rotate it expose the area of interest within the whole structure, at this time the model is relatively far from viewer (low image quality). Then going closer, we begin to see a partial view of the

whole molecule and correct the direction we need to go (middle level). Eventually, we approach the element we want to inspect closely; a very high resolution shaded object showing details of the element is viewable. Another benefit is, comparing to the whole scene shaded, this strategy just shades a small portion of the structure. It should have a very significant performance improvement.

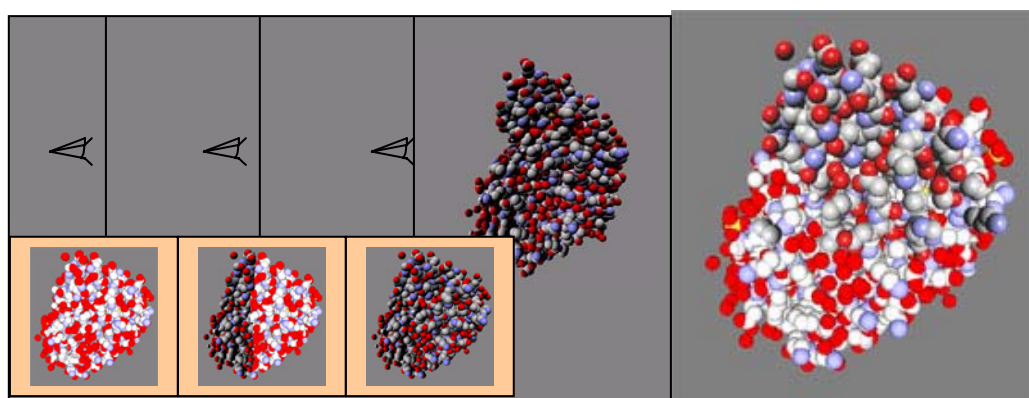


Figure 4 - 21 using LOQ with shader for better performance

2. **Add multiple options** for choosing different shader styles / (different shader select)

This interface includes a few buttons that select different shader styles. The purpose for changing shader styles would be to help clearly show the position and other effects. For example, a transparent shader will give the user the ability to look through objects and view other objects behind or inside. A solid color shader gives a better depth of field.

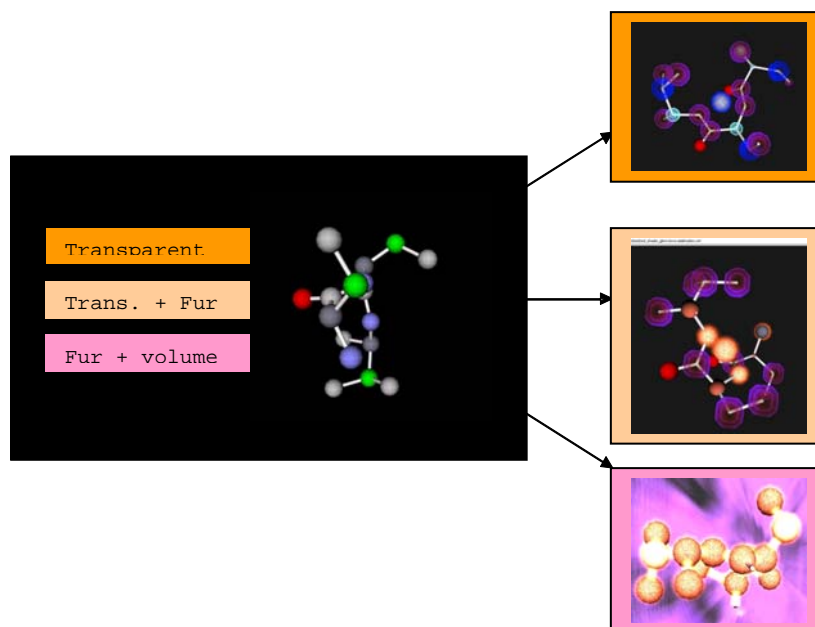


Figure 4 - 22 Multi-shader selector interfaces

3. **Adjustable slider for specular lighting for different atoms** => highlighting the emphasis atom element. (We will show this function in multi-user shard environment)
4. **Interface for building molecular structures easily:** (1). freely moving atoms (2) Adding atoms by functioned path

4.3.2 Implementations

Taking advantage of the VRML language which accepts script, we designed set of prototypes for controlling different atoms in molecules by changing their colors and specular highlights with shaders to identify or highlight the particular objects. Also, we implemented some manipulating methods, like rotation and move certain atoms to help biologist conveniently build their predication of 3D structures. A function based molecule structure generation will also be demonstrated. All demos apply some shaders in some

atoms. Here, we'll show some code fragment or prototypes for realizing different functions.

1. **Add level of quality (LOQ) function with shaders into the X3D/VRML scenes.** This function helps improve real-time interaction performance.

For realizing LOQ function with shaders, we have the following file CH4O_LOD.wrl and three individual level1, level2, and level3 files. In the file CH4O_LOD.wrl, a switch script for distance determine between the viewer and the object. This distance is the control for selecting the different level of quality files. We will use the CH₄O structure to show a simple technique for adding shader in as high-level detail to improve performance. Code fragment in example 4.10, is a piece of switch script that controls level of quality:

```

LOD {
  center 0.0 0.0 0.0
  range [ 7.5, 12.0 ]
  level [
    # High-detail
      Inline { url "CH4O_shader.wrl" }
    # Medium-detail
      Inline { url "CH4O_wrl" },
    # Low-detail
      Inline { url " CH4O_wireframe.wrl" },
  ]
}

```

Example 4.10 Code segment for LOQ control

The level 2 code is as follows

```

Shape {
  appearance DEF MATslategrey_1_75 Appearance {
    material Material {
      diffuseColor 0.439215686275 0.501960784314 0.564705882353
      ambientIntensity 0.5
      specularColor 1.0 1.0 1.0
      shininess 0.75
    }
    geometry Sphere { radius 0.425 }
  }
}

```

Example 10 (a) Low level of quality scene for CH4O

The level 3 code is as follows

```

Shape {
  appearance ShaderAppearance
  {
    vertexShader DEF VertexShader1 VertexShader
    {
      url "BumpPlastic.fx"

      field SFVec4f LightPos 1 0.5 2 0
      field SFFloat Bumpy 5
      exposedField SFNode diffuseMap ImageTexture { url "dna_1-1.jpg" }
      exposedField SFNode normalMap ImageTexture { url "dna_1-1_hf.jpg" }
    }
    .....
  }
  geometry DEF _IndexedFaceSet IndexedFaceSet {
    coord Coordinate {point [.....]}
    texCoord MultiTextureCoordinate {coord [
    TextureCoordinate {point [.....]}
    TextureCoordGen { mode "TANGENT" parameter [ 0 2 ] }}
    coordIndex [.....]
    creaseAngle 3.16
    texCoordIndex [.....]
  }
}

```

Example 10 (b) High level of quality scene for CH4O

In a simple sphere model of low level of quality, we can see the code is very simple. Call primitive shape Sphere and give it a size. However, if we want to show more detail like when doing bump mapping, we have to give the normal index coordinates and texture coordinate index etc, of each vertex. I ignored the detail value to save space in the above level 3 codes, which are a big chunk of data. Please refer to the simple code for LOD with shader, LOQ, in appendix source code section. Level 3, the highest detail,

shows a bump mapping on the surface of a sphere. There is a very high detail of the object shown. Showing this technique for the surface of a single atom might not be very interesting. However, imagine if we have a very large size molecule with many atoms, we'd like to see the whole structure from a distant viewpoint and easily rotate the whole structure. Then move closer to see the details of group of atoms from the whole structure. In this case the LOD + shader will give a quick manipulation with low level of quality and fewer polygons number and high level of quality shader applied for closer views and only to the atoms close enough. That means only those atoms close to you need to be rendered with a shader. This will ensure the real-time performance of the interaction.

As the user gets closer to the 3D structure, the detail of object and the quality of the images will increase as shown in the Figure 4.23:

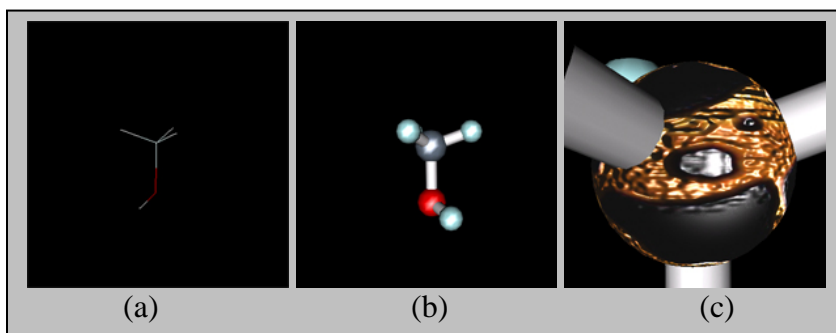


Figure 4 - 23 Level of detail and quality example for CH₄O

(a) Shows the whole structure with single wire frame; (b) CH₄O stick-ball structure without shader; (c) CH₄O structure has C with its bumpy shader

The Figure 4.24 shows the combination of using LOD and shaders. We first defined a level of detail and quality for each atom, and then we assign each a detail level with a distance as the viewer moves closer to the atoms. Whoever closes to the viewer

enough shows their detail. Example 4.11, shows the code fragment for repeat use of the DEF atom Carbon which has shader in its high level detail defined. More complex shaders can be designed with LOD and shader techniques in X3D/VRML. i.e., a shader with changeable ambient color can be changed to darker for distance view.

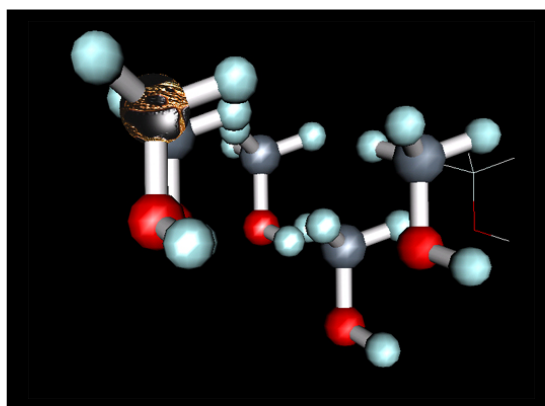


Figure 4 - 24 Level of detail approach while close to model of CH4O

Fragment of code below shows how to call in each of element “C” with different levels of detail.

```
DEF Atom_C_LOD Inline { url " CH4O_LOD.wrl" }
...
USE Atom_C_LOD ...
```

Example 4.11 Reuse predefined C with shader as high level detail

Another example of the level of quality with shader show as following:

2. Add multiple shader styles options

Take advantage of flexibility of VRML interactivity. We designed a button to select different shaders of a demo chemistry structure atom’s surfaces. In the code segment of Example 4.12, we first defined different shader choices, and then we defined

a switch to control the choice iteratively. This is nothing new except we used shader as a multiple-choice object. We'll not show detail of this function.

```

DEF Multistyle Switch {
    whichChoice 0 choice {...shader 1 ...}
    whichChoice 1 choice {...shader 2 ...}
    whichChoice 2 choice {...shader 3 ...}
    ....

DEF switchscript Script {
    eventOut SFInt32 value_changed
    eventIn SFBool isActive
    field SFInt32 current 0
    url ["vrmlscript:
        function isActive(value) {
            if (value == true) {
                current = (current + 1)%3;
                value_changed = current;
            }
        }
    "
    ]
}
....

ROUTE touch.isActive TO switchscript.isActive
ROUTE switchscript.value_changed TO Multistyle.whichChoice

```

Example 4.12 multi-shader interface code fragment

3. Controllable specular lighting

Beside the above two controls, we also defined some scripts for controlling shader parameters through a slide sensor, such as controlling the phongN value which effects specular light of the Blinn-Phong shader on the objects surface This function will be useful for collaborative work. By highlight particular atoms, collaborators in different location can easily identified individual atoms, etc. We will talk more about this in Chapter 5.

The following code fragment shows how a script control parameter of phongN in Blinn-Phong shader is controlled with the slide sensor. First we pick the index [0] value

of the translation change in planeSensor ps and rearranged the value scope more fitting to phongN and assign the new value to phongN.

We used vertex and fragment shader of Phong shading as a test sample. The fragment shader fragment is shown as follows.

```
void main(in float4 position : TEXCOORD0, //in world space
in float3 normal : TEXCOORD1, //in world space
uniform float3 viewPosition, //in world space
uniform float3 Ka, uniform float3 Kd, uniform float3 Ks,
uniform float phongN,
uniform float3 ambientLight,
uniform float3 lightColor,
uniform float3 lightPosition, //in world space
out float4 oColor : COLOR)
{
    float3 P = position.xyz;
    float3 N = normalize(normal);
    float3 ambient = Ka * ambientLight; // ambient light contribution

    //compute the diffuse light contribution
    float3 L = normalize(lightPosition - P);
    float diffuseLight = max(dot(N, L), dot(N, L)*1.1);
    float3 diffuse = Kd * lightColor * diffuseLight;

    // compute the specular light contribution
    float3 V = normalize(viewPosition + P);
    float3 H = normalize(L + V);
    float specularLight = (pow(max(dot(N, H), -dot(N, H)), phongN));
    if(diffuseLight < 0)
    {
        diffuseLight = -diffuseLight;
        float3 specular = lightColor * specularLight;
        oColor.xyz = (ambient - diffuse*0.2 + abs(specular));
    }
    //specularLight = 0;
    else{
        float3 specular = lightColor * specularLight;
        oColor.xyz = (ambient +(diffuse) + specular);}

    oColor.w = 0.9;
}
```

Example 4.13 Fragment phong shader example for parameters control

Uniform variables and input variables are all coming from the VRML/X3D scenes. The interface of shader nodes allows defined variables to go though it and send value of defined variable out and in as shown in Example 4.13(a):

```
FragmentShader{
  exposedField MFString url []
  exposedField SFBool mustEvaluate
  exposedField MFString paramName []
  exposedField MFString paramType []}
```

Example 4.13(a) fragment shader interface with main program

```
fragmentShader DEF Hydrogen_Frag FragmentShader {

  exposedField SFColor Ka IS Ka
  exposedField SFColor Kd 0.8 0.6 0.4
  exposedField SFColor Ks 0.7 0.7 0.7
  exposedField SFColor ambientLight 0.2 0.2 0.2
  exposedField SFFloat phongN 100
  exposedField SFColor lightColor 1.0 1.0 1.0
  exposedField SFVec3f lightPosition 0 0 15
  exposedField SFFloat transparency IS transparency

  url [ "phong_shader/phong_frag.cg" ]}
```

Example 4.13(b) Example of a fragment phone shader declaration in main program

```
Transform {
  translation -2 0 0
  children [
    DEF slidersensor PlaneSensor {
      minPosition 0.0 0.0
      maxPosition 1.5 0.0
      offset 0 0 0
      autoOffset TRUE
    }
    DEF regler Transform {
      scale 3 6 3
      children Shape {
        appearance Appearance {
          material Material {
            diffuseColor .5 0 0
            specularColor .5 .4 .2
            shininess .4
          }
          geometry Sphere { radius .05 }
        }
      }
    }
    Transform {
      translation 0.56 0 0
      rotation 0 0 1 1.5708
      children Shape {
        appearance Appearance {
          material Material {
            diffuseColor .3 .3 .3
            specularColor .7 .7 .8
            shininess .2
          }
          geometry Cylinder { radius .1 height 2 }
        }
      }
    }
  ]}]}
```

Example 4.13(c) fragment shader interface with main program

Let's use variable "phongN" as an example to be manipulated using a slider in VRML/X3D. We give an MFString name as "phongN" and paramType as SFFTIME. As the interface for uniform phongN to output from main program and input to fragment shader as follows.

