SVG 3D Graphical Presentation for Web-based Applications

Lu Jisheng

A thesis submitted to

The University of Gloucestershire

In accordance with the requirements of the degree of

Doctor of Philosophy

In the Faculty of Media, Arts and Technology

February, 2015

ABSTRACT

Due to the rapid developments in the field of computer graphics and computer hardware, web-based applications are becoming more and more powerful, and the performance distance between web-based applications and desktop applications is increasingly closer. The Internet and the WWW have been widely used for delivering, processing, and publishing 3D data. There is increasingly demand for more and easier access to 3D content on the web. The better the browser experience, the more potential revenue that web-based content can generate for providers and others.

The main focus of this thesis is on the design, develop and implementation of a new 3D generic modelling method based on Scalable Vector Graphics (SVG) for web-based applications. While the model is initialized using classical 3D graphics, the scene model is extended using SVG. A new algorithm to present 3D graphics with SVG is proposed. This includes the definition of a 3D scene in the framework, integration of 3D objects, cameras, transformations, light models and textures in a 3D scene, and the rendering of 3D objects on the web page, allowing the end-user to interactively manipulate objects on the web page.

A new 3D graphics library for 3D geometric transformation and projection in the SVG GL is design and develop.

A set of primitives in the SVG GL, including triangle, sphere, cylinder, cone, etc. are designed and developed.

A set of complex 3D models in the SVG GL, including extrusion, revolution, Bezier surface, and point clouds are designed and developed.

The new Gouraud shading algorithm and new Phong Shading algorithm in the SVG GL are proposed, designed and developed. The algorithms can be used to generate smooth shading and create highlight for 3D models.

The new texture mapping algorithms for the SVG GL oriented toward web-based 3D modelling applications are proposed, designed and developed. Texture mapping algorithms for different 3D objects such as triangle, plane, sphere, cylinder, cone, etc. will also be proposed, designed and developed.

This constitutes a unique and significant contribution to the disciplines of web-based 3D modelling, as well as to the process of 3D model popularization.

Signed .... [redacted] .... Date ...20/02/2015......

ACKNOWLEDGEMENTS

I would like to thank all the people who helped me build the foundations for the SVG 3D framework for web-based application during my Ph.D. studies. Special thanks go to my supervisor Dr. Shujun Zhang. His advice and guidance have been invaluable, his enthusiasm and scientific knowledge both motivating and inspiring. I had a great time at the School of Computing and Technology, and I hope that our cooperation will continue in the future.

Many thanks also to my co-advisors Dr. Vicky Bush. It has been a great experience to work with her and share her knowledge in many interesting discussions.

I would like to thank my Family: my parents, my wife, my son, for their understanding, support and motivation, which they offered me every day throughout the last four years. This work would not have been possible without your help.

Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Background

Computer graphics includes the process and outcomes associated with using computer technology to convert created or collected data into visual representations. Computer graphics has grown phenomenally in recent decades, progressing from simple two-dimensional (2D) graphics to complex, high-quality, three-dimensional (3D) environments. In entertainment, computer graphics is used extensively in movies and computer games (Parent, 2012). Animated movies are increasingly being made entirely with computers. There are also significant uses of computer graphics in non entertainment applications. For example, virtual reality systems are often used in training (Wong, 2005). Computer graphics is also an indispensable tool for scientific visualization and for computer-aided design (CAD) (Delmarcelle, 1993). The computer graphics field is motivated by the general need for interactive graphical user interfaces that support mouse, windows and widget functions. Other sources of inspiration include digital media technologies, scientific visualization, virtual reality, arts and entertainment.

Over the past years, advances in display and computing technology have revolutionized the visualization of computer graphics. Now people can interact with rich, realistic, 3D graphics using relatively low cost equipment.

Interactive 3D graphics provides the capability to produce moving pictures or animation. This is especially useful when exploring time varying phenomena such as weather changes in the atmosphere, the deflection of an airplane wing in flight, or telecommunications usage patterns. Interaction provides individual users the ability to control parameters like the speed of animations and the geometric relationship between the objects in a scene to one another.

Today, interactive 3D graphics are used in a wide variety of fields. They can be used to model organs in a medical simulation system (Marescaux, 1998). Entertainment fields such as movies or video games use 3D models to produce photo-realistic 3D scenes and characters (Foskey, 2002). They can also be used to demonstrate complex data or molecular structure (Kumar, 2008) in the science sector. The architecture industry uses 3D models to demonstrate proposed buildings and landscapes through Software Architectural Models (Gross, 2005). In engineering, 3D models have been widely used for product design and development (Gobithasan, 2005).

The emergence of the Internet and World-Wide Web (WWW) provides a flexible means for linking applications, data, information, and users. To seamlessly interlink associated data and couple visual representations with this data creates opportunity for new approaches to visualization. The term *web-based application* is used to describe applications that use the Internet and WWW as an information source, a delivery mechanism for applications, or both. A web-based application is any application software that runs in a web browser or is created in a browser-supported programming language and relies on a common web browser to render the application (Gaedke, 2000).

The web is a powerful tool since it provides communication around the world. The web has become popular for efficient communications in all sectors, i.e. education, business and government. Web-based applications should be seriously considered as the ideal mode of communication because it provides the method for sharing information in a fast, cost effective manner. Incorporation of 3D graphics into a web application is needed to provide a model and simulation that incorporates the desired 3D model. This is now realizable with the advancements in computer technology.

Web-based applications are rapidly entering domains where window-based applications have previously been the only viable alternative. Web-based applications present some advantages over window-based applications. Firstly, it is not necessary to install the dedicated software; the user only has to access a web page through his preferred web browser. Therefore, with a web application, it can be ensured that the user is using the latest version of the software, since the update process is done in the server instead of in each client machine, as happens with window-based applications. Secondly, the cross-platform character of the web-based applications can attract extra users, since they are operating-system independent.

Interactive 3D graphics is one of the largest areas where web-based applications have not seen much success until recently (Jiménez, 2013). Part of this has been because this type of interactivity demands much of the underlying device and software, such as the virtual machines for scripting languages so often involved in these types of applications. Another reason is the lack of a 3D graphical application programming interface (API) for most web application technologies.

In this case, there are some new approaches for delivering 3D graphics in a web-based application. (El-Khalili, 2005) used Virtual Reality Modelling Language (VRML) and Java-3D to present a prototype for 3D computer-aided learning tools of the human anatomy. (MacEachren, 1998) used VRML 2.0 to model geospatial data, and Java to develop an interface to interact with the VRML world. (Hibbard, 1998) designed and developed the VisAD system, which enables many users to implement the visualization of a shared set of numerical data and computation sources. VRML has been applied for simulations of engineering system design such as submarine design and workshop layout design. But these applications are window-based. If VRML is used in a web-based application, a plug-in is required. And the 3D model file size generated by VRML is normally bigger than 1 MB (Mega byte), and will affect the performance of the web-based application.

X3D (Brutzman, 2007) is the successor of VRML 2.0, and it has been the International Standard Organization (ISO) open standard for 3D web content delivery in 2005. X3D combines geometry and runtime behavioural description into a single file. X3D can be integrated into web services, and its API-Scene Access Interface (SAI) allows any JavaScript, Java or C/C++ based applications to communicate with X3D through this API. (Rahman, 2007) discussed the dynamic visualization of 3D spatial data such as buildings and other large objects using geo-data base management system (DBMS) coupled with web-based system that works with VRML and X3D. As with VRML, X3D also need plug-in for web-based applications. Similarly to VRML, the 3D model file size generated by X3D is normally bigger than 1 MB (Mega byte), and will affect the performance of the application created by X3D.

O3D (Ortiz, 2010) is Google's open source project, featuring a plug-in and a JavaScript API for creating 3D graphics applications that run in a web browser. O3D uses Open Graphics Library (OpenGL) and DirectX for graphic rendering. The system consists of two layers. The first layer – the plug-in, provides a shader and geometry abstraction mapped to OpenGL and DirectX. The higher second layer provides a JavaScript API similar to Open Scene Graph (OpenSG) or Java3D. 3D Markup Language for Web (3DMLW) (Turonova, 2009) is a file format based on eXtensible Markup Language (XML) developed by 3D Technologies R&D. It is designed for creating and representing both 3D and 2D interactive content on the internet. A 3DMLW document is written in a markup-based language similar to eXtensible HyperText Markup Language (XHTML). 3DMLW supports key-frame animation, Bezier-splines, particle systems, physics and collisions, all defined in a declarative manner as XML elements. It uses OpenGL for graphics rendering and Open Audio Library (OpenAL) for audio processing. This technology is similar to X3D, because it also encodes its contents in a XML-based file format, supports scripting to enhance interactivity, and needs a web browser plug-in to render the contents within the file.

However, 3D graphics on the web remains primitive today because the complex technology has been difficult to use with typical PCs and browsers. In fact, browsers generally cannot natively run complex 3D content or offer either high frame rates or full-screen graphics (Ortiz, 2010).

1.2 Motivation

Today, 3D graphics is primarily used in applications such as games and virtual reality, which are rendered using powerful computers and specialized software. However, businesses, engineering firms, and other users also want the realism and additional details that 3D adds. Users want their browser-based experiences to be more like those they have on a desktop. Consumers are becoming more accustomed to 3D contents because of the use of the technology in movies, videogames, and other types of entertainments.

There is thus a demand for more and easier access to 3D contents on the web. The better the browser experiences, the more potential revenues that web-based contents can generate for providers and others.

Now several organizations are working on technologies that may finally widen 3D's presence on the web by transforming browsers into more powerful computing platforms that can deliver a PC-like experience, including the playing of 3D content (Skarler, 2009). This would enable applications such as product modelling, presentation, and configuration; 3D web-based meetings and worker collaboration; the simulation of processes such as surgery or mechanical procedures; virtual tours; and augmented reality. Nonetheless, 3D on the web will have to clear some obstacles before the technology can become reliable and mainstream.

A key motivation for the use of 3D models is that humans think graphically with an innate ability to process graphical information resulting in fast and effective communication. Furthermore, human information processing of graphical information is involuntary and automatic, leaving more conscious problem solving abilities available.

Another motivation behind using 3D modelling is for sensory appeal and immersion into the model. This results in enjoyment and ultimately increased understanding of the model and what it represents.

The criteria for the evaluation the effectiveness of a web-based 3D model differ between each application. One of the key properties of a web-based 3D model is that to make it accessible to the "normal" user with only standard computer equipment (2D monitor, mouse, keyboard, sound support). The key criteria to evaluate the effectiveness of a web-based 3D model are availability and accessibility (Yoon, 2008). So the follows criteria should be adopted in this thesis:

1. Functionality, it can be used to build, modify and view 3D geometrical models of all primitive geometries and free form surfaces for a variety of web-based applications. This means the technique is available for develop web-based 3D model.

2. Applicability, it has native support from the majority of web browsers and can be viewed on a web browser without any plug-ins. The 3D model developed by this technique is accessible from web browser without any plug-ins.

3. Efficiency, it can be used to create a 3D model with the smaller file size than the model created for a window-based application.

As discussed above, although some successful web-based 3D modelling techniques, such as VRML, X3D, and O3D have been reported in the literature, based on the above criteria, they all have drawbacks. Most techniques have no native browser support and are only available through third party plug-in. Besides, the file sizes of some 3D models built using the existing methods are very big, which leads to the upload time and transaction time is much longer than a user expects. In summary, in one aspect, there are a lot of requirements of web-based 3D models and in the other aspect, there are no existing 3D modelling methods that are suitable to the practical web-based 3D applications,

SVG (Scalable Vector Graphics) is an XML-based markup language for describing 2D graphics applications and images, and a set of related graphics script interfaces (Spanaki, 2004). SVG graphics can be interactive and animated. Bindings for scripting languages and network interfaces enable developers to build rich interactive graphics applications. SVG is developed by W3C for describing 2D vector graphics for storage, presentation and distribution on the Web. All major modern web browsers—including Mozilla Firefox, Internet Explorer 9 and 10, Google Chrome, Opera, and Safari—support basic SVG and can render the markup directly without the use of a plug-in (Peterson, 2012). This includes support for fonts, images, graphical elements such as circles or paths, as well as gradients and some of the filters. There are several advantages to native and full support: a plug-in is not needed, SVG can be freely mixed with other content in a single document, and rendering and scripting become considerably more reliable.

SVG provides an open, standard based format for creating graphics. Using SVG has numerous advantages over other conventional bitmapped graphics, such as JPEG, GIF, and PNG. The files are generally much smaller than bitmaps, resulting in quicker download times. The graphics can be scaled to fit different display devices without the pixelation associated with enlarging bitmaps. The graphics are constructed within the browser, reducing the server load and network response time generally associated with web imagery. The file is an XML-based file format; it allows the creator to conveniently embed arbitrary information inside of the file. SVG is well suited for graphics rich environments. SVG can be used for GIS, embedded systems, location-based services, animated picture messaging, multimedia messaging, animation and interactive graphics, entertainment, e-Business, and graphic user interfaces.

Based on the above analysis, it can be seen that SVG meet almost of all the criteria except that the existing SVG is only in 2D. However, if 3D is integrated into SVG, it can even play an important role in many fields such as product demonstration, city planning, site exhibition, 3D e-Business (such as 3D shopping malls), etc. So it is practically important required to further study the feasible methods for 3D modelling for web-based applications. Therefore, this project will be focused on proposing, designing and developing a generic method for building SVG 3D models for web-based applications.

1.3 Aim and Objective

The context of this PhD project is based on two assumptions: firstly, the rapid developments in the field of computer graphics and computer hardware now allows for real time visualization of complex 3D data over the internet; secondly, digital 3D models are playing an increasingly important role in many fields such as product demonstration, city planning, site exhibition, 3D online gaming, decision-making and 3D e-Business (such as 3D shopping malls), etc.

The overall aim of this thesis is to propose, design and develop a new generic framework for efficient SVG 3D modelling for various interactive manipulation web-based applications.

The main focus of this thesis is on the design, develop and implementation of a new 3D generic modelling method based on Scalable Vector Graphics (SVG) for web-based applications.

In order to achieve this overall aim, the objectives of this project are:

1. To investigate and evaluate existing methods for 3D modelling for web-based applications by a systematic literature review.

2. To evaluate the existing 2D SVG applications and to investigate the possibility of using SVG to realize 3D graphical representations for various web-based applications.

3. To analyze the geometric structure and features of 3D models and propose a new generic framework for 3D modelling for web-based applications.

4. To design and develop the dedicated framework- SVG Graphics Library (SVG GL) for 3D models creation, interactive manipulation and view in web browser.

5. To research, design and develop new algorithms for shading and texture mapping.

6. To design and develop a software environment for implementing and validating the proposed framework and algorithms.

7. To validate the proposed framework and algorithms through 4 web-based applications.

8. To evaluate the proposed framework and algorithms through 4 typical web-based applications.

1.4 Contributions to New Knowledge Generation

The primary contribution of this project will be its proposition, design and development of a new generic framework for modelling and constructing SVG-based 3D models for efficient web-based applications. This framework can be applied widely in interactive manipulation web-based environments.

The main contributions of this PhD project are:

1. Propose, design and development of a new framework-SVG GL for SVG 3D modelling based on classical 3D graphic theory and SVG. While the model is initialized using classical 3D graphics, the scene model is extended using SVG. A new algorithm to present 3D graphics with SVG is proposed. This includes the

definition of a 3D scene in the framework, integration of 3D objects, cameras, transformations, light models and textures in a 3D scene, and the rendering of 3D objects on the web page, allowing the end-user to interactively manipulate objects on the web page.

2. Design and develop a new 3D graphics library for 3D geometric transformation, and projection in the SVG GL.

3. Design and develop a set of primitives in the SVG GL, including triangle, sphere, cylinder, cone, etc..

4. Design and develop a set of complex 3D models in the SVG GL, including extrusion, revolution, Bezier surface, and point clouds.

5. Propose, design and develop the new Gouraud shading algorithm and new Phong Shading algorithm in the SVG GL. The algorithms can be used to generate smooth shading and create highlight for 3D models.

6. Propose, design and develop the new texture mapping algorithm for the SVG GL oriented toward web-based 3D modelling applications. Texture mapping algorithms for different 3D objects such as triangle, plane, sphere, cylinder, cone, etc. will also be proposed, designed and developed.

This constitutes a unique and significant contribution – both theoretical and practical – to the disciplines of web-based 3D modelling, as well as to the process of 3D model popularization.

1.5 Thesis Structure

The thesis is arranged as follows.

Chapter 1, “Introduction”, presents the background of this PhD project, motivations, and overall aim and objectives.

Chapter 2, “Literature Review and Current 3D Technologies”, first reviews the research literature on the state of the art of 3D web-based modelling, and how this relates to other branches of 3D modelling. The technologies for window-based 3D presentation are introduced. Most importantly, current technologies for web-based 3D presentations are discussed, weighting up the advantages and drawbacks of each technology.

Chapter 3, “Research Design and Methods”, provides the research design of this project, and also discusses the methods for data collection and presentation.

Chapter 4, “SVG Theory and Its Applications, and OpenGL”, is focused on SVG. The SVG Theory and its applications are reviewed, the problem with the existing SVG 2D are also discussed. The need for 3D SVG research is documented.

Chapter 5, “A New Framework-SVG GL for Web-Based Graphical Presentation”, proposes the new framework-SVG GL. The framework is described from the functional perspective, as well as from the technical design angle.

Chapter 6, “New Algorithms for Shading in the SVG GL”, is oriented more specifically on the illumination and shading, and develops new Gouraud shading algorithm and new Phong shading algorithm in the SVG GL.

Chapter 7, “New Algorithms for Texture Mapping in the SVG GL”, is oriented more specifically on the texture mapping. New texture mapping algorithms developed for the SVG GL are presented in details.

Chapter 8, “Design and Development of the Software Environment for Validating the Proposed Framework and Algorithms”, presents the analysis of the system requirements, the system design and development environment of the software environment of the SVG GL.

Chapter 9, “The Discussions of the Proposed Methods for 3D Web-Based Presentations”, describes 4 demo applications of interactive 3D model based on the SVG GL, and discusses the potential application fields of the SVG GL.

Chapter 10, “Conclusions and Further Work”, concludes the thesis with a summary of research outcomes, as well as a discussion of its original contributions and of future works.

Chapter 2 Literature Review and Current 3D Technologies

2.1 Introduction

3D computer graphics is the science, study, and method of projecting a 3D representation of geometric data that is stored in the computer onto a 2D image using visual tricks such as perspective and shading to simulate the eye's perception of those objects. 3D computer graphics represent a 3D object using a collection of points in 3D space, connected by various geometric entities such as triangles, lines, curved surfaces, etc. 3D computer graphics are often referred to as 3D models. 3D modelling is the process of developing a mathematical, wireframe representation of any 3D object via specialized software (Watt, 1999). A model is not technically a graphic until it is displayed.

The development of 3D computer graphics has been driven both by the needs of the user community and by advances in hardware and software. Due to the progress in display and computing technology, now people can interact with rich, realistic, 3D computer graphics with relatively low cost equipment.

The applications of 3D computer graphics are many and varied. A major use of 3D computer graphics is in design processes, particularly for engineering and architectural systems, almost all products are now computer designed (Figure 2.1). Generally referred to as CAD, computer-aided design methods are now routinely used in the design of buildings, automobiles, aircraft, watercraft, spacecraft, computers, textiles, and many other products (Groover, 1983; Bliss, 2002; Gobithasan, 2005).

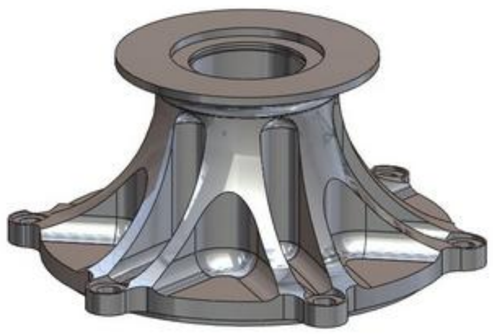


Figure 2.1 A fuel pump mount model designed by 3D CAD (free model download from http://3dprinterbaski.com/cadcam-sistemlerinin-genel-yapisi)

3D Computer graphics is also widely used in scientific visualization (Delmarcelle, 1993; Grissom, 1995; Bertoline, 1998). Science and engineering, and even certain aspects of mathematical and statistical analysis, involve the documentation of variation of one quantity against another, either predicted according to some law, or measured from some experiment, or gathered from some poll. Scientific visualization is primarily concerned with the visualization of 3D phenomena (architectural, meteorological, medical, biological, etc.), where the emphasis is on realistic renderings of volumes, surfaces, illumination sources (Figure 2.2).