By looking at the code, we see the value of variable phongN is fixed. That means, if we want to control it in main program, we have to make it changeable. Let's design a slider controller first, create a cylinder as the slider controller and a scaled ball "regler" as the slider. A plane sensor is needed for tracking the controller's movement:

```
ROUTE slidersensor.translation_changed TO Hydrogen_Frag.phongN
```

Example 4.13(d) route function control phongN value with slider

However, since the data type of phongN and the translation type of the slider are not compatible, we need to write a fraction script to convert the value to the right data type and value range.

```
DEF fractionscript Script {
  eventIn SFVec3f set_translation
  eventOut SFFloat oneDFloat
  eventOut SFCOLOR threeDFloat

  url "vrlscript:
  function set_translation(value) {
    oneDFloat = ((value[0])*10000/100);
    threeDFloat [0]= value[0];
  }"}
}
```

Example 4.13(e) fraction functions for data field type conversion

And we correct the route as follows:

```
ROUTE slidesensor.translation_changed TO regler.set_translation
ROUTE slidesensor.translation_changed TO fractionscript.set_translation
ROUTE fractionscript.oneDFloat TO Hydrogen_Frag.phongN
```

Example 4.13(f) ROUTE control interface for specular light

Furthermore, if we want to use this specular light control as a prototype, meaning, if we would like to control all specular of the same atoms, a prototype needs to be defined. Create object with the defined proto.

```

PROTO My_P [

  exposedField SFFloat phongN 100
  exposedField SFFloat transparency 0.3
  exposedField SFCOLOR Ka 0.8 0.6 0.4

  ]{Shape {
    appearance ShaderAppearance {
      transparent TRUE
      vertexShader VertexShader {
        url [ "phong_shader/phong_vert.cg" ]
      }
      fragmentShader DEF Hydrogen_Frag FragmentShader {
        exposedField SFCOLOR Ka IS Ka
        field SFCOLOR Kd 0.8 0.6 0.4
        field SFCOLOR Ks 0.7 0.7 0.7
        field SFCOLOR ambientLight 0.2 0.2 0.2
        exposedField SFFloat phongN IS phongN
        field SFCOLOR lightColor 1.0 1.0 1.0
        field SFVec3f lightPosition 0 0 15
        exposedField SFFloat transparency IS
      }
      transparency
        url [ "phong_shader/phong_frag.cg" ]
    }
    geometry Sphere { radius .20 }
  }}

  Transform { translation 0 0 0 children [
    DEF my_P_1 My_P {# phongN 100} ]}

```

Example 4.13(g) A proto with specular light of shader control

Figure 4.25 shows the comparison of before and after the parameter had been change and shows the result of the effect by the control.

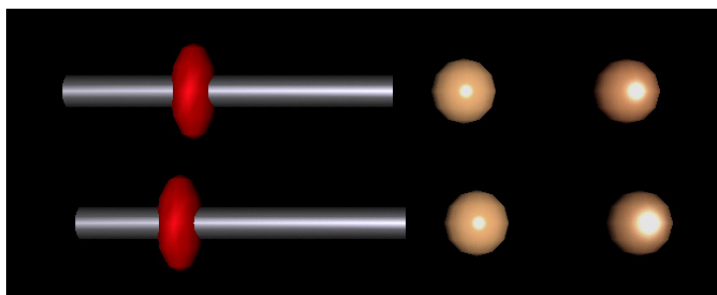


Figure 4 - 25 controlled specular light of shader on a atom surface

Same other examples for control bump level:

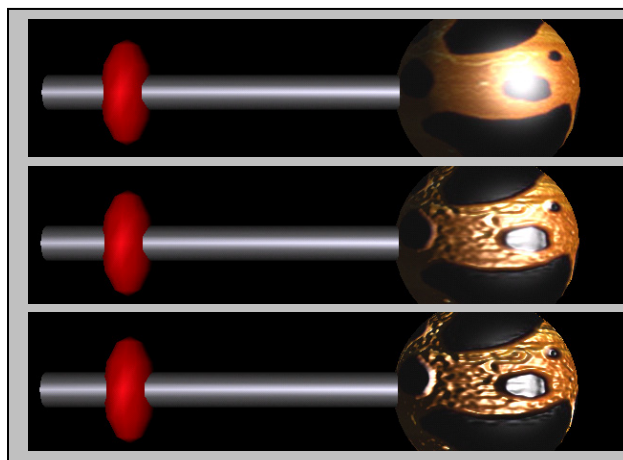


Figure 4 - 26 A controlled bump shader on a atom surface

4. Building molecule structure easily:

With the defined shaded atoms' prototypes, we can create new atoms with shader in the 3D structure. After added atoms in, we would like to move them around to a desired position. A plane sensor has been used to track the new position of the atom. We can also predefine some functions for some known structures or patterns. This will save time for building a structure. Some resulting images are shown in Figure 4.27.

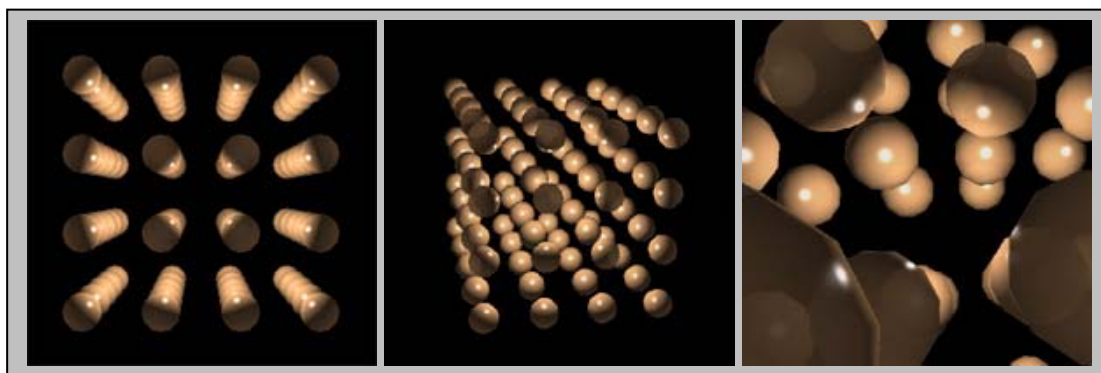


Figure 4 - 27 resulting image of adding new atoms in predefined functions

2.4 Future interface design for 3D structural biology

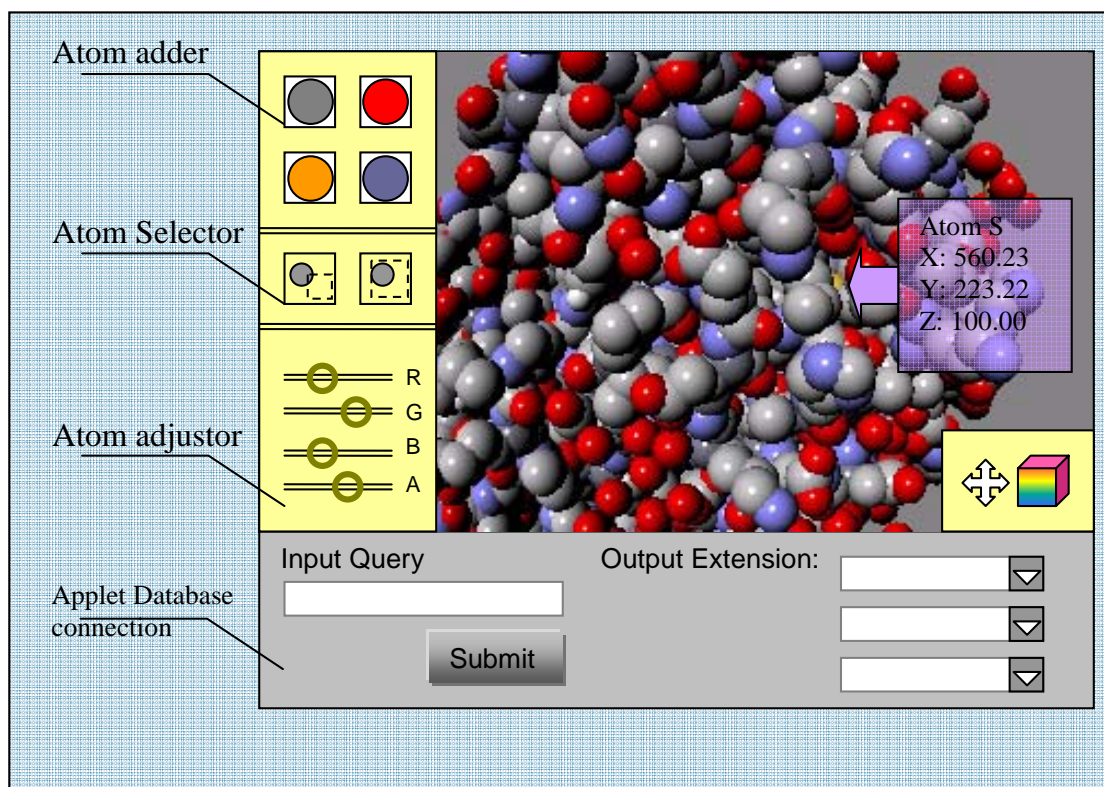


Figure 4 - 28 user interface design for the structural biology.

In section 4.2, we developed a set of basic functions which improved user interaction with 3D visualizations of Bio-molecule information. In this section, we extend the basic functions and gave a proposed user interactive interface design for future 3D structural biology research.

This interface design includes three parts. One is the data display area. Two is the interaction EAI with database. Last important part is manipulation controls, which includes selector, adjuster, etc. the information of molecule could be the query result of the on Java which connect to a database.

5 Multi-User shared Environment in X3D/VRML with Shader supported

5.1 Concepts of Multi-user shared environment and ASEC

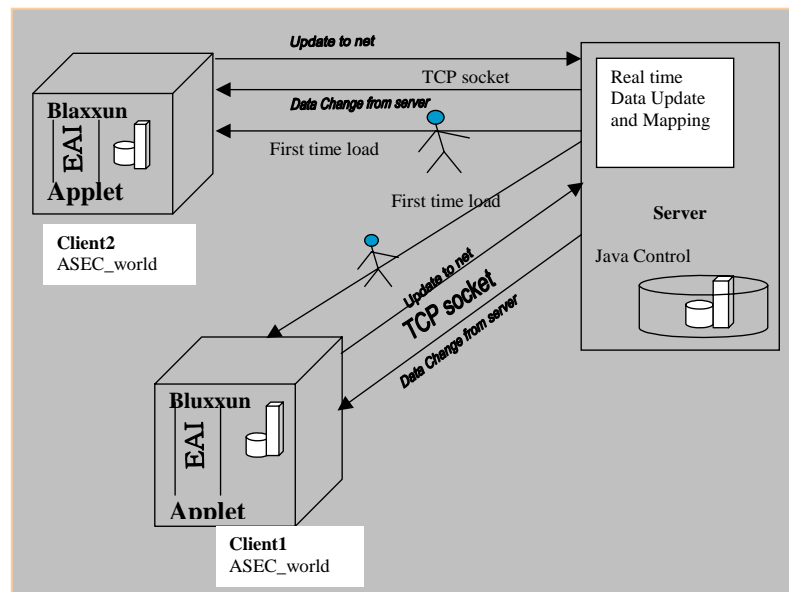


Figure 5 - 1 High-level structure of ASEC System

A multi-user shared environment (MUSE) was built in 2000 for the ACM SIGGRAPH Education Committee. It was a 3D multi-user real-time interactive shared environment. This environment included galleries and a tutorial class room, etc. The galleries collected all posters from 1993 to 2000 from the Education Committee SPACE competition. The 3D environment made users feel more like a real gallery. Users can click the posters on the wall to read more detail information in another pop-up window. Users also can jump among 1993 room to 2000 room. In the tutorial section, a classroom for a texture mapping technology is set up for shared between instructor and students. A

shared environment here means, when users getting into same room, they will see each other's avatar movement, they can chat with each other by typing in a message. This ASEC system was build on DeepMatrix multi-user shared environment. [DeepMatrix 2000] The architecture of system is shown as follows.

Figure 5.1 shows the high-level structure of how a multi-user shared environment works. The initial environment (VRML world) is transferred to the client only once during the initial load; after that it exists in the cache of the client browser, assuming no changes. The shared objects and Avatars are the only items transferred from the server-client afterward. Those shared objects are controlled by the Java (applet) and communicate with the VRML world using the eventIn / eventOut field of the Network nodes defined in the VRML file, pursued by VRML EAI (External Authoring Interface).

Figure 5.2 shows that a button clicked leading slides tutorial shared in a classroom in one of designed multi-user shared room. In this slide tutorial of texture mapping for SIGGRAPH97, when a page button is clicked in the teacher's (client#1) scene, the slides advance to next page, and the new page is displayed on each student's (client#2) screen.

•Interactive with slides:

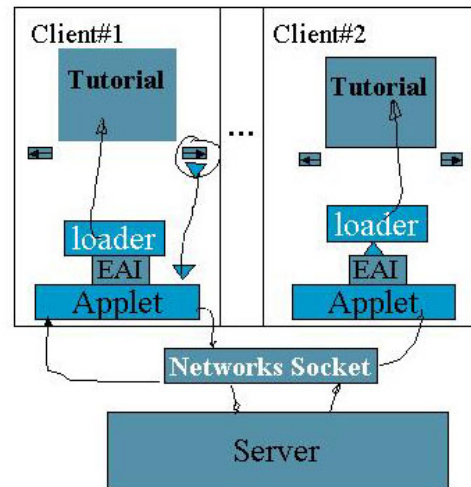


Figure 5 - 2 the slides tutorial updated information real-time sharing structure

A loader program continually checks for any changes and any updated information needed to be transferred. Once a click happens on the forward button, an eventOut of new URL address, containing the new slides, is sent to the client's local applet by the loader through EAI. The updated information is sent to the server through the network socket. After the server updates the data with new URL, the new URL address is broadcasted to all the clients. All clients (students) see the updated slides. Since only data that is transferred is the URL string, the data sharing process achieves a real-time update in all clients.

The original multiple user environments achieved a real-time interactive environment on the web with the following characteristics:

1. Modeling: describe objects with the lowest amount of polygons as possible

2. Viewing: removed all scene data from behind the viewer (back surface), clipping as well
3. Shading: Gouraud Shading instead of Phong Shading, which is very time consuming if it is computed on a single CPU.
4. Rendering: Scan-line Rendering instead of Ray-Tracing or Radiosity rendering
5. Extensible prototype defined makes it possible to have communicated with the external world.
6. Flexible JavaScript enriched the client's interactivity with the 3D scene and the data transformation is only the changed ones though the network.

You might notice that we didn't claim the output is realistic. Scenes do not look realistic because of VRML's poor shading and rendering algorithm. They don't have a very good image resolution. If there were shading and rendering algorithm like Phong-shading, Ray-Tracing or Radiosity the scene could approach to a very realistic image. Unfortunately, their computation time on the CPU keeps them away from real-time methods.

Using the newly discovered GPU technology, we applied the high image quality visualization into the X3D/VRML scene with real-time procedural shaders, which gives highly realistic, real-time image result without much increase in CPU computation time. This is because Shaders are running on the GPU. We talked about how to embed shaders into the single X3D/VRML scene. We designed a shader language converter for solving the shader hardware dependent problem. Those preparations gave the feasibility to share

shaders in a multi-user shared environment. In this section, we will talk about how to share these real-time procedure shaders in a multi-user shared environment. We first give a whole structure of how the original system works, and then we discuss how shaders are added and designed to be shared in that system. Then, we will go through some examples of adjusting parameters in shader to explain how the shader has been shared.

5.2 *Implantation and Improvement analysis of MUSE with shaders*

We first add shaders into the individual scene to improve the rendering quality of the multi-user shared environment. Then we shared the updated shader information to all users. We'll give two scenarios of control and share shaders in X3D/VRML scenes.

First example is motivated by collaboration work for bio-molecule modeling and manipulation. For example, when two researchers share a same data set of a protein molecule, one of them tries to point out which atom he is talking about. The easiest way is highlighting this particular atom. This is the same function as gaining readers attention by highlighting some keyword in an article. First of all each of the atoms have its own Phong shaders which resulted from chapter 4, which can be controlled by a slidersensor in section 4.2.2.3. We will not repeat code here. Again, the movement of slidersensor is sent to the predefined `Hydrogen_Frag` shader as its variable `phongN` is updated. (Show as follows).

```
ROUTE slidersensor.translation_changed TO Hydrogen_Frag.phongN
```

When the local researcher moves the slider, the particular atom will be highlighted. For sharing the highlighting information with researcher on the remote side,

a Network node needs to be defined with few variables. One variable is eventIn of datatype SFVec3f, which carries the updated 3D coordinate of the slider. The other one is an eventIn of data type SFBool, which works as a flag to the loader signaling there is information that has been updated. Accordingly, a pair of eventOuts needs to be defined as receiver for the updated information. Since the original system network node contains only Boolean variables, we have to add a SFVec3f variable to carry the 3D coordinate. Example 5.1 shows the code fragment for the prototype of a Network node which will carry the datatype as SFVec3f and the script for computing the SFVec3f data. Example 5.1(a) shows a net node of NetworkSFV3f defined as global variable, in charge of transferring data. Example 5.1(b) shows the new ROUTE for the updated scene information transferred by the network.

As we can see, the whole updating process only deals with sliders in both scenes. The advantage of this design is it minimized the transferring data. We'd like to mention here, the transferred data has to go through the SLC after it is received by destination user due to the possibility of different user works on different platform. Since the variable names are just interface with main program, we assume they are same in most case in different languages. The shader itself is not necessary transferred. This minimizes the server transferring load.

With network node help, the new updated data is sent to the remote site and the highlight object is shown on the remote screen. Sharing highlighted objects is one example of the multi-users system benefits collaborative work online.

```

PROTO NetworkSFVec3f [
  eventIn SFVec3f set_value
  eventOut SFVec3f value_changed
  eventIn SFVec3f value_fromnet
  eventOut SFVec3f value_tonet
  ... ..
  exposedField SFBool   pilotOnly TRUE
  ... ..
]
{ Script {
  eventIn SFVec3f InSc IS set_value
  eventOut SFVec3f OutSc IS value_changed
  eventIn SFVec3f netInSc IS value_fromnet
  eventOut SFVec3f netOutSc IS value_tonet
  ... ..
  url "javascript:
  function InSc(value) {
    netOutSc = value;
    if( local == true )
      OutSc = value;
  } function netInSc(value) {
    OutSc = value;
  }
  "
}
}

```

Example 5.1 A NetworkSFVec3f nodes for transferring the 3D movement of the slider

```

Group {
  children [
    DEF MATRIX_TRACKER ProximitySensor {
      size 100000 100000 100000
    },
    DEF net NetworkSFVec3f { ... ..},
  ]
}

```

Example 5.1(a) net node defined for transferring the 3D movement

```

#ROUTE slidersensor.translation_changed TO Hydrogen_Frag.phongN

ROUTE slidersensor.translation_changed TO net.set_value
ROUTE net.value_changed TO net.value_tonet
ROUTE net.value_fromnet TO net.set_value
ROUTE net.value_changed TO slidersensor.set_value
ROUTE slidersensor.translation_changed TO Hydrogen_Frag.phongN

```

Example 5.1(b) ROUTE of net node transferring the 3D movement of the slider

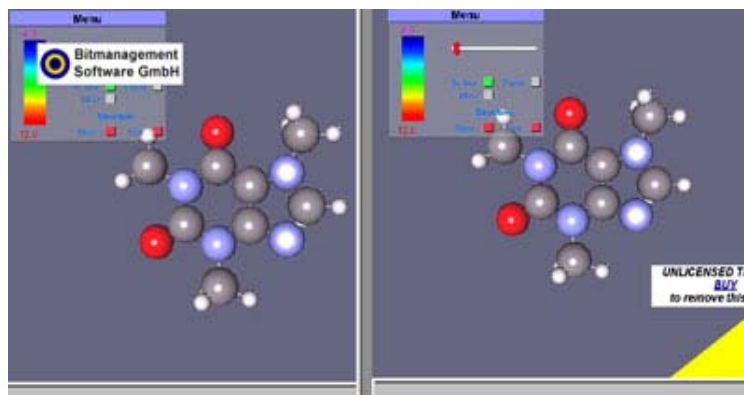


Figure 5 - 3 Shader sharing of Caffeine structure in multi-user environment

Figure 5.3 shows a multi-users shared environment for a simple caffeine structure. When user on the right screen move the slider to control the highlight of some particular atoms “N”, user on the left screen can see that some of the atoms “N” have been highlighted.

Another scenario we are giving next is a directly manipulation of a gallery image. Traditional paintings on canvas are two dimensional images; texturing canvas gives a 2D image more depth. We design this scenario based on the gallery poster section in ASEC world. Pick one poster on the wall of one scene. This painting has a lot of texture on it. We first create a high-field image of this painting. Then we use a bump-map shader to apply the high-field to the surface of the canvas. After that, we add a slider bar which is the controller for the bump level.

We are not going to show code details. Functions are similar to the last example except the bump data would be transferred. Figure 5.3 shows the architecture of shader sharing procedure.

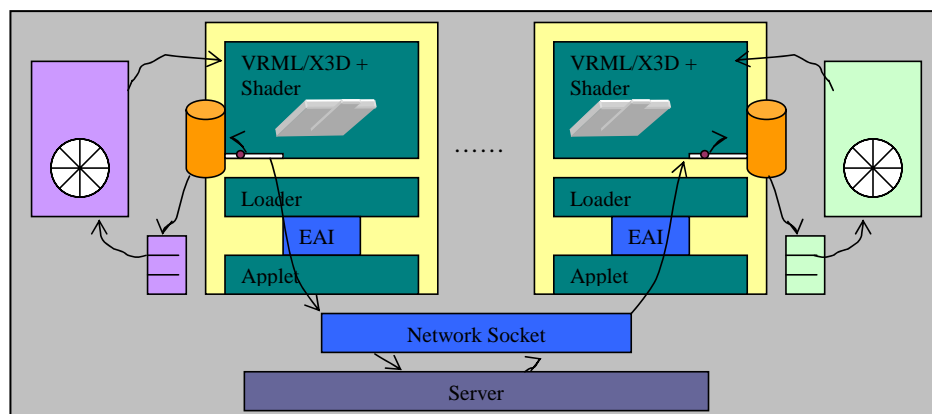


Figure 5 - 4 Shader sharing in multi-user environment

In Figure 5.3, when the slider is moved by user in the left window, two actions happen, the “value_changed” of the slider is sent to the network node and to the shadein X3D with customized shaders

Extending the two simple scenarios we gave in last section, a few potential applications can be implemented based on this shader-X3D/VRML multi-user shared environment.

- Bio-molecule structural application
- Online game
- E-real estate
- Online – education
- E-CAD production
- E-historical site Construction
- Etc.

We talked a lot about bio-molecule research in the last chapter. A powerful interface could be designed based on a more detailed user study. Shaders give more space

for game designers to realize their ideas. First of all, the LOQ with shader will reduce the number of polygon calculations and achieve a high quality image. Secondly, a realistic real-time interactive scene gives players a better visual experience. Third, complex shaders, for example deformation (which change the shape of an object), can give user a dynamic customizable appearance. For example the user can customize himself/ herself with a transparent shader or protect him/her with a shell. When enemies shoot at him, he can become invisible or protect himself inside the shell. These functions give another layer of interest to the game.

For the applications in real estate, a customizable lighting will be attractive to the users. VRML97 does not have a very good method for dealing with lighting. Lighting is very important for presenting an interior design. As the time an agent shows customers a house, they always turn the lights on to show the structure of each room. A shadow shader with diffuse color changeable light can not only show off the house design, it also gives the customer an idea of what would be their best choice.

As talking to E-CAD design, a lot of industry companies are international based. For example, the Delph Product Corp. sends their product assembling line to Shanghai, China. Afterward, they spend a lot of time and money traveling between its design center Rochester, NY and Shanghai. The cost for communication became a big budget each year. A lot of international based companies face similar problem. A multi-user environment for collaborating is desired. Our multi-user shader X3D scene can allow designers to share their work with their design partners, i.e. customized color, material

with adjusting shader parameters and preview the updated result instantly online. It also can be used as a remote training platform.

Shader can add a lot of surface effect. For example, people might not like to reconstruct a historic site the way it is. However, they would like to show what it looked like when it was famous or when special events happened. A customizable shader can restore the beauty of the building as it was in the past. An interactive function like a touch sensor can bring the visitors to a museum to its old time.

There are many more applications that can take advantage of this shader-shared real-time realistic environment.

6 Conclusion

The contents of this dissertation can be summarized in three parts as follows.

A Shader Language Converter was designed (and partially developed) for solving the platform dependant problem that exists with current shading languages paving the road for applications to share Shader information via the Internet. This process was further developed and demonstrated when we built a framework for embedding shaders into a 3D application with X3D. Other enhancements were demonstrated that would further improve the process of implementing Shader sharing and dissemination via the Internet.

An application for molecule structure visualization utilizing shaders was developed. It included converting PDB files, the output file format of the Arp/Warp program, to CML, Chemical Mark-up Language. A conversion from CML to X3D was then performed and during this conversion process predefined shaders for the elements contained in protein molecules were added. The next step was converting the X3D file to VRML, which allowed the molecule to be visualized in a 3D scene. Some basic functionality was added to the user interface that allows manipulation of the X3D/VRML scene. These functions included; slider controlled Shader parameters such as, specular lighting, surface color, and surface bump level. Also functions for multiple surface styles and an interface for adding atoms into the scene with predefined functions. Another function that was added was LOQ, level of quality, which reduces both shaders and surface detail in predefined levels related to the distance the molecule structure is being

viewed in the 3D scene. The usage of applying Phong shader to the surface of the basic atoms in molecular structures is a primary introduction of using shader in more complex way. Some future work of more complex shaders can be used for taking the advantage of the GPUs parallelized computation capability. For example, an electrostatic or electron density surface can be dynamically computed and displayed.

We also discussed methods by which the previous developments would best be utilized for collaboration via the Internet with the implementation of multi-user shared real-time interactive environment. A network node carries the updated Shader information send it to the server and the server would broadcast the updated information to all clients on the network who are sharing the scene. The types of applications that would benefit from this Shader shared multi-user environments are Bio-molecule structural application, Online game, E-real estate, Online – education, E-CAD production, E-historical site Construction etc

Reference:

3DLab: <http://www.3dlabs.com/> (2005)

Ames, A.L. Nadeau, D.R., and Moreland, J. L. The VRML 2.0 sourcebook (2nd ed.), John Wiley & Sons, Inc., New York, NY, 1997

Apodaca A.A. and Gritz L., *Advanced RenderMan: Creating CGI for Motion Pictures*, ed, Morgan-Kauffman, (1999)

ATI: www.ati.com (2005)

Badger, J.: An evaluation of automated model-building procedures for protein crystallography. *Acta Crystallographica*. International Union of Crystallography pp. 823-827. (2003)

Bernstein, H. J. and Sayle, R: RasMol Molecular Graphics Visualization Tool. (2000): www.openrasmol.com

Bitmanagement (2005): www.bitmanagement.de

Blinn, J. F. : a Generalization of Algebraic Surface Drawing. *ACM Transactions on Graphics*, 1:3, pp. 235-256. (1982)

Blinn, J. F.: Models of Light Reflection for Computer Synthesized Pictures. *Computer Graphics*, Vol. 11, No. 2, pp. 192-198, July 1977 (*Proceedings of SIGGRAPH 77*).

Buck, L., Foley, T., Horn D., Sugerma, J., Fatahalian, K., Houston, M., Hanrahan, P., Brook for GPUs: stream computing on graphics hardware Full text *ACM Transactions on Graphics* Volume 23 , Issue 3 Special Issue: Proceedings of the 2004 SIGGRAPH Conference (August 2004)

Casher O., Leach C., Page, C. S. and Rzepa, H. S.. Virtual Reality Modelling Language (VRML) in Chemistry. *Chemistry in Britain*, 34, 26 (1998)

Chemical Markup Language (2005): www.xml-cml.org

ChemVis Group on Chemistry Visualization. (2005)

Chem-Vis: www2.chemie.uni-erlangen.de/projects/ChemVis/index.html

Cook R. L.: Shade trees. In Proceedings of the 11th annual conference on Computer graphics and

- interactive techniques, ACM Press, pp. 223--231. (1984)
- Cook, R. L., L. Carpenter and E. Catmull, "The Reyes Image Rendering Architecture", *SIGGRAPH 87*, pp. 95-102.
- De Carvalho G. N. M.: High-level procedural shading VRML/X3D, *Proceedings of the SIGGRAPH 2003 conference on Web graphics: in conjunction with the 30th annual conference on Computer graphics and interactive techniques*, San Diego, California (2003)
- De Carvalho, G. N. M., and Gill, T., Parisi, T.: X3D Programmable Shaders *Proceedings of the ninth international conference on 3D Web technology 2004*: pp. 99 - 108 (2004)
- DeepMatrix: <http://www.geometrek.com/> (2000)
- Engel K. and Ertl. T. Texture-based Volume Visualization for Multiple Users on the World Wide Web. In Gervautz, M. and Hildebrand, A. and Schmalstieg, D., editor, *Virtual Environment '99*, pages 115-124. Eurographics, Springer, 1999.
- Fernando, R., and Kilgard, M. J.: *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley (2003)
- Flux: www.mediamachines.com (2005)
- Google hits for various web3D technologies: (2004): www.macWeb3D.org/ewiki/index.php?id=most%20popular%20Web3D%20format
- Gouraud, H.: Continuous Shading of Curved Surfaces. *IEEE Transactions on Computers* Vol. C-20, No. 6 pp. 623-629, June 1971.
- Hanrahan, P. and J. Lawson, "A Language for Shading and Lighting Calculations", *SIGGRAPH 90*, pp. 289-298.
- HIV-Animation *HIV life-cycle* , available at www.hopkins-aids.edu/hiv_lifecycle/hivcycle_txt.html (2005)
- Joy, B., Steele, G., Gosling, J., and Bracha, G. 2000.*Java(TM) Language Specification*, 2nd ed. Addison-Wesley.