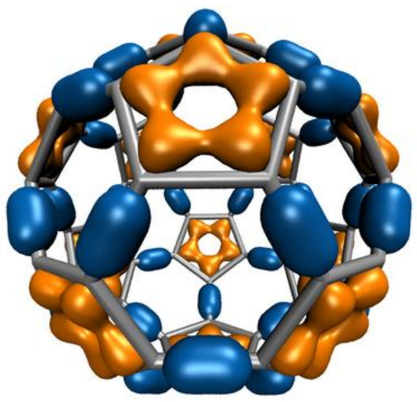


Figure 2.2 Molecular orbital for a Carbon-60 molecule (free model download from http://www.ks.uiuc.edu/Research/vmd/vmd-1.8.7/cuda.html)

3D Computer graphics is now commonly used in entertainment (Machover, 1998, Sumner, 2008; Parent, 2012). Computer games, with colourful and animated screen displays, were among the first application of 3D computer graphics. Cartoon animation was a logical extension of these ideas. Cartoons are often rendered directly from 3D models (Figure 2.3). Many traditional 2D cartoons use backgrounds rendered from 3D models, which allows a continuously moving viewpoint without huge amounts of artist time.

Figure 2.3 3D cartoon characters (free model download from http://gallerycartoon.blogspot.co.uk/2014/05/ice-age-3-3d-cartoon-pictures.html)

3D Computer graphics can also be used in other areas, they can be used to model organs in a medical simulation system (Marescaux, 1998; Wong, 2005). The architecture industry uses 3D models to demonstrate proposed buildings and landscapes through Software Architectural Models (Gross, 2005).

2.2 3D Computer Graphical Presentations

A 3D computer graphics system can be thought of as having 2 major components, each of which performs a distinct and clearly defined key role in the process of image presentation. These two components are responsible for 3D scenes modelling and 3D rendering. Figure 2.4 gives a schematic view of the process used in 3D computer graphics, showing the role that each of those components plays. Each of these major components can be broken down into groups of important subcomponents.

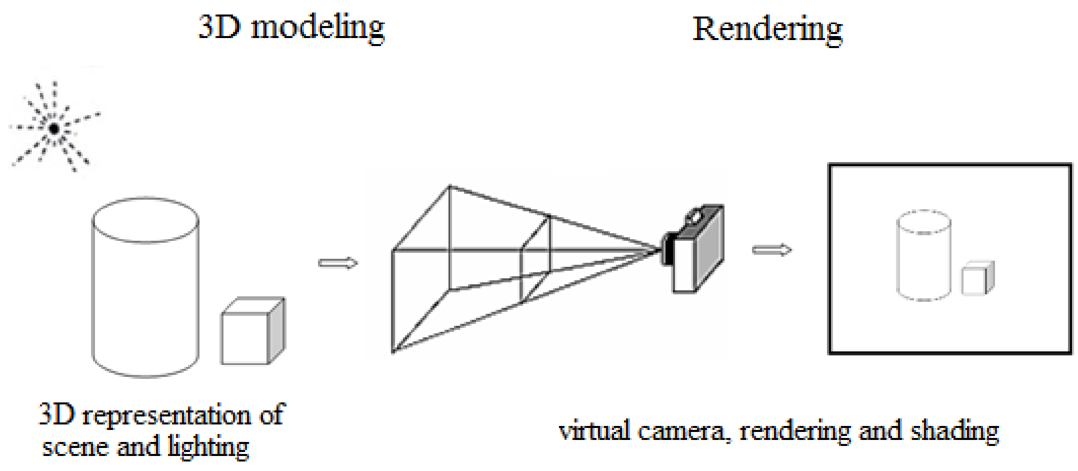

3D modeling
Rendering
3D representation of scene and lighting
virtual camera, rendering and shading

Figure 2.4 The 3D computer graphics presentation

2.2.1 3D Scene Modelling

The 3D Scene Modelling in 3D computer graphics is responsible for providing an internal mathematical representation of any 3D object that is eventually to be imaged. The 3D Scene Modelling system needs to support some concept of a geometric coordinated system and provide some way of describing the geometry of the 3D object to be imaged. A 3D modelling system will also provide a way for the user to specify what materials an object is made of and how the scene is lit.

1. Coordinate Systems

The key to the geometry of a 3D computer graphics system is a compact means for storing and utilizing descriptions of local coordinate systems. The local coordinate system is used in the definition of the various components of a model, describing the geometry and other characteristics of the scene.

Consistent with the usual representation of 3D coordinates in mathematics, most current implementation of 3D computer graphics systems use right-handed coordinate system (Foley, 2013). This gives a natural organization with respect to the display screen, with the *x*-coordinate measuring horizontal distance across the screen, the *y*-coordinate measuring vertical distance up the screen, and *z*-coordinate proving the third spatial dimension as distance in front of the screen (Figure 2.5).

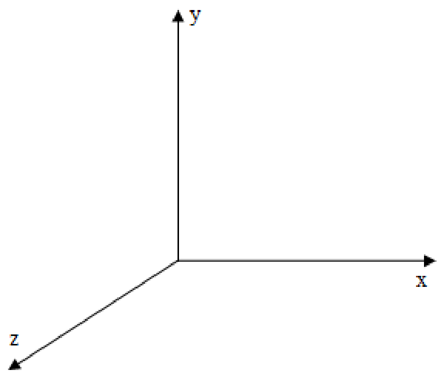


Figure 2.5 Right-handed coordinate system

2. Geometric Modelling

The basic geometric unit in 3D computer graphics system is the 3D point that is typically represented as a 3D-vector and stored as an array of three elements, representing the *x*, *y*, and *z* components of the point.

Virtually all 3D computer graphics systems provide the ability to work with simple geometric primitives that can be specified as lists of 3D points. These primitives include point, lines, and polygons. Points can be arranged together to indicate a sampled surface, lines to form a wireframe representation, and polygons to form polyhedral surfaces. More sophisticated modellers will provide parametric surfaces, which are defined via an underlying piecewise polynomial formulation. Polynomial coefficients are adjusted to give the surface a specific shape, and these coefficients are often given intuitive form by encoding them via simple geometric devices, such as control polyhedral.

A typical surface formulation is a biparametric surface, which describes a surface in three spatial dimensions (*x*, *y*, *z*) via a set of three functions of two parameters *u* and *v*:

$$x = \mathrm{X}(u, v), y = \mathrm{Y}(u, v), z = \mathrm{Z}(u, v) \qquad (2.1)$$

A set of points on a parametric surface can be obtained algorithmically by looping over a collection of sample points on the (u, v) plane.

Implicit surfaces are a common alternative to parametric surfaces. Here, surfaces are defined as the set of points satisfying a mathematical expression of the form.

$$\mathrm{F}(x, y, z) = 0 \qquad (2.2)$$

Thus, these surfaces are defined implicitly. Any point (x, y, z) in 3D space can be tested to determine whether or not it is above ($\mathrm{F}(x, y, z) > 0$), below ($\mathrm{F}(x, y, z) < 0$), or on the surface ($\mathrm{F}(x, y, z) = 0$).

3. Materials

In the context of a 3D computer graphics system, a material is an attribute of a geometric object that provides a description of how the surface of the object will appear when viewed from a particular direction under a particular illumination.

A usual material specification system will provide parameters for the specification of a material's colour, diffuse reflectance factor, and specular reflectance factor. From the point of view of usual practice, colour in 3D computer graphics is most often represented by RGB or "red-green-blue" colour system (Figure 2.6). An RGB colour is stored as a triple of three numbers giving the relative amount of each of the three colours primaries.

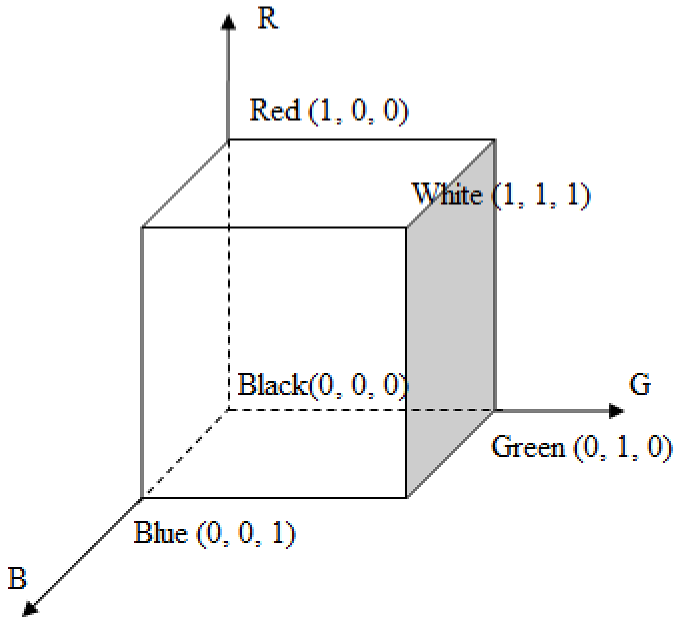


Figure 2.6 RGB colour cube

A material specification will also include the capability to provide texture maps. A texture map provides a pattern of colour that is to be applied to the surface of an object during the rendering process. These can be anything from a digital image that will be projected onto the surface to a regular geometric pattern like a checker-board.

4. Lights

The purpose of lights in a 3D computer graphics system is to provide the illumination source for the simulated shading calculations done by the renderer in making an image. Thus, all light sources must define a colour of the illumination that they provide, usually specified in RGB coordinate. The illumination colour combines the intensity of the light and its chromatic attributes. Lights are arranged in a scene along with geometric objects but usually carry no geometric properties other than their position and direction of orientation.