- Kernighan, B. W, Ritchie D. M. 1988. *The C Programming Language*. Prentice Hall.
- Krieger, J. H.: Doing Chemistry in a virtual world: VRML, a new web technology, holds promise for chemistry in three dimensions *American Chemical Society* (1996)
- Krone Oliver thesis: available at : <http://www-lehre.inf.uos.de/~okrone/DIP/Diplomarbeit.html> (2003)
- Lindholm, E., Kilgard, M. J., and Moreton, H.: A User-Programmable Vertex Engine. *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. pp 149-158. (2001)
- Liu, F., Owen, G. S. Zhu, Y. Universal Converter for platform independent shader in X3D *ACM SIGGRAPH 2004 Web Graphics* Los Angeles Aug. 2004
- Liu, P., Wang, Y. F., Ewis, H. E., Abdelal, A. T., Lu, C. D., Harrison, R. W., and Weber, I. T.: Covalent Reaction Intermediate Revealed in Crystal Structure of the Geobacillus Stearothermophilus Carboxylesterase Est30. *Journal of Molecular Biology*. pp.342-551 (2004)
- Marc Olano , Anselmo Lastra, A shading language on graphics hardware: the pixelflow shading system, *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, p.159-168, July 1998
- Mark, W.R, Glanville, R. S., Akeley, K., and M.J. Kilgard: Cg: a system for programming graphics hardware in a C-like language. *ACM Transactions on Graphics* 22(3), pp. 896-907. ACM Press, July (2003)
- Molda : <http://www.molda.org/> (2005)
- MOLDEN: <http://www.caos.kun.nl/~schaft/molden/molden.html> (2004)
- MolScript : <http://www.avatar.se/molscript/> (2005)
- Ngan, A., Durand F., Matusik, W.: Experimental Validation of Analytical BRDF Models. *Sketch of SIGGRAPH2004* July (2004)
- NIH: The Structure of life : http://www.nigms.nih.gov/news/science_ed/structlife/ (2005)
- NVIDIA : www.nvidia.com (2005)
- Open Inventor: SGI: <http://oss.sgi.com/projects/inventor/> (2005)
- OpenGL www.opengl.org (2005)
- Parisi, T.,: Flux: lightweight, standards-based Web graphics in XML, *Proceedings of the SIGGRAPH 2003*

- conference on Web graphics: in conjunction with the 30th annual conference on Computer graphics and interactive techniques*. San Diego, California (2003)
- PDB: Protein Data Bank Resource. (2005): www.rcsb.org/pdb/
- PDB2VRML-c: <http://www.geocities.com/gnubioq/pdb2vrml/> (2005)
- PDB2VRML-chem: <http://www.chm.davidson.edu/VRML/pdb2vrml.html> (2005)
- PDB2VRML-Perl: <http://molmovdb.mbb.yale.edu/MolMovDB/vrml/> (2005)
- PDB2VRML-vrml : http://www.pc.chemie.tu-darmstadt.de/research/vrml/pdb2vrml_right.shtml
- Peeper, C. and Mitchell, J. L.: *Introduction to the DirectX® 9 High Level Shading Language* (2002):
[www.ati.com/ developer/ ShaderX2_ IntroductionToHLSL.pdf](http://www.ati.com/developer/ShaderX2_IntroductionToHLSL.pdf)
- Perlin, K., 1985, An Image Synthesizer. *Computer Graphics* 1985: 19(3), pp. 287- 296. 43
- Perrakis A., Morris R., and Lamzin V.S.: Automated protein model building combined with iterative structure refinement. *Nature Struct. Biol.*, pp 458-463 (1999)
- Phong, B.T.: Illumination for Computer Generated Pictures, *Communications of the ACM*, Vol. 18, No. 6, pp. 311-317, June 1975.
- Pixar www.pixar.com (2005)
- Polys, N.: Stylesheet Transformations for Interactive Visualization: Towards Web3D Chemistry Curricula. *Proceeding of the eighth international conference on 3D Web technology pp. 85 – 91* (2003):
www.3dez.net/X3D/CML/
- Proudfoot, K., Mark, W. R., Tzvetkov, S., AND Hanrahan, P. 2001. *A real-time procedural shading system for programmable graphics hardware*. In SIGGRAPH 2001.
- Purcell, T. J., Buck, I., Mark, W. R., and Hanrahan, P.: Ray Tracing on Programmable Graphics Hardware *ACM Transactions on Graphics* 21: 3, pp 703-712 (2002).
- RasMal : <http://www.umass.edu/microbio/rasmol/> (2005)
- Raster3D: www.bmsc.washington.edu/raster3d/raster3d.html
- Rost. R. J. And Licea-Kane B.: *The OpenGL(R) Shading Language*. Addison-Wesley (2004)
- Roucoux, K., *Heaven and Earth: Unseen by the Naked Eye* Press by Phaidon (2002)

- Schroder, H. K., Willassen, N. P., and Smalas, A. O.: Structure of a non-psychrophilic trypsin from a cold-adapted fish species. *Acta Crystallogr D Biol Crystallogr* 54 pp. 780 (1998)
- SGI : www.SGI.org (2005)
- ShaderGen : <http://developer.3dlabs.com/downloads/shadergen/index.htm> (2005)
- ShaderTechs : www.cgshaders.org (2005)
- Spartan: <http://www.wavefun.com/> (2005)
- Steve Upstill. *The Renderman Companion*. Addison Wesley, Reading, MA, 1989.
- Stowell, D., *The Molecules of HIV* available at www.mcl.d.co.uk/hiv/; <http://www.mcl.d.co.uk/hiv/?q=HIV%20life-cycle> 2005
- Stroustrup, B. 2000. *The C++ Programming Language*, 3rd ed. Addison-Wesley.
- Tie, Y., Boross, P. I., Wang, Y. F., Gaddis, L., Hussain, A. K., Leshchenko, S., Ghosh, A. K., Louis, J. M., Harrison, R. W., and Weber, I. T.: High Resolution Crystal Structures of HIV-1 Protease with a Potent Non-Peptide Inhibitor (Uic-94017) Active Against Multi-Drug-Resistant Clinical Strains. *Journal of Molecular Biology*. pp.338-341 (2004)
- W3C: World Wide Web Consortium: www.w3c.com (2005)
- Web3D Consortium. (2005): www.Web3D.org
- Web3D-Shader-Group (2005): www.web3d.org/x3d/workgroups/x3d-shaders.html
- Web3D-ToolKits: X3D Toolkits CD SIGGRAPH2003. (2003)
- Web3D-Tools (2005): www.web3d.org/applications/tools/viewers_and_browsers/
- Wikipedia (2005): www.en.wikipedia.org/wiki/Graphics_processing_unit
- www.ccc.uni-erlangen.de/
- wwwvis.informatik.uni-stuttgart.de/
- Xeena: IBM: <http://www.alphaworks.ibm.com/tech/xeena>; <http://www.garshol.priv.no/download/xmltools/prod/Xeena.html> (2005)
- Xj3D: CVS available: <http://www.xj3d.org/cvs.html> (2005)
- Zara, S.: PDB2CML Implementation (1999): www.xml-cml.org/software/pdb2cml.html
- Zou, Y.: Built VRML model of 3D molecular structures (1999): www.molvmltripod.com

Appendix A. shaders examples in Cg and GLSL

A1a – wood_vert.cg

```
// VertexShader
const float3 LightPosition =float3( 0, 0, 4.0 );
const float Scale =2.0 ;
void main(
    out float lightIntensity ,
    out float3 Position ,
    in uniform float4x4 gl_ModelViewMatrix ,
    in float4 gl_Vertex :POSITION ,
    in float4 gl_Normal :NORMAL ,
    in uniform float4x4 gl_NormalMatrix ,
    in uniform float4x4 gl_ModelViewProjectionMatrix ,
    out float4 gl_Position :POSITION ){
    float4 pos =mul( gl_ModelViewMatrix, gl_Vertex );
    Position =(float3)( gl_Vertex )*Scale ;
    float3 tnorm =normalize( mul( gl_NormalMatrix, gl_Normal ).xyz );
    lightIntensity =max( dot( normalize( LightPosition -(float3)( pos ) ), tnorm
    ), 0.0 )*9.5 ;
    gl_Position =mul( gl_ModelViewProjectionMatrix, gl_Vertex );}
```

A1b – wood_frag.cg

```
// FragmentShader
const float GrainSizeRecip = 1.0;
const float3 DarkColor = float3(0.6, 0.3, 0.1);
const float3 colorSpread = float3(0.15, 0.075 , 0.0);

void main (
    in float lightIntensity: TEXCOORD0,
    in float3 Position: TEXCOORD1,
    out float4 gl_FragColor:COLOR0)

{
    float3 location = Position;
    float3 floorvec = float3(floor(Position.x * 10.0),
        0.0, floor((Position).z * 10.0));
    float3 noise = Position * 10.0 - floorvec - 0.5;
    noise *= noise;
    location += noise * 0.12;

    float dist = location.x * location.x + location.z * location.z;
    float grain = dist * GrainSizeRecip;

    float brightness = frac(grain);
    if (brightness > 0.5)
        brightness = (1.0 - brightness);
    float3 color = DarkColor + brightness * (colorSpread);

    brightness = frac(grain*7.0);
    if (brightness > 0.5)
        brightness = 1.0 - brightness;
    color -= brightness * colorSpread;
    color = clamp(color * lightIntensity, 0.0, 1.0);
    gl_FragColor = float4(color, 1.0);}
```

A1aa – wood.vert

```
// VertexShader in GLSL

const vec3 LightPosition =vec3( 0, 0, 4.0 );
const float Scale =2.0 ;

varying float lightIntensity ;
varying vec3 Position ;
void main(void)
{
    vec4 pos =gl_ModelViewMatrix * gl_Vertex;
    Position =vec3( gl_Vertex )*Scale ;
    vec3 tnorm =normalize( gl_NormalMatrix * gl_Normal.xyz );
    lightIntensity =max( dot( normalize( LightPosition -vec3( pos ) ),
    tnorm ), 0.0 )*9.5 ;
    gl_Position =gl_ModelViewProjectionMatrix * gl_Vertex;
}

```

A1bb – wood.Frag

```
// FragmentShader in GLSL

const float GrainSizeRecip = 1.0;
const vec3 DarkColor = vec3(0.6,0.3,0.1);
const vec3 colorSpread = vec3(0.15,0.075,0.0);

varying float lightIntensity;
varying vec3 Position;

void main (void)
{
    vec3 location = Position;
    vec3 floorvec = vec3(floor(Position.x * 10.0), 0.0, floor(Position.z * 10.0));
    vec3 noise = Position * 10.0 - floorvec - 0.5;

    noise *= noise;
    location -= noise * 0.12;
    float dist = location.x * location.x + location.z * location.z;
    float grain = dist * GrainSizeRecip;
    float brightness = fract(grain);
    if (brightness > 0.5)
        brightness = (1.0 - brightness);
    vec3 color = DarkColor + brightness * (colorSpread);

    brightness = fract(grain*7.0);
    if (brightness > 0.5)
        brightness = 1.0 - brightness;
    color -= brightness * colorSpread;
    color = clamp(color * lightIntensity, 0.0, 1.0);
    gl_FragColor = vec4(color, 1.0);
}

```

A1aaa – wood.vert.Xml

```

<?xml version="1.0"?>
<VertexShader>
  <func>
    <returnType>void</returnType>
    <funcName>main</funcName>
  </func>
  <parameters>
    <declaration>
      <modifier>in</modifier>
      <qualifier>
        </qualifier>
      <type>float4</type>
      <para>position</para>
      <semantic>TEXCOORD0</semantic>
    </declaration>
    <declaration>
      <modifier>in</modifier>
      <qualifier>
        </qualifier>
      <type>float3</type>
      <para>normal</para>
      <semantic>TEXCOORD1</semantic>
    </declaration>
    <declaration>
      <modifier>
        </modifier>
      <qualifier>uniform</qualifier>
      <type>float3</type>
      <para>viewPosition</para>
    </declaration>
    <declaration>
      <modifier>
        </modifier>
      <qualifier>uniform</qualifier>
      <type>float3</type>
      <para>baseColor</para>
    </declaration>
    <declaration>
      <modifier>
        </modifier>
      <qualifier>uniform</qualifier>
      <type>float3</type>
      <para>lightPosition</para>
    </declaration>
    <declaration>
      <modifier>out</modifier>
      <qualifier>
        </qualifier>
      <type>float4</type>
      <para>oColor</para>
      <semantic>COLOR</semantic>
    </declaration>
  </parameters>
  <body>
    <declaration>
      <modifier>
        </modifier>
      <qualifier>
        </qualifier>
      <type>float3</type>
      <op_assign>
        <para>lightColor</para>
        <func>
          <para>float3</para>
          <parameters>
            <para>1.5f</para>
            <para>1.5f</para>
            <para>1.5f</para>
          </parameters>
        </func>
      </op_assign>
    </declaration>
    <declaration>
      <modifier>
        </modifier>
      <qualifier>
        </qualifier>
      <type>float3</type>
      <op_assign>
        <para>P</para>
        <op_reference>
          <para>position</para>
          <para>xyz</para>
        </op_reference>
      </op_assign>
    </declaration>
    <declaration>
      <modifier>
        </modifier>
      <qualifier>
        </qualifier>
      <type>float3</type>
      <op_assign>
        <para>N</para>
        <func>
          <para>normalize</para>
          <parameters>
            <para>normal</para>
          </parameters>
        </func>
      </op_assign>
    </declaration>
    <op_reference>
      <para>-lightPosition</para>
      <para>z</para>
    </op_reference>
    <declaration>
      <modifier>
        </modifier>
      <qualifier>
        </qualifier>
      <type>float3</type>
      <op_assign>
        <para>L</para>
        <func>
          <para>normalize</para>
          <parameters>
            <op_plus>
              <para>lightPosition</para>
              <para>P</para>
            </op_plus>
          </parameters>
        </func>
      </op_assign>
    </declaration>
  </body>
</VertexShader>

```

```

<modifier>
  </modifier>
  <qualifier>
  </qualifier>
  <type>float</type>
  <op_assign>
    <para>diffuseLight</para>
    <func>
      <para>max</para>
      <parameters>
        <func>
          <para>dot</para>
          <parameters>
            <para>N</para>
            <para>L</para>
          </parameters>
        </func>
        <para>0</para>
      </parameters>
    </func>
  </op_assign>
</declaration>
<declaration>
  <modifier>
  </modifier>
  <qualifier>
  </qualifier>
  <type>float3</type>
  <op_assign>
    <para>diffuse</para>
    <op_multi>
      <para>baseColor</para>
      <op_multi>
        <para>lightColor</para>
        <para>diffuseLight</para>
      </op_multi>
    </op_multi>
  </op_assign>
</declaration>
<declaration>
  <modifier>
  </modifier>
  <qualifier>
  </qualifier>
  <type>float3</type>
  <op_assign>
    <para>V</para>
    <func>
      <para>normalize</para>
      <parameters>
        <op_multi>
          <para>viewPosition</para>
          <para>P</para>
        </op_multi>
      </parameters>
    </func>
  </op_assign>
</declaration>
<declaration>
  <modifier>
  </modifier>
  <qualifier>
  </qualifier>
  <type>float3</type>
  <op_assign>
    <para>H</para>
    <func>
      <para>normalize</para>
      <parameters>
        <op_plus>
          <para>L</para>
          <para>V</para>
        </op_plus>
      </parameters>
    </func>
  </op_assign>
</declaration>
<declaration>
  <modifier>
  </modifier>
  <qualifier>
  </qualifier>
  <type>float</type>
  <op_assign>
    <para>specularLight</para>
    <func>
      <para>pow</para>
      <parameters>
        <func>
          <para>max</para>
          <parameters>
            <func>
              <para>dot</para>
              <parameters>
                <para>N</para>
                <para>H</para>
              </parameters>
            </func>
            <para>-dot</para>
            <parameters>
              <para>N</para>
              <para>H</para>
            </parameters>
          </func>
          <para>256</para>
        </parameters>
      </func>
    </op_assign>
  </declaration>
  <if>
    <condition>
      <op_rela_not_greater>
        <para>diffuseLight</para>
        <para>0</para>
      </op_rela_not_greater>
    </condition>
    <body>
      <op_assign>
        <para>-specularLight</para>
        </op_assign>
      </body>
    </if>
  </op_assign>
</declaration>
<para>specularLight</para>

```

```

</if>
<declaration>
  <modifier>
  </modifier>
  <qualifier>
  </qualifier>
  <type>float3</type>
  <op_assign>
    <para>specular</para>
    <op_multi>
      <para>lightColor</para>
      <para>specularLight</para>
    </op_multi>
  </op_assign>
</declaration>
<op_div>
  <op_plus>
    <para>diffuse</para>
    <op_multi>
      <para>.3</para>
      <para>specular</para>
    </op_multi>
  </op_plus>
  <para>1.5</para>
</op_div>
<para>0.5</para>
</body>
</func>
</VertexShader>

```

A1c – brick_vert.cg

```

// VertexShader IN Cg for brick
const float3 LightPosition =float3( 0.0, 0.0, 4.0 );
const float specularContribution =0.3 ;
const float diffuseContribution =0.4 ;
void main(
  out float LightIntensity ,
  out float2 MCposition ,
  in uniform float4x4 gl_ModelViewMatrix ,
  in float4 gl_Vertex :POSITION ,
  in float4 gl_Normal :NORMAL ,
  in uniform float4x4 gl_NormalMatrix ,
  in uniform float4x4 gl_ModelViewProjectionMatrix ,
  out float4 gl_Position :POSITION
)
{
  float4 ecPosition =mul( gl_ModelViewMatrix, gl_Vertex );
  float3 tnorm =normalize( mul( gl_NormalMatrix, gl_Normal ).xyz );
  float3 lightVec =normalize( LightPosition -(float3)( ecPosition ) );
  float3 reflectVec =reflect( -lightVec, tnorm );
  float3 viewVec =normalize( (float3)( -ecPosition ) );
  float spec =max( dot( reflectVec, viewVec ), 0.0 );
  spec =pow( spec, 16.0 );
  LightIntensity =diffuseContribution *max( dot( lightVec, tnorm ), 0.0
)+specularContribution *spec ;
  MCposition =(float2)( gl_Vertex );
  gl_Position =mul( gl_ModelViewProjectionMatrix, gl_Vertex );
}

```

A1d – brick_frag.cg

```

const float3 BrickColor = float3(1.0, 0.3, 0.2);
const float3 MortarColor = float3(0.85, 0.86, 0.84);
const float ColumnWidth = 0.30;
const float RowHeight = 0.15;
const float Bwf = 0.94;
const float Bhf = 0.90;
float4 main(
    float2 MCposition : TEXCOORD1,
    float LightIntensity : TEXCOORD0

):COLOR0
{
    float3 color;
    float ss, tt, w, h;

    ss = MCposition.x / ColumnWidth;
    tt = MCposition.y / RowHeight;

    if (frac(tt * 0.5) > 0.5) // frac = fract( ) in GLSL
        ss += 0.5;

    ss = frac(ss); // fract( ) is fract( in open GLSL)
    tt = frac(tt);

    w = step(ss, Bwf);
    h = step(tt, Bhf);
    color = (MortarColor*(1.0 - w * h) + BrickColor*(w * h)) * LightIntensity;

    float4 gl_FragColor = float4 (color, 1.0);
    return gl_FragColor;
}

```

A1cc – brick.vert

```

// Vertex shader in GLSL for brick

uniform vec3 LightPosition=vec3(0.0, 0.0, 4.0);
const float specularContribution = 0.3;
const float diffuseContribution = 0.7;//0.7 = 1.0 - specularContribution

varying float LightIntensity;
varying vec2 MCposition;

void main(void)
{
    vec4 ecPosition = gl_ModelViewMatrix * gl_Vertex;
    vec3 tnorm = normalize(gl_NormalMatrix * gl_Normal);
    vec3 lightVec = normalize(LightPosition - vec3 (ecPosition));
    vec3 reflectVec = reflect(-lightVec, tnorm);
    vec3 viewVec = normalize(vec3 (-ecPosition));
    float spec = max(dot(reflectVec, viewVec), 0.0);
    spec = pow(spec, 16.0);
    LightIntensity = diffuseContribution * max(dot(lightVec, tnorm), 0.0) +
        specularContribution * spec;

    MCposition = vec2 (gl_Vertex);
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

```

A1dd – brick.frag

```

const vec3  BrickColor = vec3(1.0, 0.3, 0.2);
const vec3  MortarColor = vec3(0.85, 0.86, 0.84);
const float ColumnWidth = 0.30;
const float RowHeight = 0.15;
const float Bwf = 0.94;
const float Bhf = 0.90;

varying vec2  MCposition;
varying float LightIntensity;

void main(void)
{
    vec3 color;
    float ss, tt, w, h;

    ss = MCposition.x / ColumnWidth;
    tt = MCposition.y / RowHeight;

    if (fract(tt * 0.5) > 0.5)
        ss += 0.5;
    ss = fract(ss);
    tt = fract(tt);

    w = step(ss, Bwf);
    h = step(tt, Bhf);

    color = mix(MortarColor, BrickColor, w * h) * LightIntensity;
    gl_FragColor = vec4 (color, 1.0);
}

```

A1e – Carbon atom-shader in Cg

```

// Carbon atom vertex shader in Cg: atom_vert.cg

void main(in float4 position : POSITION, //in object space
          in float4 position1:POSITION,
          in float4 normal   : NORMAL,   //in object space

          //mandatory parameters
          uniform float4x4 modelViewProjection,
          uniform float4x4 model,
          uniform float4x4 modelIT,
          uniform float3 viewPosition,

          //output parameters
          out float4 oPosition : POSITION, //
          out float3 oObjectPos : TEXCOORD0, //
          out float3 oNormal   : TEXCOORD1 //
)
{
    float4 nposition = float4(position.x,position.y,position.z,position.w);
    oPosition = mul(modelViewProjection, nposition);

    //transform the vertex position and normal into World Space:
    oObjectPos = mul(model, position).xyz;
    oNormal = mul(modelIT, normal).xyz;
}

```

A1f –Carbon atom-shader in Cg

```

// Carbon atom fragment shader in Cg: atom_frag.cg

void main(in float4 position  : TEXCOORD0, //in world space
          in float3 normal    : TEXCOORD1, //in world space
          uniform float3 viewPosition, //mandatory in world space
          uniform float3 baseColor,
          uniform float3 lightPosition, //defined in world space
          out float4 oColor : COLOR
)
{
    float3 lightColor = float3(1.5f, 1.5f, 1.5f);

    float3 P = position.xyz;
    float3 N = normalize(normal);

    //get lightPosition into space
    lightPosition.z = -lightPosition.z;
    //compute the diffuseColor value, assume lightColor is white
    float3 L = normalize(lightPosition + P);
    float diffuseLight = max(dot(N, L), 0);
    float3 diffuse = baseColor * lightColor * diffuseLight;

    float3 V = normalize(viewPosition * P);
    float3 H = normalize(L + V);
    float specularLight = pow(max(dot(N, H), -dot(N, H)), 256);
    if(diffuseLight <= 0)
        specularLight = -specularLight;
    float3 specular = lightColor * specularLight;

    oColor.xyz = (diffuse+ .3*specular)/1.5;
    oColor.w = 0.5;
}

```

A1ee –Carbon atom-shader in GLSL

```

// Carbon atom vertex shader in GLSL: atom.vert
// VertexShader

uniform mat4 modelViewProjection ;
uniform mat4 model ;
uniform mat4 modelIT ;
uniform vec3 viewPosition ;
void main(void)
{
    vec4 nposition = vec4( gl_vector .x ,gl_vector .y ,gl_vector .z ,gl_vector .w );
    gl_vector = modelViewProjection * nposition;
    gl_MultiTexCoord0.xyz = model * position.xyz ;
    gl_MultiTexCoord1.xyz = modelIT * normal.xyz ;
}

```

A1ff –Carbon atom-shader in GLSL

```

// Carbon atom fragment shader in GLSL: atom.frag
// FragmentShader

uniform vec3 viewPosition ;
uniform vec3 baseColor ;
uniform vec3 lightPosition ;

void main(void)
{
    vec3 lightColor =vec3( 1.5f, 1.5f, 1.5f );
    vec3 P =gl_MultiTexCoord0 .xyz ;
    vec3 N =normalize( gl_MultiTexCoord1.xyz );
    lightPosition.z -=lightPosition .z ;
    vec3 L =normalize( lightPosition +P );
    float diffuseLight =max( dot( N, L ), 0 );
    vec3 diffuse =baseColor*lightColor *diffuseLight ;
    vec3 V =normalize( viewPosition *P );
    vec3 H =normalize( L +V );
    float specularLight =pow( max( dot(N, H ),-dot(N, H ) ), 256 );
    if(diffuseLight <=0 ){
        specularLight =-specularLight;
    }

    vec3 specular = lightColor * specularLight;
    gl_Color.xyz = (diffuse+ .3*specular)/1.5;
    gl_Color.w = 0.5;
}

```

Appendix B. Partially mapping tables in for Shader Language Converter

B1 – Data Types/Sampler types mapping table

(same as Table 3.4 X3D, XML and shading languages basic data type mapping table)

X3D	XML	Cg	HLSL	GLSL
SFBool	bool	bool	bool	bool
MFBBool	bool[]	bool[]	bool[]	bool[]
MFInt32	float[]	float[]	int[]	int[]
SFInt32	float	float	int	int
SFFloat	float	float	float	float
MFFloat	float[]	float[]	float[]	float[]
SFDouble	double	double	double	float
MFDouble	double[]	double[]	double[]	float[]
SFTime	double	double	double	float
MFTime	double[]	double[]	double[]	float[]
SFNode	Node fields	Node fields	Node fields	Node fields
MFNode	Node fields	Node fields	Node fields	Node fields

SFVec2f	float2	float2	float2	vec2
MFVec2f	float2[]	float2[]	float2[]	vec2[]
SFVec3f	float3	float3	float3	vec3
MFVec3f	float3[]	float3[]	float3[]	vec3[]
SFVec4f	float4	float4	float4	vec4
MFVec4f	float4[]	float4[]	float4[]	vec4[]
SFVec3d	float3	float3	float3	float3
MFVec3d	float3[]	float3[]	float3[]	float3[]
SFVec4d	float4	float4	float4	float4
MFVec4d	float4[]	float4[]	float4[]	float4[]
SFRotation	float4	float4	float4	vec4
MFRotation	float4[]	float4[]	float4[]	vec4[]
MFColor	float4[]	float4[]	float4[]	vec4[]
SFColor	float4	float4	float4	vec4
SFImage	int[]	int[]	int[]	int[]
MFImage	int[]	int[]	int[]	int[]
SFString	Not supported	Not supported	Not supported	Not supported
MFString	Not supported	Not supported	Not supported	Not supported
SFMatrix3f	float3x3	float3x3	float3x3	mat3
MFMatrix3f	float3x3[]	float3x3[]	float3x3[]	mat3[]
SFMatrix4f	float4x4	float4x4	float4x4	mat4
MFMatrix4f	float4x4[]	float4x4[]	float4x4[]	mat4[]

Table 3.5 X3D, XML and shading languages Sampler type mapping table

XML	Cg	HLSL	GLSL
sampler1D	sampler1D	sampler1D	sampler1D
sampler2D	sampler2D	sampler2D	sampler2D
sampler3D	sampler3D	sampler3D	sampler3D
samplerCUBE	samplerCUBE	samplerCUBE	samplerCube
samplerRECT	samplerRECT	samplerRECT	No-support
sampler1D	sampler1D	sampler1D	sampler1DShadow
Sampler2D	Sampler2D	Sampler2D	sampler2DShadow

B2 – Operators mapping table (special case only)

XML	Cg	HLSL	GLSL
Op_mul	Mul() (one element is matrix)	Mul()	*
Op_mul	*	*	*
Op_others	others	others	others

B3 – modifiers/qualifiers mapping table

(Table 3.6 X3D, XML and shading languages basic Qualifiers mapping table)

XML	Cg	HLSL	GLSL
-----	----	------	------

const	const	uniform	uniform
uniform	uniform	varying	attribute
In, out, inout	In, out, inout	In, out, inout	varying

B4 – Transformation Matrices mapping table

(Table 3.7 X3D, XML and shading languages Transformation matrix mapping table)

XML	Cg	HLSL	GLSL
gl_NormalMatrix	gl_NormalMatrix	gl_NormalMatrix	gl_NormalMatrix
gl_ModelViewProjectionMatrix	gl_ModelViewProjectionMatrix	gl_ModelViewProjectionMatrix	gl_ModelViewProjectionMatrix
gl_ModelViewMatrix	gl_ModelViewMatrix	gl_ModelViewMatrix	gl_ModelViewMatrix

B5 – Semantics in Cg or HLSL/building variables in GLSL mapping table

(Table 3.8 XML and shading languages built-in variable mapping table)

XML	Cg	HLSL	GLSL
gl_Position	POSITION	POSITION	gl_Position
gl_Vertex	POSITION	POSITION	gl_Vertex
gl_Normal	NORMAL	NORMAL	gl_Normal
gl_Color	COLOR	COLOR	gl_Color
gl_SecondColor	COLOR0	COLOR0	gl_SecondColor
gl_MultiTexCoord0	TEXCOORD0	TEXCOORD0	gl_MultiTexCoord0
gl_MultiTexCoord1	TEXCOORD1	TEXCOORD1	gl_MultiTexCoord1
gl_MultiTexCoord2	TEXCOORD2	TEXCOORD2	gl_MultiTexCoord2
gl_MultiTexCoord3	TEXCOORD3	TEXCOORD3	gl_MultiTexCoord3
gl_MultiTexCoord4	TEXCOORD4	TEXCOORD4	gl_MultiTexCoord4
gl_MultiTexCoord5	TEXCOORD5	TEXCOORD5	gl_MultiTexCoord5
gl_MultiTexCoord6	TEXCOORD6	TEXCOORD6	gl_MultiTexCoord6
gl_MultiTexCoord7	TEXCOORD7	TEXCOORD7	gl_MultiTexCoord7

B5 – functions mapping table

For simplification, we show the maps of Cg, GLSL, and XML here only. HLSL should be very similar to Cg function set.