2.2.2 3D Rendering

Rendering is simply the process of transforming a 3D object description into a 2D image. It is generally done by simulation of the physical process that occurs in a camera when a picture is recorded on film.

Briefly, the main steps in the rendering process are (Tucker, 2004):

Step 1. Point of view: orienting the 3D scene as if it was being viewed from a particular point in space.

Step 2. Projection: associating points in the 3D scene with their images on a 2D virtual image plane by projecting the 3D scene onto the plane.

Step 3. Visible surface determination: deciding which surfaces projected onto the image plane would actually be visible from the present viewpoint.

Step 4. Shading calculation: determining what colour would be reflected or transmitted to the viewpoint from the geometry visible at the sample point, taking into account the scene’s geometry, lighting, and material.

1. Virtual Camera

The role of the virtual camera in a 3D computer graphics system is to provide both a point of view from which to render an image and the basic parameters of the mathematical projection that will be used to form the virtual image. The camera’s position and orientation are specified as part of the scene description. It is typical for the camera to be positioned in the global coordinate system, usually with some positioning controls that correspond to the operation of a real studio camera.

Theoretically, cameras can have any projection characteristics, corresponding to the entire variety of lens type. However, practical 3D graphics implementations usually implement only the standard parallel or perspective projections that are common in architectural and design drafting.

A perspective projection is one in which all light rays coming from the scene converge at a common point, known as the centre of the projection. If a projection plane is interposed between the scene and the centre of projection, the point at which a ray from the scene through the centre of the projection intersects the projection plane is the image of that point (Figure 2.7).

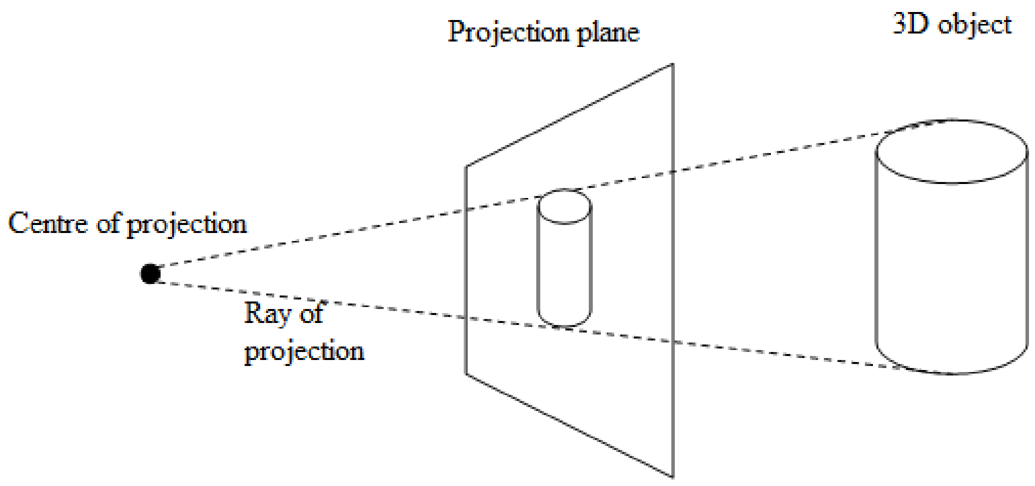


Figure 2.7 Geometry of perspective projection

2. Renderer

The renderer in a 3D computer graphics system is essentially the engine that drives the picture-making process. The renderer views the 3D scene through the virtual camera and constructing an image of what it sees, by first sampling points on the scene geometry and calling on the shader to calculate colour for each sample, and then combining these sampled colours into the pixels of the image.

3. Shader

The shader is the algorithm that uses the information collected by the renderer about a point sample on the scene geometry, its material, and the available lighting to calculate a colour for the sample point. This is done by a physical simulation of how light is reflected toward the camera from the position on the surface at which the sample is being taken.

This section has examined the procedure of 3D computer graphics presentation. The next section will focus on the technologies used for 3D graphical presentations.

2.3 Technologies for 3D Graphical Presentations

3D computer graphics are created with the aid of digital computers and specialized 3D software (Hees, 2006). In general, the term may also refer to the process of creating such graphics, or the field of study of 3D computer graphic techniques and their related technology.

Developing 3D computer graphics requires a programming library which provides an API for 3D graphics. These libraries can be classified into two categories. The first group of libraries provides what is commonly called *immediate mode* rendering where the developer tells the library what 3D models should be drawn, and how each of them should be transformed, each time the scene is to be drawn. The other category provides *retained mode* rendering, a type of rendering where the developer constructs a scene using abstract data types provided by the library and then the library traverses the scene and renders each 3D model on the screen. The main differences between these two categories are who is in charge of the rendering process and who owns the properties (such as transformation) of each 3D model within the scene.

The rendering process can be performed on the CPU and is then called software rendering. Until recently, the most common types of CPUs can only run one process at a time. Because rendering 3D graphics is a process that is both computationally intensive and easy to parallelized, a Graphics processing unit (GPU) is commonly used to perform the task of rendering instead of the CPU. When rendering is done on a GPU, it is called hardware-accelerated rendering. Hardware-accelerated rendering is generally much faster than software rendering since it offloads the CPU to run the program at hand and the GPU is built to process many vertices and fragments in parallel.

Two of the most popular libraries for 3D graphics programming are OpenGL and Direct3D. OpenGL is an open standard managed by the non-profit technology consortium Khronos Group (Patric, 2012) and is available on many devices, from desktop computers and workstations to game consoles and mobile phones. Direct3D (Michael, 1997) is a library created by Microsoft as one of the APIs that make up the DirectX suite of multimedia programming libraries available exclusively on Microsoft's own platforms, such as the Windows operating system and the Xbox gaming consoles. Both of these libraries are immediate mode libraries and utilize hardware-acceleration.

2.3.1 OpenGL

Open Graphics Library (OpenGL) is a cross-language, multi-platform API for rendering 2D and 3D computer graphics (Shreiner, 2009; Sellers, 2010). The API is typically used to interact with a Graphics Processing Unit (GPU), to achieve hardware-accelerated rendering. In addition to being language-independent, OpenGL is also platform-independent.

OpenGL was developed by Silicon Graphics Inc (SGI) from 1991 and released in January 1992 and is widely used in CAD, virtual reality, scientific visualization, information visualization, flight simulation, and video games. OpenGL is managed by the Khronos Group currently.

The OpenGL specification describes an abstract API for drawing 2D and 3D graphics. Although it is possible for the API to be implemented entirely in software, it is designed to be implemented mostly or entirely in hardware.

OpenGL has many language bindings, some of the most noteworthy being the JavaScript binding WebGL (API, based on OpenGL ES 2.0, for 3D rendering from within a web browser); the C bindings WGL, GLX and CGL; the C binding provided by iOS; and the Java and C bindings provided by Android (Sayed, 2010).

2.3.2 Direct3D

Direct3D (Jones, 2004; Zink, 2011) is part of Microsoft's DirectX API. Direct3D is available for Microsoft Windows operating systems (Windows 95 and above), and for other platforms through the open source software Wine. It is the base for the graphics API on the Xbox and Xbox 360 console systems. Direct3D is used to render 3D graphics in applications where performance is important, such as computer games. Direct3D also allows applications to run full screen instead of embedded in a window, though they can still run in a window if programmed for that feature. Direct3D uses hardware acceleration if it is available on the graphics card, allowing for hardware acceleration of the entire 3D rendering pipeline or even only partial acceleration. Direct3D exposes the advanced graphics capabilities of 3D graphics hardware, including z-buffering, spatial anti-aliasing, alpha blending, atmospheric effects, and perspective-correct texture mapping. Integration with other DirectX technologies enables Direct3D to deliver such features as video mapping, hardware 3D rendering in 2D overlay planes, and even sprites, providing the use of 2D and 3D graphics in interactive media ties.

Direct3D is a 3D API. That is, it contains many commands for 3D rendering; however, since version 8, Direct3D has superseded the old DirectDraw framework and also taken responsibility for the rendering of 2D graphics (Michael, 1997).

OpenGL is an open standard API that provides a number of functions to render 2D and 3D graphics, and is available on most modern operating systems including but not limited to Windows, Mac OS X and Linux. Direct3D is a proprietary API by Microsoft that provides functions to render 2D and three-dimensional 3D graphics, and uses hardware acceleration if available on the graphics card. It was designed by Microsoft Corporation for use on the Windows platform. Direct3D can also be used on other operating systems through special software (emulator).

2.3.3 Other Technologies

There are also other higher-level 3D scene graph technologies which provide additional functionality on top of the lower-level rendering API.

1. Java 3D

Java 3D (Selman, 2002) officially released in 1998, is a cross platform API that enables the development of 3D graphics applications using the popular Java programming language. It is considered the 3D extension for Java. It allows developers to create complex and interactive 3D desktop applications, or web based 3D applets, that can work efficiently on multiple platforms.

Java3D is a scene graph based 3D API for the Java platform. It takes advantage of OpenGL or Direct3D. Java 3D allows the programmer to specify how the 3D scene is structured rather than providing functions for drawing 3D graphics directly. Java 3D can use Direct3D or OpenGL on Windows system, and use OpenGL on the other supported platforms, such as Mac OS X, Linux and Solaris

Creating 3D desktop application in Java3D can be a relatively simple process for Java programmers. However, creating Java3D applets to embed 3D contents within a webpage can become a more complex task. This requires some skills in

HTML programming, as well as JavaScript programming to enhance the interactivity with the 3D web applets (Salisbury, 1999)

2. Glide API

Glide (Mitra, 1999) is a 3D graphics API developed by 3Dfx Interactive for their Voodoo Graphics 3D accelerator cards. Although it originally started as a proprietary API, it was later open sourced by 3Dfx. It was dedicated to gaming performance, supporting geometry and texture mapping primarily, in data formats identical to those used internally in their cards. Wide adoption of 3Dfx led to Glide being extensively used in the late 1990s, but further refinement of Microsoft's Direct3D and the appearance of full OpenGL implementations from other graphics card vendors, in addition to growing diversity in 3D hardware, eventually caused it to become superfluous.

Glide is based on the basic geometry and 'world view' of OpenGL. The result was an API that was small enough to be implemented entirely in late-1990s hardware. However, this focus led to various limitations in Glide, such as a 16-bit colour depth limit in the display buffer.