Function category	GLSL	XML	Cg
Basic functions & built-in functions	normalize(v)	Func_normalize	normalize(v)
	dot(a, b)	Func_dot	dot(a, b)
	mat matrixCompMult (mat x, mat y) and function operator: " * "	Func_mul(x, y) * operation (op_mul)	mul(x, y) x, y could be matrix or vector
	lerp(x, y, k)	Func_lerp	lerp(x, y, k)
	Smoothstep()	Func_Smoothstep	Smoothstep()
	reflect(I, N) (I, N could be float or vec2-4)	Func_reflect	reflect(I, N)/reflect(I, N, eta)
	step(a, x)	Func_step	step(a, x)
	clamp(x, a, b)	Func_clamp	clamp(x, a, b)
	fract()	Func_fract	fract()
	mix(x, y, a)	x*(1-a) + y*a	x*(1-a) + y*a

	<pre>//lit function : float dCol; dCol = max(diffuse, 0.0); Guidoutput.diffCol = float4(dCol); float sCol; sCol = min(diffuse, specular); if (sCol < 0) sCol = 0.0; else sCol = pow(specular, 32); Guidoutput.specCol = float4(sCol);</pre>	<pre>//lit function : float dCol; dCol = max(diffuse, 0.0); Guidoutput.diffCol = float4(dCol); float sCol; sCol = min(diffuse, specular); if (sCol < 0) sCol = 0.0; else sCol = pow(specular, 32); Guidoutput.specCol = float4(sCol);</pre>	lit(diffuse, specular,32)
	Others	Func_others	Others
Texture lookup functions	<pre>vec4 textureID (sampler1D sampler, float coord [, float bias]) vec4 textureIDProj (sampler1D sampler, vec2 coord [, float bias]) vec4 textureIDProj (sampler1D sampler, vec4 coord [, float bias]) vec4 textureIDLod (sampler1D sampler, float coord, float lod) vec4 textureIDProjLod (sampler1D sampler, vec2 coord, float lod) vec4 textureIDProjLod (sampler1D sampler, vec4 coord, float lod) Others</pre>	<pre>Func_textureID Func_textureID Func_textureID Func_textureID Func_textureIDProj Func_textureIDProj Others</pre>	<pre>texID(sampler1D tex, float s) texID(sampler1D tex, float s, float dwdx, float dwdy) texID(sampler1D tex, float sz) texID(sampler1D tex, float sz, float dwdx, float dwdy) texIDProj (sampler1D tex, float2 sq) texIDProj (sampler1D tex, float3 sqz) Others</pre>

Appendix C. Source Code

C1 – Partially Shader Language Converter Source Code

Due to the paper space, we are not going to show each line of the code (total 80 pages even in font 8). But would like to show two of the most important functions. Since the conversion is consist 3 parts symetracly. We show 2 of them, rests can follow this one. Cg2XML_Click function and XML2GLSL_Click function which developed in C#

1. Function 1 Cg2Xml_Click () function

```
// Function 1 Cg2Xml_Click ( ) function
// Author Feng Liu
// Last Modified by Sep 26 2004
private void cg2Xml_Click(object sender, System.EventArgs e)
{
    mOp2Prior=maps_Load("cgOperator2Priority.txt");
    mOp2Xml=maps_Load("cgOperator2Xml.txt");
    mQual2Xml=maps_Load("cgQualifier2xml.txt");
    mType2Xml=maps_Load("cgType2xml.txt");
    mSema2Xml=maps_Load("cgSemantic2xml.txt");
    //prepare for a xml dom tree

    XmlNode xmlnode,xmlnode2;

    //XmlText xmltext;
    //XmlAttribute xmlattr;
    //xmldoc=new XmlDocument();
    //let's add the XML declaration section
    createXmlDom();
    xmlnode=xmldoc.CreateNode(XmlNodeType.XmlDeclaration,"","");
    xmldoc.AppendChild(xmlnode);
    //*****
    string sLine1 = textBox1.Text;
    string shaderNameType = sLine1.Substring(sLine1.IndexOf("-")+1,4);
```

```

if (shaderNameType=="vert")
{
    xmlnode2=xmlDoc.CreateNode(XmlNodeType.Element,"VertexShader","");
}
else xmlnode2=xmlDoc.CreateNode(XmlNodeType.Element,"FragShader","");
xmlDoc.AppendChild(xmlnode2);
//****Feng*****
string sLine="";
//read source code from a file
fname=textBox1.Text;

fname=fname.Trim();
StreamReader objReader=null;
if(textBox1.Modified || !(richTextBox1.Modified))
{ richTextBox1.Clear();
    try
    {
        objReader= new StreamReader(fname);
        // listBox1.Items.Clear();
        while((sLine=objReader.ReadLine())!=null )
        {
            richTextBox1.AppendText(sLine+"\n");
            Console.WriteLine("sLine="+sLine);
        }
        objReader.Close();
        textBox1.Modified = false;
    }
    catch(Exception e1)
    {
        MessageBox.Show(e1.Message, "Error",MessageBoxButtons.OK,
        MessageBoxIcon.Exclamation);
        return;
    }
}
ArrayList aList=new ArrayList(richTextBox1.Lines);
ArrayList nList=new ArrayList();
aList=statement_Adjust(aList);
int i=0;
bool btag=false;
string tLine;
while(i<aList.Count )
{
    sLine=(string)aList[i];

sLine=sLine.Trim();
btag=false;
if(!btag && isStruct(sLine))
{
    int start=i;
    int tag=0;
    nList.Clear();
    while( i<aList.Count)
    {
        tLine=(string)aList[i];
        nList.Add(tLine);
        if(tLine.IndexOf("{")!=-1) tag=tag+1;
        if(tLine.IndexOf("}")!=-1) tag=tag-1;
        i=i+1;
    }
    if(tag==0) break;
}
//foreach(string s in nList)
//    listBox1.Items.Add(s);
int t=0;
xmlnode=struct_Convert(nList,ref t);
xmlnode2.AppendChild(xmlnode);
//i=i+1;
btag=true;

```

```

    }//end if isStruct
    if(!btag && isDeclaration(sLine)>0)
    {
        xmlnode=declaration_Convert(sLine);
        xmlnode2.AppendChild(xmlnode);
        btag=true;
        i=i+1;
    }//end if isDeclaration
    if(!btag && isFunc(sLine))
    {
        int start=i;
        int tag=0;
        nList.Clear();
        while( i<aList.Count)
        {
            tLine=(string)aList[i];
            nList.Add(tLine);
            if(tLine.IndexOf("{")!=-1) tag=tag+1;
            if(tLine.IndexOf("}")!=-1) tag=tag-1;

            i=i+1;
            if(tag==0) break;
        }
        //nList.Clear();

        //foreach(string s in nList)
        //    listBox1.Items.Add(s);
        int t=0;
        xmlnode=func_Convert(nList,ref t);
        xmlnode2.AppendChild(xmlnode);
        //i=i+1;
        btag=true;
    }//end if isFunc
    if(!btag) i++;
} //end while i<aList.count
xmldoc.Save(fname+".xml");
try
{
    richTextBox2.LoadFile(fname+".xml",RichTextBoxStreamType.PlainText);
    //richTextBox3.LoadFile(fname+".xml",RichTextBoxStreamType.PlainText);
}
catch(Exception e1)
{
    {MessageBox.Show(e1.Message, "Error",MessageBoxButtons.OK,
    MessageBoxIcon.Exclamation);
    return; }
}
// System.Diagnostics.Process.Start( fname+".xml" );
} // end of Cg2Xml_Click () function

```

2. // Function 2 Xml2GLSL_Click() function:

```

// Function 1 Cg2Xml_Click ( ) function
// Author Feng Liu
// Last Modified by Sep 26 2004
private void Xml2GL_Click(object sender, System.EventArgs e)
{
    // this is really xml2Cg
    mOp2Prior=maps_Load("cgOperator2Priority.txt");
    mOp2Xml=maps_Load("cgOperator2Xml.txt");
    mQual2Xml=maps_Load("cgQualifier2xml.txt");
    mType2Xml=maps_Load("cgType2xml.txt");
    mSema2Xml=maps_Load("cgSemantic2xml.txt");
    mglType2Xml=maps_Load("glType2xml.txt");
    mglFunc2Xml=maps_Load("glFunction2xml.txt");
    mglSema2Xml=maps_Load("glSemantic2xml.txt");
    mglQual2Xml=maps_Load("glQualifier2xml.txt");

    //XmlElement xmlelem;
    XmlNode xmlnode,xmlnode2;
    //XmlText xmltext;
    //XmlAttribute xmlattr;
    //xmldoc=new XmlDocument();
    //let's add the XML declaration section
    createXmlDom();
    xmlnode=xmldoc.CreateNode(XmlNodeType.XmlDeclaration,"","");
    xmldoc.AppendChild(xmlnode);
    //****Feng*****
    string sLine1 = GLSLfileName.Text;
        string shaderNameType =
sLine1.Substring(sLine1.IndexOf(".")+1,sLine1.Length-sLine1.IndexOf(".")-1);

    if (shaderNameType=="vert")
    {
        xmlnode2=xmldoc.CreateNode(XmlNodeType.Element,"VertexShader","");
    }
    else xmlnode2=xmldoc.CreateNode(XmlNodeType.Element,"FragShader","");
    xmldoc.AppendChild(xmlnode2);
    //****Feng*****
    string sLine="";
    //read source code from a file
    string fname=GLSLfileName.Text;
    fname=fname.Trim();
    StreamReader objReader=null;
    if(GLSLfileName.Modified || !(richTextBox3.Modified))
    {
        richTextBox3.Clear();
        try
        {
            objReader= new StreamReader(fname);
            // listBox1.Items.Clear();
            while((sLine=objReader.ReadLine())!=null )
            {
                richTextBox3.AppendText(sLine+"\n");
                Console.WriteLine("sLine="+sLine);
            }//end while objReader.readline()
            objReader.Close();
            GLSLfileName.Modified = false;
        }
        catch(Exception e1)
        {
            MessageBox.Show(e1.Message, "Error",MessageBoxButtons.OK,
            MessageBoxIcon.Exclamation);
            return; }
    }
    ArrayList aList=new ArrayList(richTextBox3.Lines);
    ArrayList nList=new ArrayList();

```

```

aList=statement_Adjust(aList);
ArrayList pList=glStatement_Adjust(aList);
int i=0;
bool btag=false;

string tLine;
while(i<aList.Count )
{
    sLine=(string)aList[i];
    sLine=sLine.Trim();
    btag=false;

    if(isglDeclaration(sLine)>0)
    {
        xmlnode=glDeclaration_Convert(sLine);
        xmlnode2.AppendChild(xmlnode);
        btag=true;
        i=i+1;
    }//end if isDeclaration
    if(isglFunc(sLine))
    {
        int start=i;
        int tag=0;
        nList.Clear();
        while( i<aList.Count)
        {
            tLine=(string)aList[i];
            nList.Add(tLine);
            if(tLine.IndexOf("{")!=-1) tag=tag+1;
            if(tLine.IndexOf("}")!=-1) tag=tag-1;

            i=i+1;
            if(tag==0) break;
        }
        //nList.Clear();

        //foreach(string s in nList)
        //    listBox1.Items.Add(s);
        int t=0;
        xmlnode=glFunc_Convert(nList,ref t);
        xmlnode2.AppendChild(xmlnode);
        //i=i+1;
        btag=true;
    }//end if isFunc
    if(!btag) i++;

} //end while i<aList.count
xmldoc.Save(fname+".xml");
try
{
    richTextBox2.LoadFile(fname+".xml",RichTextBoxStreamType.PlainText);
}
catch(Exception e1)
{
    MessageBox.Show(e1.Message, "Error",MessageBoxButtons.OK,
    MessageBoxIcon.Exclamation);
    return;
}
// System.Diagnostics.Process.Start( fname+".xml" );
} // end of Xml2GLSL function

```

C2 – XSL source code for CML to X3D + shader conversion (protoDeclaration part only)

```

<?xml version='1.0'?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

<!--
  <Header>
    <meta name="filename"      content="cmlToX3d_3.xsl" />
    <meta name="Orig-author"   content="Nicholas F. Polys" />
    <meta name="author"        content="Feng Liu : Modify for include shader in" />
    <meta name="revised"       content="March 1 2005 by Feng Liu" />
    <meta name="description"   content=" XSL stylesheet to convert CML-XML files to X3D
files for Xj3D" />
    <meta name="url"
content="http://www.web3D.org/TaskGroups/x3d/translation/cml2X3d.xsl" />
  </Header>

  Recommended tools:
  - Gnome Libs and Perl Modules
  - SAXON XML Toolkit (and Instant Saxon) http://users.iclway.co.uk/mhkay/saxon
  - Can also be used with Apache server, Xalan & Cacoon

-->

  <!--xsl:output method="xml" encoding="UTF-8" media-type="model/x3d+xml" cdata-section-
elements="Script"
indent="yes" doctype-system="http://www.web3d.org/specifications/x3d-3.0.dtd"/-->

<xsl:output method="xml" encoding="UTF-8" media-type="model/x3d+xml" cdata-section-
elements="Script"
indent="yes" doctype-system="x3d-3.0.dtd"/>

<!--The folowing part are predefined Carbon with Phong shader as appearance-->
<ProtoDeclare name="Carbon">
  <ProtoInterface>
    <field accessType="inputOutput" name="position" type="SFVec3f"/>
    <field accessType="inputOutput" name="Mat" type="SFFloat" value=".6"/>
  </ProtoInterface>
  <ProtoBody>
    <Group>
      <Transform DEF="atoC">
        <IS>
          <connect nodeField="translation" protoField="position"/>
        </IS>
      <Shape>
        <!--#####-->
        <ShaderAppearance>
          <VertexShader url="Carbon-v.cg"/>
          <FragmentShader url="Carbon-f.cg">
            <field accessType="inputOutput" name="baseColor" type="SFColor" value
="0.6 0.6 0.6"/>
            <field accessType="inputOutput" name="lightPosition" type="SFVec3f" value
="-10 10 -10"/>
          </FragmentShader>
        </ShaderAppearance>
        <!--#####-->
        <Sphere radius=".68"/>
      </Shape>
    </Group>
  </ProtoBody>
</ProtoDeclare> ... ..

```

C2 – XSL source code for X3D + shader 2 VRML + shader conversion

```

<?xml version='1.0'?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
                xmlns:saxon="http://icl.com/saxon" saxon:trace="no">

<!-- XSL namespaces are in transition! Tools are slow to catch up.
     *** Edit the topmost stylesheet tag on line 2 of this file to match the xmlns
namespace URI for your XSL tool. ***
W3C:
Saxon:          <xsl:stylesheet xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
IBM XSLEditor:  <xsl:stylesheet xmlns:xsl='http://www.w3.org/XSL/Transform/1.0'>
IE 5:          <xsl:stylesheet xmlns:xsl='http://www.w3.org/TR/WD-xsl'>
XT:            <xsl:stylesheet xmlns:xsl='http://www.w3.org/XSL/Transform'>
-->
<!--
<head>
  <meta name="filename"           content="X3dToVrml97.xslt" />
  <meta name="author"            content="Don Brutzman" />
  <meta name="LatticeXvl author"  content="Marc Jablonski" />
  <meta name="revised"           content="25 May 2003" />
  <meta name="revised"           content="1 March 2005" by Feng Liu added shader
appearance support/>
  <meta name="description" content="XSL stylesheet to convert X3D files to VRML 97
format, fourth draft matching x3d-compromise.dtd" />
  <meta name="url"
content="http://www.web3D.org/TaskGroups/x3d/translation/X3dToVrml97.xslt" />
</head>

Recommended tool:
- SAXON XML Toolkit (and Instant Saxon) from Michael Kay of ICL,
http://saxon.sourceforge.net
- Can also be used with Apache server -->
<!-- * recurse through each of the tree node elements, including ProtoInstance * -->
<xsl:template match="*">
  <xsl:param name="indent"><xsl:text>0</xsl:text></xsl:param>
  .....
          local-name()='ProtoInstance' or $nodeType='ProtoInstance' or
$EParentType='ProtoInstance' or @nodeType='ProtoInstance')
          and (local-name(..)='LoadSensor' or $parentType='LoadSensor'
or $EParentType='LoadSensor' or ../@nodeType='LoadSensor' and
@containerField='watchList') ">
    <!-- appears to be a valid watchList node --></xsl:when>
    <!-- 4-way node-type tests: native VRML node, ProtoDeclared ProtoInstance,
ExternProtoDeclared ProtoInstance, or internal DTD declaration by content -->
<!--#####-->
    <xsl:when test="local-name()='ShaderAppearance' or
$nodeType='ShaderAppearance' or $EParentType='ShaderAppearance' or
@nodeType='ShaderAppearance' "><xsl:text>appearance </xsl:text></xsl:when>
    <xsl:when test="local-name()='VertexShader' or $nodeType='VertexShader'
or $EParentType='VertexShader' or @nodeType='VertexShader'
"><xsl:text>vertexShader </xsl:text></xsl:when>
    <xsl:when test="local-name()='FragmentShader' or $nodeType='FragmentShader'
or $EParentType='FragmentShader' or @nodeType='FragmentShader'
"><xsl:text>fragmentShader </xsl:text></xsl:when>
<!--#####-->
    <xsl:when test="local-name()='Appearance' or $nodeType='Appearance'
or $EParentType='Appearance' or @nodeType='Appearance' "><xsl:text>appearance
</xsl:text></xsl:when>
    <xsl:when test="local-name()='AudioClip' or $nodeType='AudioClip'
or $EParentType='AudioClip' or @nodeType='AudioClip' "><xsl:text>source
</xsl:text></xsl:when>
    <xsl:when test="local-name()='Box' or $nodeType='Box'

```

```

<!--#####-->
    <xsl:when test="(local-name()='MovieTexture' or $nodeType='MovieTexture'
or $EPnodeType='MovieTexture' or @nodeType='MovieTexture')
and (local-name(..)='ShaderAppearance' or
$parentType='ShaderAppearance' or $EPparentType='ShaderAppearance' or
../@nodeType='ShaderAppearance' or
local-name(..)='texture' ) "><xsl:text>texture </xsl:text></xsl:when>

    <xsl:when test="(local-name()='MovieTexture' or $nodeType='MovieTexture'
or $EPnodeType='MovieTexture' or @nodeType='MovieTexture')
and (local-name(..)='Sound' or $parentType='Sound'
or $EPparentType='Sound' or ../@nodeType='Sound' or
local-name(..)='source' ) "><xsl:text>source
</xsl:text></xsl:when>
    <xsl:when test="((local-name()='Normal' or $nodeType='Normal'
or $EPnodeType='Normal' or @nodeType='Normal') and
not(@containerField='skinNormal'))"><xsl:text>normal </xsl:text></xsl:when>
    <xsl:when test="local-name()='PixelTexture' or $nodeType='PixelTexture'
or $EPnodeType='PixelTexture' or @nodeType='PixelTexture'
<xsl:with-param name="DEF" select="../../@DEF"/>
    </xsl:call-template>
</xsl:when>
<xsl:when test="@nodeField and
(preceding::ProtoDeclare[@name=$protoName]/ProtoInterface/field[@name=$nodeField])">
</xsl:when>

```

C3 – example caffeine in X3D_With_Shader

```

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE X3D
  SYSTEM "x3d-3.0.dtd">
<X3D profile="Immersive">
  <head>
    <meta content="translatedCML.x3d" name="filename"/>
    <meta content="http://www.xml-cml.org" name="description"/>
    <meta content="today" name="revised"/>
    <meta content="cml2x3d.xsl" name="author"/>
  </head>
  <Scene>
    <ProtoDeclare name="Carbon">
      <ProtoInterface>
        <field accessType="inputOutput" name="position" type="SFVec3f"/>
        <field accessType="inputOutput" name="Mat" type="SFFloat" value=".6"/>
      </ProtoInterface>
      <ProtoBody>
        <Group>
          <Transform DEF="atoC">
            <IS>
              <connect nodeField="translation" protoField="position"/>
            </IS>
            <Shape>
              <ShaderAppearance>
                <VertexShader url="test-v.cg"/>
                <FragmentShader url="test-f.cg">
                  <field accessType="inputOutput" name="baseColor"
                    type="SFColor" value="0.6 0.6 0.6"/>
                  <field accessType="inputOutput" name="lightPosition"
                    type="SFVec3f" value="-10 10 -10"/>
                </FragmentShader>
                <Material DEF="atoC_mat" diffuseColor="0 0 0" shininess=".8"
                  specularColor=".29 .3 .29">
                  <IS>
                    <connect nodeField="transparency" protoField="Mat"/>
                  </IS>
                </Material>
              </ShaderAppearance>
              <Sphere radius=".68"/>
            </Shape>
            <Shape>
              <Appearance/>
              <Text string="C">
                <FontStyle size=".8"/>
              </Text>
            </Shape>
          </Transform>
        </Group>
      </ProtoBody>
    </ProtoDeclare>
    <ProtoDeclare name="Hydrogen">
      <ProtoInterface>
        <field accessType="inputOutput" name="position" type="SFVec3f"/>
        <field accessType="inputOutput" name="Mat" type="SFFloat" value=".6"/>
      </ProtoInterface>
      <ProtoBody>
        <Group>
          <Transform DEF="atoH">
            <IS>
              <connect nodeField="translation" protoField="position"/>
            </IS>
            <Shape>
              <ShaderAppearance>
                <VertexShader url="Fur.fx">

```

```

        <field accessType="inputOutput" name="GlowColor"
        type="SFVec4f" value="0.98 0.98 0.98 0.4"/>
        <field accessType="inputOutput" name="GlowThickness"
        type="SFFloat" value="0.015"/>
    </VertexShader>
    <Material DEF="atoC_mat" diffuseColor="0 0 0" shininess=".8"
    specularColor=".29 .3 .29">
        <IS>
            <connect nodeField="transparency" protoField="Mat"/>
        </IS>
    </Material>
</ShaderAppearance>
<Sphere radius=".30"/>
</Shape>
<Shape>
    <Appearance/>
    <Text string="H">
        <FontStyle size=".4"/>
    </Text>
</Shape>
</Transform>
</Group>
</ProtoBody>
</ProtoDeclare>
<ProtoDeclare name="Nitrogen">
    <ProtoInterface>
        <field accessType="inputOutput" name="position" type="SFVec3f"/>
        <field accessType="inputOutput" name="Mat" type="SFFloat" value=".6"/>
    </ProtoInterface>
    <ProtoBody>
        <Group>
            <Transform DEF="atoN">
                <IS>
                    <connect nodeField="translation" protoField="position"/>
                </IS>
            </Transform>
            <Shape>
                <ShaderAppearance transparent="TRUE">
                    <VertexShader url="test-v.cg"/>
                    <FragmentShader url="test-f.cg">
                        <field accessType="inputOutput" name="baseColor"
                        type="SFCColor" value="0.56 0.56 .98"/>
                        <field accessType="inputOutput" name="lightPosition"
                        type="SFVec3f" value="-10 10 -10"/>
                    </FragmentShader>
                    <Material DEF="atoC_mat" diffuseColor="0 0 0" shininess=".8"
                    specularColor=".29 .3 .29">
                        <IS>
                            <connect nodeField="transparency" protoField="Mat"/>
                        </IS>
                    </Material>
                </ShaderAppearance>
                <Sphere radius=".65"/>
            </Shape>
            <Shape>
                <Appearance/>
                <Text string="N">
                    <FontStyle size=".8"/>
                </Text>
            </Shape>
        </Transform>
    </Group>
</ProtoBody>
</ProtoDeclare>
<ProtoDeclare name="Oxygen">
    <ProtoInterface>
        <field accessType="inputOutput" name="position" type="SFVec3f"/>
        <field accessType="inputOutput" name="Mat" type="SFFloat" value=".6"/>
    </ProtoInterface>

```

```

</ProtoInterface>
<ProtoBody>
  <Group>
    <Transform DEF="ato0">
      <IS>
        <connect nodeField="translation" protoField="position"/>
      </IS>
      <Shape>
        <ShaderAppearance>
          transparent TRUE

          <VertexShader url="test-v.cg"/>
          <FragmentShader url="test-f.cg">
            <field accessType="inputOutput" name="baseColor"
              type="SFColor" value=".98 .0 .0"/>
            <field accessType="inputOutput" name="lightPosition"
              type="SFVec3f" value="-10 10 -10"/>
          </FragmentShader>
          <Material DEF="atoC_mat" diffuseColor="0 0 0"
            shininess=".8" specularColor=".29 .3 .2">
            <IS>
              <connect nodeField="transparency" protoField="Mat"/>
            </IS>
          </Material>
        </ShaderAppearance>
        <Sphere radius=".63"/>
      </Shape>
      <Shape>
        <Appearance/>
        <Text string="0">
          <FontStyle size=".8"/>
        </Text>
      </Shape>
    </Transform>
  </Group>
</ProtoBody>
</ProtoDeclare>
<ProtoDeclare name="Sulphur">
  <ProtoInterface>
    <field accessType="inputOutput" name="position" type="SFVec3f"/>
    <field accessType="inputOutput" name="Mat" type="SFFloat" value=".6"/>
  </ProtoInterface>
  <ProtoBody>
    <Group>
      <Transform DEF="atoS">
        <IS>
          <connect nodeField="translation" protoField="position"/>
        </IS>
        <Shape>
          <ShaderAppearance>
            <VertexShader url="test-v.cg"/>
            <FragmentShader url="test-f.cg">
              <field accessType="inputOutput" name="baseColor"
                type="SFColor" value=".98 .78 .20"/>
              <field accessType="inputOutput" name="lightPosition"
                type="SFVec3f" value="-10 10 -10"/>
            </FragmentShader>
            <Material DEF="atoC_mat" diffuseColor="0 0 0"
              shininess=".8" specularColor=".29 .3 .29">
              <IS>
                <connect nodeField="transparency" protoField="Mat"/>
              </IS>
            </Material>
          </ShaderAppearance>
          <Sphere radius="1.3"/>
        </Shape>
      </Transform>
    </Group>
  </ProtoBody>
</ProtoDeclare>

```

```

        <Appearance/>
        <Text string="S">
            <FontStyle size=".8"/>
        </Text>

    </Shape>
</Transform>
</Group>
</ProtoBody>
</ProtoDeclare>
<ProtoDeclare name="unknown">
    <ProtoInterface>
        <field accessType="inputOutput" name="position" type="SFVec3f"/>
        <field accessType="inputOutput" name="Mat" type="SFFloat" value=".6"/>
    </ProtoInterface>
    <ProtoBody>
        <Group>
            <Transform DEF="ato_">
                <IS>
                    <connect nodeField="translation" protoField="position"/>
                </IS>
            </Transform>
            <Shape>
                <ShaderAppearance>
                    <VertexShader url="test-v.cg"/>
                    <FragmentShader url="test-f.cg">
                        <field accessType="inputOutput" name="baseColor"
                            type="SFCOLOR" value=".0 .98 .0"/>
                        <field accessType="inputOutput" name="lightPosition"
                            type="SFVec3f" value="-10 10 -10"/>
                    </FragmentShader>
                    <Material DEF="atoC_mat" diffuseColor="0 0 0"
                        shininess=".8" specularColor=".29 .3 .29">
                        <IS>
                            <connect nodeField="transparency" protoField="Mat"/>
                        </IS>
                    </Material>
                </ShaderAppearance>

                <Sphere radius="0.6"/>
            </Shape>
            <Shape>
                <Appearance/>
                <Text string="?">
                    <FontStyle size=".8"/>
                </Text>
            </Shape>
        </Transform>
    </Group>
</ProtoBody>
</ProtoDeclare>
<Group>
    <Transform>
        <ProtoInstance name="Carbon" DEF="caffeine_karne_a_1">
            <fieldValue name="position" value="-2.8709 -1.0499 0.1718"/>
        </ProtoInstance>
    </Transform>
    <Transform translation="0 0 0">
        <ProtoInstance name="line" DEF="caffeine_karne_b_25">
            <fieldValue name="bond_set" value="-1.6764 -3.1997 0.1458, -2.0682 -3.5218
1.1381"/>
        </ProtoInstance>
    </Transform>
</Group>
</Scene>
</X3D> <!--end of the X3D with Shader caffeinn structure-->

```

C3b – example caffeine in VRML_Shader

```

#VRML V2.0 utf8
# X3D-to-VRML-97 XSL translation autogenerated by X3dToVrml97.xsl
# http://www.web3D.org/TaskGroups/x3d/translation/X3dToVrml97.xsl

# [X3D] VRML V3.0 utf8
# [X3D] profile=Immersive
# [X3D] noNamespaceSchemaLocation=http://www.web3d.org/specifications/x3d-3.0.xsd
# [X3D] version=3.0

# [head]
# [meta] filename: translatedCML.x3d
# [meta] description: http://www.xml-cml.org
# [meta] revised: today
# [meta] author: cml2x3d.xsl
# [Scene]

PROTO Carbon [
  exposedField SFVec3f position 0 0 0
  exposedField SFFloat Mat .6
] {
  Group {
    children [
      DEF atoC Transform {
        translation IS position
        children [
          Shape {
            appearance ShaderAppearance {
              vertexShader VertexShader {
                url [ "test-v.cg" ]
              }
              fragmentShader FragmentShader {
                exposedField SFColor baseColor 0.6 0.6 0.6
                exposedField SFVec3f lightPosition -10 10 -10
                url [ "test-f.cg" ]
              }
            }
            material DEF atoC_mat Material {
              diffuseColor 0 0 0
              shininess .8
              specularColor .29 .3 .29
              transparency IS Mat
            }
          }
          geometry Sphere {
            radius .68
          }
        ]
      }
      Shape {
        appearance Appearance {
        }
        geometry Text {
          string [ "C" ]
          fontStyle FontStyle {
            size .8
          }
        }
      }
    ]
  }
}

PROTO Hydrogen [
  exposedField SFVec3f position 0 0 0
  exposedField SFFloat Mat .6