The technologies and APIs used in developing 3D graphical systems have been examined in this section. The next section will focus on the challenges posed by presenting such systems via the web.

2.4 3D Computer Graphical Presentations for Web-Based Applications

Due to the rapid developments in the field of computer graphics and computer hardware, web-based applications are becoming more and more powerful, and the performance distance between web-based applications and desktop applications is increasingly closer. The Internet and the WWW have been widely used for delivering, processing, and publishing 3D data. In recent years, web-based 3D models for visualizing data have attracted many researchers.

Including 3D computer graphics on web pages is not a new trend. In 1994, a way to present 3D scenes in the web browser through a markup language called VRML was standardized. VRML allows 3D scenes to be specified in a language similar to HTML. While there are niche applications that use VRML, there are few popular sites today that include VRML documents. It seems to have become a standard that never really grew popular enough to see wide-scale usage for a number of reasons.

Chief among these reasons is likely that the processing power available, first and foremost in reasonably priced computers, back in 1994 was not good enough to provide convincing 3D graphics. This has changed recently with dedicated graphics processors finding their way into more and more types of devices. Thanks to this, there has been a renewed interest in technologies that provide 3D computer graphics on web pages and in web applications. Different approaches have been developed for web-based 3D modelling, as described below. Many tools are available to use the web as a delivery mechanism, and deals with the transformation of multi-dimensional data, information, and knowledge into an effective 3D visual form.

2.5 Technologies for Web-Based 3D Graphical Presentations

There are many approaches to delivering 3D graphics in a web-based application. Technologies available today will be described and discussed in this section. It will be discussed how the concept of web-based 3D evolved, from the creation of VRML to the newest WebGL.

2.5.1 VRML

Created in 1994, by the VRML Consortium, this high level 3D content development language was responsible for introducing the concept of 3D graphics for web, and was the first ISO standard for the creation and visualization of 3D contents on the Internet (Walsh, 2000).

The VRML 1.0, officially released in 1995, was proposed as a common language for the creation of 3D scenes distributed over the Internet. For that, it was created with the intent of being a cross platform, extensible, and bandwidth conservative language. It can be said that VRML 1.0 brought the platform-independent 3D concept for the web. However, this release was very limited because it only allowed to create non-realistic and static 3D scenes, and was not possible to interact with the 3D objects within that scene.

In order to provide a more immersive, realistic and interactive 3D world, a second major version of VRML was released, the VRML 2.0 (defined later as VRML97). This release brought support for interactivity, sound, animation, and ultimately the ability to create more complex 3D worlds.

In order to display, interact, and navigate on the VRML 3D world described in the .wrl file, a VRML web browser plug-in or a standalone player have to be used to interpret this file. VRML allows the creation of full 3D environments, but it has some limitations. For example, it does not allow for video streaming, binary compression, and multi-texturing.

The advantage of VRML is that it is widely platform-independent, easy to create by exporting from standard 3D-Graphics Software, and it works well with the newer browser generations; VRML can be visualized efficiently on standard PCs without the need to purchase additional custom hardware; VRML is a very simple yet powerful language that can be learnt quite easily, hence it enables developers to create new VRML worlds or enhance existing ones without technical knowledge of 3D visualization

In addition to the above advantages, VRML also have significant disadvantages. The flexibility of VRML made it difficult to write rendering engines that were fast. In addition, interfaces with web browsers, i.e. the HTML page in which the 3D scene is embedded were unreliable and not standardized. This was also a major flaw, since it is important to be able to combine interactive 2D and 3D contents.

2.5.2 X3D

The technology X3D developed by the Web3D consortium as the third generation of VRML, and became an ISO standard in 2004. It was developed with the main goal of overcoming the deficiencies of VRML, and to make the creation of 3D graphics an easier and more intuitive task, accessible to a wide range of developers, including 3D graphics programmers and even non-programmers.

Like its predecessor VRML, X3D is a standard for real-time interactive visualizations based on a markup language. X3D uses a tree-structured scene graph to represent the graphics nodes that make part of the 3D world. This scene graph includes the geometry, appearance, animation and event routing.

Just like VRML, X3D needs a player to parse and render an encoded X3D scene, which may also allows for user interaction and object animation. In fact, every browser needs an X3D player plug-in in order to render X3D scenes.

The advantages of X3D for software visualization are rich graphics, extensibility, and XML integration. The disadvantages of X3D are lack of software visualization user controls, a primitive animation model, and weak support for filtering and layout (Craig, 2008).

2.5.3 WebGL

WebGL (Danchilla, 2012) is a relatively new cross-platform JavaScript API developed by the Khronos Group that extends the capability of the classic JavaScript programming language, allowing the generation of native 3D graphics in any compatible web browser, without needing extra plug-in. The WebGL API is based on the OpenGL ES 2.0 standard, so it enables a direct access to each GPU (Graphic Processing Unit) located on the client. It uses the HTML5 canvas element and is accessed using Document Object Model (DOM) interface.

Firefox, Chrome and Opera support WebGL by default on Windows, Mac OS X and Linux. Safari also supports WebGL on Mac OS X, but it has to be enabled manually. Internet Explorer does not support WebGL without the use of a plug-in. Both Firefox and Opera provides support for WebGL. WebGL even already runs on several mobile devices, including the iPhone. This means that, for all these browsers, no plug-in has to be installed to run web applications using WebGL. However, relatively new graphics hardware and drivers are required on clients' computers.

The advantage of WebGL is that WebGL is not based on a plug-in. It runs directly in the browser, and is a public standard. A WebGL application can be developed without leaving the familiar web development environment of HTML and JavaScript; all calls to the graphics API are made in JavaScript. There is no official development environment; any JavaScript development environment and debugger can be used.

However WebGL needs a GPU support for shader rendering to be supported and viewable by the user. The performance of WebGL is also limited by the dynamic nature of JavaScript. The current browsers do a great job of optimizing this already, but because of how JavaScript is designed, it won't get much faster anymore. WebGL is low level; it is complicated for new user. For developers without OpenGL ES experience, WebGL appears to be very complicated.

2.5.4 JOGL

Java OpenGL (JOGL) (Wolff, 2005) is an OpenGL binding library that allows OpenGL to be used in the Java programming language. It allows most OpenGL features through the use of Java Native Interface (JNI). It offers access to both the standard GL functions along with the GLU functions; however the OpenGL Utility Toolkit (GLUT) library is not available for window-system related calls, as Java has its own windowing systems: Abstract Window Toolkit (AWT), Swing, and some extensions. An application that uses JOGL can run on

Windows, Mac OS X and Linux.

The advantage of JOGL is that it provides full access to the OpenGL APIs (version 1.0, 4.3, ES 1, ES 2 and ES 3) as well as nearly all the vendor extensions. Hence, all the features in OpenGL are included in JOGL. JOGL integrates with the AWT, Swing and Standard Widget Toolkit (SWT). It also includes its own Native Windowing Toolkit (NEWT). Hence, it provides complete support for windowing.

However, the OpenGL programming style is based around affecting a global graphics state, which makes it difficult to structure Java code into meaningful classes and objects. JOGL does provide class structuring for the OpenGL API, but the vast majority of its methods are in the very large GL and GLU classes. JOGL was designed for the most recent versions of the Java platform. It also only supports true colour (15 bits per pixel and higher) rendering; it does not support colour-indexed modes.

2.5.5 Other Technologies

These technologies mentioned above are not the only 3D technologies for the web. There are other technologies that are worth mentioning.

1. 3DMLW

3DMLW is an open source platform, or technology, for the creation of interactive 2D and 3D contents for the web. 3DMLW is a technology based on XML. It has scripting support for the creation of dynamic and interactive contents, and event handling, which includes mouse, keyboard and collision events. It also allows the use of textures, lighting, shading, audio, particle engines and physics engines with collision detection.

3DMLW has been evolving to become a cross platform and cross browser compatible technology. By now, it is fully functional for Firefox, Safari, Opera, Chrome and Internet Explorer browsers, and for Microsoft Windows applications. The Mac OS X and Linux distributions are in beta versions and still cause problems.

This technology is similar to X3D and Ajax3D, because it also encodes its contents in a XML-based file format, supports scripting to enhance interactivity, and needs a web browser plug-in to render the contents within the file. However, it is still a limited technology when compared to X3D, because X3D has more advanced graphics facilities.

2. O3D

O3D is an open source JavaScript API created by Google for creating interactive 3D graphics applications that run in a web browser window or in a desktop application. O3D is viewed as bridging the gap between desktop based 3D accelerated graphics applications and HTML based web browsers.

An O3D application runs in an O3D browser plug-in. This plug-in provides hardware acceleration, advanced texturing, advanced shading capabilities and sophisticated rendering techniques. Despite providing truly impressive 3D environments within browser, O3D still needs the use of a web browser plug-in and it is intended for web developers with a solid background in 3D graphics. Also, the rendering of very complex and detailed 3D worlds may become very slow if the client computer does not have a good graphics card.

3. Flash 3D

The Adobe Flash Player is one of the most popular platforms to create interactive and visually outstanding 2D and/or 3D text, animations, web games. Usually, the Flash applications were developed by using Adobe ActionScript language. This scripting language, created by Macromedia in 1998, is based on ECMAScript and can be used for enhancing and complementing the functionalities of the Flash Player.

Flash is now a powerful, and massive used platform, to create very visual appealing, complex, data-rich and interactive 2D and (limited) 3D contents for the web. Because of that, and due to the emerging 3D content demand, powerful 3D flash engines are emerging and evolving rapidly. 3D flash engines like Papervision3D, Sophie3D, Away3D, among others, eases the development of 3D contents for the web, using Flash and ActionScript.

Flash applications can also be created and displayed in mobile phones, portable electronic devices and Internet-connected digital home devices, through a lightweight version of Adobe Flash Player called Adobe Flash Lite.

However, in order to display and interact with the Flash content, the Adobe Flash Player plug-in has to be installed in the user web browser. Flash3D requires the use of proprietary tools to create 3D web applications. Besides, create 3D contents, developers must have a solid knowledge of Flash and ActionScript.