```

```

] {
  Group {
    children [
      DEF atoH Transform {
        translation IS position
        children [
          Shape {
            vertexShader VertexShader {
              url [ "test-v.cg" ]
            }
            fragmentShader FragmentShader {
              exposedField SFCOLOR      baseColor 0.98 0.98 0.98
              exposedField SFVec3f      lightPosition -10 10 -10
              url [ "test-f.cg" ]
            }
            material DEF atoC_mat Material {
              diffuseColor 0 0 0
              shininess .8
              specularColor .29 .3 .29
              transparency IS Mat
            }
          }
          geometry Sphere {
            radius .30
          }
          Shape {
            appearance Appearance {
            }
            geometry Text {
              string [ "H" ]
              fontStyle FontStyle {
                size .4
              }
            }
          }
        ]
      }
    ]
  }
}
PROTO Nitrogen [
  exposedField SFVec3f position 0 0 0
  exposedField SFFloat Mat .6
] {
  Group {
    children [
      DEF atoN Transform {
        translation IS position
        children [
          Shape {
            appearance ShaderAppearance {
              transparent TRUE
            }
            vertexShader VertexShader {
              url [ "test-v.cg" ]
            }
            fragmentShader FragmentShader {
              exposedField SFCOLOR      baseColor 0.56 0.56 .98
              exposedField SFVec3f      lightPosition -10 10 -10
              url [ "test-f.cg" ]
            }
            material DEF atoC_mat Material {
              diffuseColor 0 0 0
              shininess .8
              specularColor .29 .3 .29
              transparency IS Mat
            }
          }
        ]
      }
    ]
  }
}

```

```

        geometry Sphere {
            radius .65
        }
    }
    Shape {
        appearance Appearance {
        }
        geometry Text {
            string [ "N" ]
            fontStyle FontStyle {
                size .8
            }
        }
    }
}
]
}
]
}
}
PROTO Oxygen [
    exposedField SFVec3f position 0 0 0
    exposedField SFFloat Mat .6
] {
    Group {
        children [
            DEF atoO Transform {
                translation IS position
            }
            children [
                Shape {
                    appearance ShaderAppearance {
                        vertexShader VertexShader {
                            url [ "test-v.cg" ]
                        }
                        fragmentShader FragmentShader {
                            exposedField SFColor    baseColor .98 .0 .0
                            exposedField SFVec3f    lightPosition -10 10 -10
                            url [ "test-f.cg" ]
                        }
                    }
                    material DEF atoC_mat Material {
                        diffuseColor 0 0 0
                        shininess .8
                        specularColor .29 .3 .2
                        transparency IS Mat
                    }
                }
                geometry Sphere {
                    radius .63
                }
            }
            Shape {
                appearance Appearance {
                }
                geometry Text {
                    string [ "O" ]
                    fontStyle FontStyle {
                        size .8
                    }
                }
            }
        ]
    }
}
] }
}
PROTO Sulphur [
    exposedField SFVec3f position 0 0 0
    exposedField SFFloat Mat .6
] {
    Group {

```



```
Transform {  
  children [  
    DEF caffeine_karne_b_25 line {  
      bond_set [ -1.6764 -3.1997 0.1458, -2.0682 -3.5218 1.1381 ]  
    }  
  ]  
}  
}  
} // eof cafeinn.cml.x3d_shader.vrml
```

C4c – specular light adjustable Interface example functions and ROUTES

```

#VRML V2.0 utf8
#Simple_Specular_Light_Control.wrl
# June 8 2005 Written by Feng Liu

PROTO My_P [

    exposedField SFFloat phongN 100
    exposedField SFFloat transparency 0.3
                                exposedField SFCOLOR Ka 0.8 0.6 0.4

] {

Shape {
    appearance ShaderAppearance {
        transparent TRUE
        vertexShader VertexShader {
            url [ "phong_shader/phong_vert.cg" ]
        }
        fragmentShader DEF Hydrogen_Frag FragmentShader {

                                exposedField SFCOLOR Ka IS Ka
                                field SFCOLOR Kd 0.8 0.6 0.4
                                field SFCOLOR Ks 0.7 0.7 0.7
                                field SFCOLOR ambientLight 0.2 0.2 0.2
                                exposedField SFFloat phongN IS phongN
                                field SFCOLOR lightColor 1.0 1.0 1.0
                                field SFVec3f lightPosition 0 0 15
                                exposedField SFFloat transparency IS transparency

                                url [ "phong_shader/phong_frag.cg" ]

        }
    }
    geometry Sphere { radius .20 }
}

Transform { translation 0 0 0
    children [
        DEF my_P_1 My_P {
            # phongN 100
        }
    ]
}

Transform { translation 1 0 0
    children [
        DEF my_P_1 My_P {
            # phongN 100
        }
    ]
}

# define a sensor

Transform {
    translation -2 0 0
    children [

                                DEF slidesensor PlaneSensor {
                                minPosition 0.0 0.0
                                maxPosition 1.5 0.0
                                offset 0 0 0

```

```

        autoOffset TRUE
    }
    DEF regler Transform {
    scale 3 6 3
    children Shape {
        appearance Appearance {
            material Material {
                diffuseColor .7 0 0
                specularColor .6 .1 .2
                shininess .8
            }
            geometry Sphere { radius .05 }
        }
    }
    Transform {
    translation 0.56 0 0
    rotation 0 0 1 1.5708
    children Shape {
        appearance Appearance {
            material Material {
                diffuseColor .3 .3 .3
                specularColor .7 .7 .8
                shininess .2
            }
            geometry Cylinder { radius .1 height 2 }
        }
    }
}

DEF fractionscript Script {
eventIn SFVec3f set_translation
eventOut          SFFloat oneDFloat
eventOut          SFCOLOR threeDFloat

url "vrmlscript:
function set_translation(value) {
oneDFloat = ((value[0])*10000/100);
threeDFloat [0]= value[0];
//threeDFloat [1]= value[1];
}"

ROUTE slidesensor.translation_changed TO regler.set_translation
ROUTE slidesensor.translation_changed TO fractionscript.set_translation
ROUTE fractionscript.oneDFloat TO my_P_1.phongN

ROUTE fractionscript.oneDFloat TO my_P_1.transparency
ROUTE fractionscript.threeDFloat TO my_P_1.Ka
# EOF Simple_Specular_Light_Control.wrl

```

C4e – bump adjustable Interface example functions and ROUTEs

```

#VRML V2.0 utf8
#Simple_BUMP_Control.wrl
# June 8 2005 Written by Feng Liu

PROTO My_P [
    exposedField SFFloat phongN 100
    exposedField SFFloat transparency 0.3
    exposedField SFCOLOR Ka 0.8 0.6 0.4
] {
Shape {
    appearance ShaderAppearance {
        transparent TRUE
        vertexShader VertexShader {
            url [ "phong_shader/phong_vert.cg" ]
        }
        fragmentShader DEF Hydrogen_Frag FragmentShader {

```

```

        exposedField SFCOLOR Ka IS Ka
        field SFCOLOR Kd 0.8 0.6 0.4
        field SFCOLOR Ks 0.7 0.7 0.7
        field SFCOLOR ambientLight 0.2 0.2 0.2
        exposedField SFFloat phongN IS phongN
        field SFCOLOR lightColor 1.0 1.0 1.0
        field SFVec3f lightPosition 0 0 15
        exposedField SFFloat transparency IS transparency
    url [ "phong_shader/phong_frag.cg" ]
}
}
geometry Sphere { radius .20 }
}
}
Transform { scale .5 0.5 0.5
children[
Shape {
    appearance ShaderAppearance {
        transparent TRUE
        vertexShader DEF VertexShader1 VertexShader
        {
            url "BumpPlastic.fx"
            exposedField SFVec4f LightPos 1 0.5 2 0
            exposedField SFFloat Bumpy 5

            exposedField SFNode diffuseMap ImageTexture { url "dna_1-1.jpg" }
            exposedField SFNode normalMap ImageTexture { url "dna_1-1_hf.jpg" }
        }
    }
    geometry DEF _IndexedFaceSet IndexedFaceSet {
        coord Coordinate {point [
            3.82137e-015 -1 8.74228e-008,-8.52767e-009 -0.980785 -0.19509,-1.67276e-008
            -0.92388 -0.382683,-2.42847e-008 -0.83147 -0.55557,
            .
            .
            .
            texCoord
            MultiTextureCoordinate { coord
            [TextureCoordinate {point [
            1 0,1 0.0625,1 0.125,1 0.1875,
            .
            .
            .
# define a sensor
    Transform {
        translation -2 0 0
        children [
            DEF slidesensor PlaneSensor {
                minPosition 0.0 0.0
                maxPosition 1.5 0.0
                offset 0 0 0
                autoOffset TRUE
            }
            DEF regler Transform {
                scale 3 6 3
                children Shape {
                    appearance Appearance {
                        material Material {
                            diffuseColor .7 0 0
                            specularColor .6 .1 .2
                            shininess .8

```

```

    }}
    geometry Sphere { radius .05 }
  }}

  Transform {
    translation 0.56 0 0
    rotation 0 0 1 1.5708
    children Shape {
      appearance Appearance {
        material Material {
          diffuseColor .3 .3 .3
          specularColor .7 .7 .8
          shininess .2
        }}
      geometry Cylinder { radius .1 height 2 }
    }}
  }}

  DEF fractionscript Script {
    eventIn SFVec3f set_translation
    eventOut SFFloat oneDFloat
    eventOut SFCOLOR threeDFloat

    url "vrmlscript:
    function set_translation(value) {
      oneDFloat = ((value[0])*10000/100);
      threeDFloat [0]= value[0];
      //threeDFloat [1]= value[1];
    }"

    ROUTE slidesensor.translation_changed TO regler.set_translation
    ROUTE slidesensor.translation_changed TO fractionscript.set_translation
    ROUTE fractionscript.oneDFloat TO VertexShader1.Bumpy
    # EOF Simple_BUMP_Control.wrl
  }

```

C5 – MUSE bump sharing Interface example functions and ROUTEs (partially on network node information exchange)

```

#VRML V2.0 utf8
#Simple_BUMP_Control.wrl
# June 8 2005 Written by Feng Liu

#Matrix PROTOs

PROTO NetworkSFVec3f [
  eventIn SFVec3f set_value
  eventOut SFVec3f value_changed
  eventIn SFVec3f value_fromnet
  eventOut SFVec3f value_tonet
  exposedField SFString tag ""
  exposedField SFBool pilotOnly TRUE
  field SFBool localCopy TRUE
  exposedField SFBool echo TRUE
  exposedField SFBool cont FALSE
]
{
  Script {
    eventIn SFVec3f InSc IS set_value
    eventOut SFVec3f OutSc IS value_changed
    eventIn SFVec3f netInSc IS value_fromnet
    eventOut SFVec3f netOutSc IS value_tonet
    field SFBool local IS localCopy
  }
}

```

```

directOutput TRUE
mustEvaluate TRUE

url "javascript:
  function InSc(value) {
    netOutSc = value;
    if( local == true )
      OutSc = value;
  }
  function netInSc(value) {
    OutSc = value;
  }
"
}
.
.
.

DEF Side_view1 Viewpoint {
  position 250 40 0
  orientation 0 1 0 1.5708
  fieldOfView 0.7
  description "Side_view1"
}
]
}

#ende Matrix things

#VRML V2.0 utf8

PROTO My_P [

  exposedField SFFloat phongN 100
  exposedField SFFloat transparency 0.3
  exposedField SFCOLOR Ka 0.8 0.6 0.4

]
{
Shape {
  appearance ShaderAppearance {
    transparent TRUE
    vertexShader VertexShader {
      url [ "phong_shader/phong_vert.cg" ]
    }
    fragmentShader DEF Hydrogen_Frag FragmentShader {

      exposedField SFCOLOR Ka IS Ka
      field SFCOLOR Kd 0.8 0.6 0.4
      field SFCOLOR Ks 0.7 0.7 0.7
      field SFCOLOR ambientLight 0.2 0.2 0.2
      exposedField SFFloat phongN IS phongN
      field SFCOLOR lightColor 1.0 1.0 1.0
      field SFVec3f lightPosition 0 0 15
      exposedField SFFloat transparency IS transparency

      url [ "phong_shader/phong_frag.cg" ]

    }
  }
  geometry Sphere { radius .20 }
}
}
Transform { scale .5 0.5 0.5
children[

```

```

transparent TRUE
vertexShader DEF VertexShader1 VertexShader
{
    url "BumpPlastic.fx"
    exposedField SFVec4f LightPos 1 0.5 2 0
    exposedField SFFloat Bumpy 5
    exposedField SFNode diffuseMap ImageTexture { url "dna_1-1.jpg" }
    exposedField SFNode normalMap ImageTexture { url "dna_1-1_hf.jpg" }
}
}
geometry DEF _IndexedFaceSet IndexedFaceSet {
    coord Coordinate {point [
        3.82137e-015 -1 8
        .
        .
    ]}

    DEF fractionscript Script {
        eventIn SFVec3f set_translation
        eventOut SFFloat oneDFloat
        eventOut SFColor threeDFloat

        url "vrlmscript:
        function set_translation(value) {
            oneDFloat = ((value[0])*10000/100);
            threeDFloat [0]= value[0];
            //threeDFloat [1]= value[1];
        }"
    }

ROUTE BKmover.translation_changed TO BKnet.set_value
ROUTE BKnet.value_changed TO blackking.translation

ROUTE slidesensor.translation_changed TO regler.set_translation
ROUTE slidesensor.translation_changed TO fractionscript.set_translation
ROUTE fractionscript.oneDFloat TO VertexShader1.Bumpy
ROUTE fractionscript.oneDFloat TO BKnet.set_value
ROUTE BKnet.value_changed TO VertexShader1.translation

```

Appendix D – Image Credits

Figure 2 - 1 Plastic shader on the spherical surface

Figure 2 - 2 Graphics Pipeline

Figure 2 - 3 GPU – CPU Interface in modern Graphics Pipeline

Figure 2 - 4 Cg Shaders loading process

Figure 2 - 5 GLSL Shaders loading process

Figure 2 - 6 Google Hit of 3D web websites

Figure 2 - 7 X3D / VRML Growing history

Figure 2 - 8 Shaders in VRML/X3D web browser on single machine

Figure 3 - 9 Coordinate System and Transformation for Vertex Processing

Figure 4 - 0 Beauty of life – Picture from Heaven & Earth by Phaidon

Figure 4 - 1 Protein example

Figure 4 - 2 Size of molecular

Figure 4 - 3 HIV protease structure

Figure 4 - 4 HIV protease cell

Figure 4 - 7 ShaderAppearance Prototype in VRML

Figure 4 - 8 VertexShader Prototype in VRML

Figure 4 - 9 FragmentShader Prototype in VRML

Appendix E – code Credits

Example 2.1 Renderman plastic shader code fragment from Pixar's *The RenderMan Interface*

Example 2.3 vertex shader of brick in GLSL from Rost's *The OpenGL Shading Language*

Example 2.4 vertex shader of wood in high level shading language from [Peeper and Mitchell 2002].

Example 3.3 Cg vp 20 binding example from nVidia *Cg Tutorial 1.0*

Example 4.2 an example of caffeine CML structure from [Polys 2003]

Example 4.4 A touch sensor example in X3D from [Krone 2003]

Appendix F – SLC source code (Available with requirement)