All those technologies discussed above can be used to build and deliver 3D models for web-based applications, but they all suffer different drawbacks. Most of the technologies have no native browser support and are only available through a third party plug-in. Besides, the file sizes of some 3D models built using the existing methods are very big, so the upload (hence on-line) transaction time is much longer than a normal user expects. And for those technologies based on Javascript, the performance will be heavily affected when rendering complex 3D scene, since Javascript is an interpreted computer programming language, and it is a programming language of the web; it is not really efficient for mathematic calculation. Therefore, it is necessary and important to propose and develop a new method for web-based 3D modelling that addresses these problems.

This section has examined the technologies and APIs used in delivering 3D graphics in a web-based application. The next section will focus on the SVG and its applications.

2.6 SVG and Its Applications

SVG is a language for describing 2D graphics in XML. SVG allows three types of graphic objects: vector graphics, raster graphics, and text. Graphical objects, including PNG and JPEG raster images, can be grouped, styled, transformed, and composited into previously rendered objects.

SVG has been in development since 1999 by a group of companies within the WWW Consortium (W3C) after the competing standards Precision Graphics Markup Language (PGML) and Vector Markup Language (VML) were submitted to W3C in 1998. SVG drew on experience from the designs of both those formats.

Konqueror was the first browser to support SVG in February 2004. The Opera browser had fairly extensive SVG support in early 2005, and Firefox developed support for basic SVG shortly after. By mid-2007, Safari had implemented support for basic SVG as well. Google released its Chrome browser with SVG support in 2008, and in 2009 Microsoft announced that Internet Explorer would finally have native support. As of 2011, all major browsers, and many minor ones, have some level of SVG support (Dailey, 2012).

There is no official development environment for SVG, but some vector graphics editors such as Inkscape (Mihaela, 2011), Adobe Illustrator (Sharon, 2013), or CorelDRAW, can be used to develop a SVG application. Even some simple text file editors, such as Microsoft NotePad, can also be used to edit a SVG file.

SVG is resolution-independent, making it ideal for rendering cross-platform user interface components, animations and applications where each element needs to be accessible via the Document Object Model (DOM). SVG can be used as a platform upon which to build graphically rich applications and user interfaces web-based applications. Developers use SVG for various sorts of interactive graphics applications (flow charts, business graphics, and mapping). Since SVG is XML-based, it can render graphics from database data, so images can be dynamically updated. This means an image contained in a website can be dynamically changed according to the data it has retrieved.

SVG is a useful, elegant, and important tool for building informative and appealing graphics. It can be used to accomplish a broad range of effects, ranging from practical to artistic, while making graphics both dynamic and interactive. All these features made SVG a good candidate for web based graphics rich applications.

SVG is particularly useful for data driven visualization of business data, charts, maps and technical drawings, as it can be generated using XSLT conversion or any scripting or programming language the developer is familiar with. SVG has been widely used for 2D graphics data representation in various fields. Some researchers have used it as a visualization language for different types of scientific data. For example, (Baravalle, 2003) use SVG and XSLT to visualize dynamically changing data. (Chang, 2002) use SVG to visualize census data online. (Lewis, 2002) use SVG to visualize medical data. (Baru, 2001) use it to represent statistical data on geographical maps shows how sending a SVG file plus some Javascript code may allow an user with a SVG enabled browser to display and interact with the information sent in many ways.

The other applications of SVG include e-learning, 2D games (Alkalay, 2007), human navigation (Kobayashi, 2003) etc. (Lee, 2002) reports an SVG-based collaborative system, Garnet, for distance-education running on desktops and PDAs. The architecture of Garnet is based on an event brokering system. SVG provides better graphics and document interactive. SVG supports many user interface (UI) events and pointing events. It provides a quick and effective mechanism to process these events. Moving or clicking the mouse over any graphics elements is able to generate immediate feedback, such as highlighting, text tips, and real-time changes to the surrounding HTML text. Animations and scripts executions can also be triggered by this mechanism (Kevin, 2003).

SVG is an open, HTTP compatible standard that allows fully interactive mapping applications-without the need for applets or a round trip to the server every time the map presentation is tweaked.

This section has given a brief introduction of SVG and its applications. The next section will discuss shading and texture mapping in 3D graphics which cannot be integrated in current SVG.

2.7 Shading for 3D Graphical Presentations

Shading refers to the practice of letting colours and brightness vary smoothly across a surface (Funt, 1992). The three most popular kinds of shading are Flat shading, Gouraud shading and Phong shading. All these shading methods can be used to give a smooth appearance to surfaces; even surfaces modelled as flat facets can appear smooth.

Flat shading shades each polygon of an object based on the angle between the polygon's surface normal and the direction of the light source, their respective colours and the intensity of the light source. It is usually used for high speed rendering where more advanced shading techniques are too computationally expensive. The disadvantage of flat shading is that it gives low-polygon models a faceted look.

Gouraud shading is an interpolation method which linearly interpolating a colour across a polygon. It is a very simple and effective method of adding a curved feel to a polygon that would otherwise appear flat. Gouraud interpolation works reasonably well; however, for large polygons, it can miss specular highlights or at least miss the brightest part of the specular highlight if this falls in the middle of a polygon.

Phong shading is also an interpolation method, but instead of linearly interpolating a colour across a polygon, it linearly interpolates a normal across a polygon. Phong shading overcomes some of the disadvantages of Gouraud shading and specular highlights can be successfully incorporated in the scheme. The Phong Shading interpolation phase is three times as expensive as Gouraud Shading, so it significantly increases the computation cost. The other disadvantage of Phong shading is that all the information about the colours and directions of lights needs to be kept until the final rendering stage so that lighting can be calculated at every pixel in the final image.

2.8 Texture Mapping

Texture mapping is the process for adding detail, or colour to the surface of a 3D model. Its use can enhance the visual realism with only a relatively small increase in computation (Tarini, 2000).

Textures can be one, two, or three dimensional. For example, a 1D texture might be used to create a pattern for colouring a curve; a 3D texture, also called solid texture, is basically the equivalent of carving the object out of a block of material. It places the texture onto the object coherently, not producing discontinuities of texture where two faces meet. 3D texture can be used to simulate the wood grain on a cube to avoid discontinuities of grain along the edges of the cube.

2D texture mapping is by far the most common use of texture mapping. 2D textures start out as 2D images which might be formed by application programs or scanned in from a photograph, regardless of their origin; they are eventually brought into processing as an array. The individual elements in these arrays are called texels, or texture elements.

This section has examined the general concepts of shading and texture mapping, further discuss will be given in Chapter 6, and Chapter 7.

2.9 Summary

Different 3D graphics presentation technologies are introduced in this chapter, some inspired primarily by the need for efficiency, and others that aim to render a realistic physical image. Specifically, different web-based 3D graphics presentation technologies have been introduced in this chapter. Although the web-based 3D graphics technologies, like VRML, X3D, WebGL, Flash3D, C3DL, JOGL, 3DMLW and O3D can provide immersive and realistic web 3D environments within a browser, they all require the installation of third party web browsers plug-ins or add-ons (to take advantage of hardware acceleration), which may be a barrier to users that want to experience an easy and immediate interaction with 3D contents.

Table 2.1 Features supported by existing 2D SVG and the proposed 3D SVG

Features	Existing 2D SVG	Proposed 3D SVG
Vector Graphic	Yes	Yes
Highly Interactive	Yes	Yes
Supported by Major Browsers Without Plug-in	Yes	Yes
XML Format	Yes	Yes
3D Object	No	Yes
3D Transform	No	Yes
3D Shading	No	Yes
3D Texture Mapping	No	Yes

SVG-as a potential platform for 3D graphics presentation for web-based application, is also introduced in this section. Table 2.1 shows features supported by existing 2D SVG and the proposed 3D SVG. Existing 2D SVG has features that show that SVG is a good candidate for investigating whether it is possible to build on this technology to develop web-based 3D graphics applications. If this is possible, SVG can become a useful technology to render 3D contents over the web, because it does not need the installation of any web browser plug-ins or add-ons.

Chapter 3 Research Design and Methods

3.1 Introduction

The overall aim of this PhD project will be to research, design, develop and implement a new framework-SVG GL for generic 3D SVG modelling for web-based applications. The context of this project is based on two assumptions:

1. The rapid developments in the field of computer graphics and computer hardware now allows for real time visualization of complex 3D data over the internet;

2. Digital 3D models are playing an increasingly important role in many fields such as product demonstration, city planning, site exhibition, 3D online gaming, decision-making and 3D e-Business (such as 3D shopping malls), etc.

In comparing it with existing methods, the core work in this project attempts to achieve high performance in dealing with 3D web-based modelling. In this new framework-SVG GL, there are fundamental works that are related to graphics library, 3D solid models of various primitives, 3D SVG models with freeform surfaces, new algorithms for shading and texture mapping of the SVG GL models. The new framework and the algorithms will be tested with 4 web-based 3D applications, and evaluated with another 4 web-based 3D applications.

3.2 Methods for Data Collection and Presentation

The collection, organization, and presentation of data are basic background material for testing and analyzing the methods provided in this project.

After identifying the research problem and selecting the appropriate methodology, researchers must collect the data that they will then go on to analyze. There are two sources of data: primary and secondary sources. Primary data are data collected specifically for the study in question. Primary data may be collected by methods such as personal investigation or mail questionnaires. In contrast, secondary data are mainly collected through literature review.

In this project, both primary data and secondary data are used, and the data needed are either from the existing literature- secondary data, or generated by own design and development of mathematical models and software computations-primary data.

Based on the overall aim and objectives presented in Section1.3, the secondary data that should be collected are the published materials about (1) the existing methods for 3D modelling for web-based applications, (2) the existing 2D SVG applications and (3) the geometric structure and features of 3D models. The primary data are mainly (1) various SVG 3D models of primitives, SVG 3D models of freeform surfaces, and (3) validation test results of the new framework and various new algorithms.

3.3 Research Design

The research procedure of this PhD project is shown in Figure 3.1.

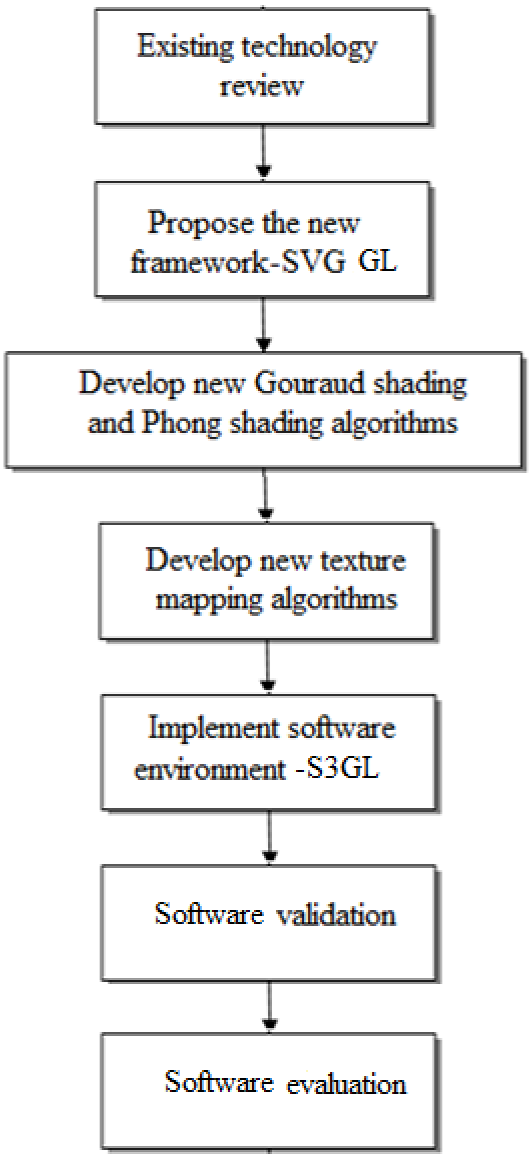

Existing technology review
Propose the new framework-SVG GL
Develop new Gouraud shading and Phong shading algorithms
Develop new texture mapping algorithms
Implement software environment -S3GL
Software validation
Software evaluation

Figure 3.1 Research procedure

1. Existing technologies review

The technologies for window-based 3D presentation are reviewed first. Then current technologies for web-based 3D presentation are discussed, and advantages and drawbacks of each technology are reviewed. Finally the SVG theory and its applications are reviewed, the problems with the existing SVG 2D are also discussed. The need for the SVG GL is documented. This work will be done through systematic literature review.

2. Propose the new framework-SVG GL

The next stage is to develop a new framework-SVG GL based on classical 3D graphic theory and SVG. While the model is initialized using classical 3D graphics, the scene model is extended using SVG. A new algorithm to present 3D graphics with SVG is proposed. Define a 3D scene in the framework, integrate 3D objects, camera, transformation, light model and texture in a 3D scene, and render 3D objects on the web page, allowing the end-user to interactively manipulate objects on the web page.

3. Develop new Gouraud and Phong shading algorithms.

Develop new Goraud shading algorithm and Phong Shading algorithm to implement Gouraud shading and Phong shading in the SVG GL. The algorithms can be used to generate smooth shading and create highlight for 3D objects.

4. Develop new texture mapping algorithms

Develop novel texture mapping algorithms-pattern based image transformed texture mapping algorithm for the SVG GL oriented toward web-based 3D modelling application. Texture mapping algorithms for different 3D objects such as triangle, plane, sphere, cylinder, cone, etc. also proposed.

5. Implement software environment-S3GL.

In order to validate the proposed new framework, the new Gouraud shading and Phong shading algorithms, and the new texture mapping algorithms, a software environment-S3GL is developed based on the proposed theory. The S3GL will be validated firstly, to prove it can be used to create desired 3D scene.

6. Software validation

Four 3D test applications are implemented based on this S3GL to validate the new framework proposed in Chapter 5, the new Gouraud shading and Phong shading algorithms proposed in Chapter 6, and the new texture mapping algorithms developed in Chapter 7.

7. Software evaluation

Four 3D demo applications are also implemented based on this S3GL to evaluate the framework and algorithms proposed in this PhD project. And discuss the potential application fields of the SVG GL

3.4 Methods for Verification and Validation

This thesis is mainly exploratory with some experimental validation work through a self designed and developed software environment.

The first objective-investigates and evaluates existing methods for 3D modelling for web-based applications. To achieve the first objective, there will be a systematic review of the published literature to evaluate the existing 3D modelling methods for web-based applications, investigating and evaluating different 3D construction methods to define 3D models, including boundary representation (Krysl, 2001), constructive solid geometry (Foley, 2013); and also evaluating different 3D web data format, such as VRML (Taubin, 1998), X3D (Vucinic, 2008), CityGML (Kolbe, 2005), O3D. The subsequent analysis of web-base application aspects that are relevant to 3D graphics, and of the required extensions of the web-based body of knowledge, was especially important due to the conviction that without the application of web-based rules, 3D graphics cannot attain their full potential efficiency of information transfer.

The second objective - evaluate the existing 2D SVG applications and to investigate the possibility of using SVG to realize 3D graphical representations for various web-based applications -was to provide a set of differentiating factors, based on the existing theory, that allow a clear distinction between SVG and other forms of 3D presentations. To achieve the second objective, it is planned to evaluate existing SVG applications and investigate the possibility of using SVG to realize 3D graphical representations, analyze the state-of-the-art of the relevant research areas and technologies by literature review, conference presentations, meetings, discussions and brain storming sessions with other researchers and professionals from the industry.

Once the principle of the SVG GL development has been established, the next step is to achieve the third objective by systematically analyzing the geometric structure and features of 3D models and proposes a new generic method for 3D modelling based on SVG.

The fourth objective-implements the proposed method and validates it through development and evaluation of typical 3D web-based applications-required long and intensive work. To achieve the fourth objective, a software environment was designed and developed to implement the proposed method. It is validated through the design and development of dedicated 3D geometrical models for different web-based applications by using the proposed method, and explores potential applications of the SVG GL modelling for web-based applications.

The purpose of developing this generic 3D modelling system is for designing and developing 3D models for web-based applications. So the criteria for validation of the 3D modelling system's effectiveness are that it can be used to build, modify and view parameterized 3D geometrical models of all primitive geometries and free form surfaces for a variety of web-based applications without the requirements of any plug-ins and special model editing software. To demonstrate the new proposed method, four demo applications are developed in this project.

1. A 3D bottle model.
2. A 3D building site.
3. A supermarket.
4. A 3D landscape.

Those applications are used to investigate the potential application fields of the SVG GL. By successfully running those demo application, it shows the SVG GL can be used for product demonstration, urban environment simulation, city planning; for warehouse demonstration and 3D terrain simulation, etc.

3.5 Ethical Issues

In this project, the data needed are either from existing literature or generated by the researcher's own design and development of mathematical models and software computations. These data were generated for public use, so there is no ethical issue in terms of privacy and data protection in this project.

3.6 Summary

This chapter has discussed the research design and methods used in this project. The project is validated through a self designed and developed software environment. Since the data needed are either from existing literature or generated by the researcher's own design and development of mathematical models and software computations, so there is no ethical issue in terms of privacy and data protection in this project. The next chapter will go on to focus on SVG theory and its application s in detail.

Chapter 4 SVG Theory and Its Applications, and OpenGL

4.1 Introduction

Chapter 3 has discussed the research design and methods used in this project. This chapter will go on to focus on SVG theory and its applications in detail.

SVG is an XML-based markup language for describing 2D graphics applications and images, and a set of related graphics script interfaces. SVG graphics can be interactive and animated. Bindings for scripting languages and network interfaces enable developers to build rich interactive graphics applications.

SVG is developed by W3C for describing 2D vector graphics for storage, presentation and distribution on the Web. The first public draft of SVG was released by the W3C in February of 1999. By the end of June 2000, 9 subsequent working drafts appeared. SVG 1.0 was released as a W3C Recommendation in September of 2001(Dailey, 2010). Since 2001, the SVG specification has been updated to version 1.1. SVG 1.1 became a W3C Recommendation on 14 January 2003; SVG 2 is currently under development, and will add new features to SVG, as well as more closely integrating with HTML, CSS, and the DOM. In 2001, SVG got a facelift to include mobile profiles. The SVG Mobile Recommendation introduced two simplified profiles of SVG 1.1, SVG Basic and SVG Tiny for devices with reduced computational and display capabilities. SVG Tiny and SVG Basic (the Mobile SVG Profiles) became W3C Recommendations on 14 January 2003. An enhanced version of SVG Tiny, called SVG Tiny 1.2, became a W3C Recommendation on 22 December 2008 (Mong, 2003).

All major modern web browsers—including Mozilla Firefox, Internet Explorer 9 and 10, Google Chrome, Opera, and Safari—support basic SVG and can render the markup directly without the use of a plug-in. SVG is supported in every browser except for versions of Internet Explorer earlier than Version 9 and versions of Android earlier than Version 3. This includes support for fonts, images, graphical elements such as circles or paths, as well as gradients and some of the filters. There are several advantages to native and full support: a plug-in is not needed, SVG can be freely mixed with other content in a single document, and rendering and scripting become considerably more reliable.

SVG provides an open, standard based format for creating graphics. Using SVG has numerous advantages over other conventional bitmapped graphics, such as JPEG, GIF, and PNG. The files are generally much smaller than bitmaps, resulting in quicker download times. The graphics can be scaled to fit different display devices without the pixelation associated with enlarging bitmaps. The graphics are constructed within the browser, reducing the server load and network response time generally associated with web imagery. The file is an XML-based file format; it allows the creator to conveniently embed arbitrary information inside of the file. SVG is well suited for graphics rich environments. SVG can be used for GIS, embedded systems, location-based services, animated picture messaging, multimedia messaging, animation and interactive graphics, entertainment, e-Business, and graphic user interfaces. After integrating 3D into SVG, it can even play an important role in many fields such as product demonstration, city planning, site exhibition, 3D e-Business (such as 3D shopping malls), etc..

4.2 SVG Theory

This section will examine the working theory of SVG, and some important features of SVG.

SVG is a language that allows for the creation of 2D vector elements, which are simply mathematical representations of graphical objects. These vectors are infinitely scalable and can be transformed within the bounds of the 2D coordinate system. SVG is based on vectors rather than pixels (Rosenbaum, 2004). While a pixel-based approach places pigment or colour at *xy*-coordinates for each pixel in a bitmap, a vector-based approach composes a picture out of shapes; each described by a relatively simple formula and filled with a texture.

SVG documents are built upon a regular XML document tree, consisting primarily of a header, processing instructions, comments, XML elements and attributes. SVG uses a ‘painter model’ for rendering. Paint is applied in successive operations to the output device such that each operation paints over some area of the output device. When the area overlaps a previously painted area, the new paint partially or completely obscures the old. When the paint is not completely opaque, the result on the output device is defined by the mathematical rules for composing. Elements in an SVG document fragment have an implicit drawing order. Elements that appear first in the document tree are rendered first; subsequent elements are drawn on top of the previous elements, taking into account opacity, blending, filters, clipping and masking. The following is a typical SVG file. And the result of this file is shown in Figure 4.1.

```
<svg height="200" width="500">
   <polygon points="100,10 40,198 190,78 10,78 160,198"
style="fill:red;stroke:blue;stroke-width:5;fill-rule:nonzero;"/>
</svg>
```

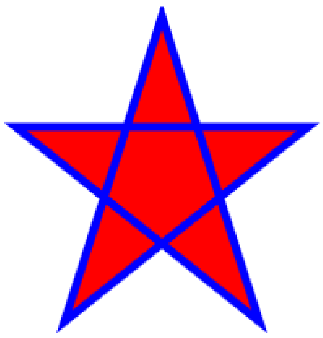


Figure 4.1 A star create by SVG

4.2.1 SVG Coordinate System

The coordinate system of SVG is a bit different from the coordinate systems of mathematics.

In mathematics, the point x=0, y=0 in a normal Cartesian coordinate system is at the lower left corner of the graph. As x increases the points move to the right in the coordinate system. As x decreases the points move to the left in the coordinate system. As y increases the points move up in the coordinate system. As y decreases the points move down in the coordinate system. In SVG coordinate system the point x=0, y=0 is the upper left corner. The y-axis is thus reversed compared to a normal graph coordinate system. As y increases in SVG, the points, shapes etc. move down.

The SVG coordinate system can be specified to any units in Table 4.1.

Table 4.1 The unit of an SVG coordinate system

em	The default font size - usually the height of a character
ex	The height of the character x
px	Pixel
pt	Point (1 / 72 of an inch)
pc	Pica (1 / 6 of an inch)
cm	Centimeter
mm	Millimeter
in	Inch

Units can be specified after the coordinate value, such as 10 cm, 25mm. If no unit is specified after a coordinate value, the default unit is assumed to be pixels (px).

4.2.2 SVG Basic Geometry Elements

According to the W3C's Recommendations, the SVG basic geometric elements are "the element types that can cause graphics to be drawn onto the target canvas". Those are: 'path', 'rect', 'circle'', 'ellipse'', ''line', 'polyline', and 'polygon' (Dahlström, 2011).

The above basic geometrical elements are more or less self explanatory. The most powerful and interesting geometric type is the <path /> element. A path is described using the concept of a current point. In an analogy with drawing on paper, the current point can be thought of as the location of the pen. The position of the pen can be changed, and the outline of a shape (open or closed) can be traced by dragging the pen in either straight lines or curves. Path elements can contain quadratic and cubic spline curves and arc segments. Geometry can be described in either absolute or relative coordinates. Mathematically, all the other geometry elements are shorthand forms for the 'path' element that would construct the same shape.

4.2.3 Text and Fonts

SVG has powerful text capabilities. SVG has the following text features: font specification, text orientation and direction, text alignment, and rich text formatting. The <text/> element will cause a single string of text to be rendered. Like any other basic shape, text can also have fillings, strokes and can be clipped or masked or can serve as a clipping path. The text strings within <text/> elements can be shifted or rotated and can also be aligned on path elements. It is even possible to animate a text along a path. SVG fully supports international text processing features for both straight line text and text on a path, including Unicode support, left to right text, bidirectional text or text that runs from top to bottom.

An SVG font is a font defined using SVG's <font/> element. The purpose of SVG fonts is to allow for the delivery of glyph outlines in display-only environments. SVG fonts that accompany web pages have to be supported only in browsing and viewing situations. To ensure that the SVG file displays the correct font, an SVG font can be either embedded within the same document that uses the font or saved as part of an external resource.

4.2.4 Filling, Stroking, Opacity

SVG elements can be filled and stroked. Filling and stroking both can be thought of in more general terms as painting operations. Generally SVG elements can be painted with: uniform single colour, linear and radial gradients and patterns. A pattern is used to fill or stroke an object using a pre-defined graphic object. The graphic object can be raster data, vector elements and animations. A gradient consists of continuously smooth colour transitions along a vector from one colour to another, possibly followed by additional transitions along the same vector to other colours. Gradients parameters can be animated as well. Opacity can be separately defined for strokes, filling or both. Group opacity treats elements as a group as opposed to treating each group element individually.

4.2.5 Styling

SVG uses styling properties to describe many of its document parameters. Styling properties define how the graphics elements in SVG content are to be rendered. There are alternative ways to style elements in SVG. One can use Cascading Style Sheets (CSS) styles, XML presentation attributes or Extensible Style sheet Language Transformations (XSLT).

XSLT offers the ability to take a stream of arbitrary XML content as input, apply potentially complex transformations, and then generate SVG content as output.

XSLT can be used to transform XML data extracted from databases into an SVG graphical representation of that data.

CSS is a widely implemented declarative language for assigning styling properties to XML content, including SVG. It represents a combination of features, simplicity and compactness that makes it very suitable for many applications of SVG.

XSLT style sheets define how to transform XML content into something else, usually other XML. When XSLT is used in conjunction with SVG, sometimes SVG content will serve as both input and output for XSLT style sheets. At other times, XSLT style sheets will take non-SVG content as input and generate SVG content as output.

4.2.6 Filters

Filter features are unique to SVG. A filter effect consists of a series of graphics operations that are applied to a given source graphic to produce a modified graphical result. The result of the filter effect is rendered to the target device instead of the original source graphic. Filters can be attached to both raster and vector elements. Vector elements are rasterized during the rendering pipeline; hence there is an opportunity to include filters.

Typical applications for filters are colour corrections, brightness and contrast adaption, blurring and sharpening, illumination filters, generation of drop shadows and halo effects, convolution filters, displacement and morphology filters, generating turbulence, etc. Filters may be combined in any order and the output of one filter may be piped to the input of the next filter. Every filter parameter can be animated which can lead to very interesting effects. Filters are very powerful visualization options, but may require a fair amount of computing power.

4.2.7 Interactivity and Scripting

Interactivity and scripting are key parts when it comes to making SVG appealing for dynamic web applications. SVG content can be interactive by utilizing the following features: user-initiated actions such as button presses on the pointing device; user can initiate hyperlinks to new Web pages by actions such as mouse clicks; users are able to zoom into and pan around SVG content; user movements of the pointing device can cause changes to the cursor that shows the current position of the pointing device.

SVG proposes a variety of user events. Three event categories are specified: mouse events, keyboard events, and state change events. Events can trigger either a script function or a SMIL interaction. Mutation events listen to changes within a particular node in the XML document tree.

The other, more flexible, way of modifying SVG documents is to use a client side scripting language. Scripts can either be embedded in SVG files or referenced (external files). SVG defines a language independent API to access and manipulate the SVG DOM. The most widely used and implemented scripting language in conjunction with SVG is ECMAScript (the standardized version of Javascript).

4.2.8 Animation

SVG supports the ability to change vector graphics over time. Almost any element and attribute can be animated in SVG.

SVG content can be animated in the following ways: the first way is to use SVG's animation elements. The various elements can define motion paths, fade in or fade out effects, and objects that grow, shrink, spin or change colour. The second way is use the SVG DOM. Every attribute and style sheet setting is accessible to scripting. SVG offers a set of additional DOM interfaces to support efficient animation via scripting. Therefore, any kind of animation can be achieved. The timer facilities in scripting languages such as ECMAScript can be used to start up and control the animations. The third way is to use SMIL, a descriptive way to define animation parameters. SMIL animations can trigger script execution and vice versa.

SVG offers interpolation of in-betweens. Interpolation options are: step-by-step, linear, or spline. Various parameters may be animated, such as colour value, position, position along a path, rotation, scale, etc.

4.2.9 Adding SVG to a Webpage

There are a number of ways to add SVG to a webpage.

1. Use the object element and reference an external SVG file. This approach is currently the most popular way to add SVG to a page served up as HTML.

```
<object type="image/svg+xml"
        width="100" height="100" style="float:right"
        data="image.svg">
</object>
```

the type is set to the SVG MIME type “image/svg+xml.” Supply the width and the height, and set the data attribute to the SVG file.

Currently, using the object element is the only native approach that works with HTML. As support for HTML5 increases, more browsers will support SVG embedded directly into HTML.

2. Use iframe, embed, or img elements (depending on the browser) to embed SVG into a webpage. Not all of these embedding methods are available for every browser.

```
<iframe src="image.svg"> Your browser does not
support iframes</iframe>
```

```
<embed type="image/svg+xml" src="image.svg" />
```

```
<img src="image.svg" />
```

The type is set to the SVG MIME type “image/svg+xml.”, and src the data attribute to the SVG file.

3. SVG can also be incorporated into a webpage by being used as a CSS background for any element:

```
#myelement
{
        background-image: url(image.svg);
}
```

4. The last approach to adding SVG to a web page is embedding the SVG directly into the web page by using the SVG element. The method works in all HTML5 browsers and also permits animation, scripting and CSS.

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>Embedded SVG</title>
</head>
<body>
<h1>Embedded SVG</h1>
<svg width="300px" height="300px"
xmlns="http://www.w3.org/2000/svg">
		<text x="10" y="50" font-size="30">My
SVG</text>
</svg>
</body>
</html>
```

4.3 Evaluation of SVG Applications

SVG has been widely used in the works over the past decade and has matured a great deal during that time, with collaboration from interested parties around the world. The great appeal of SVG is that, like HTML, it’s easy to read and edit, while allowing for complex interactivity and animations through scripting and Synchronized Multimedia Integration Language (SMIL), which is another W3C standard.

SVG can be used for static images within a Web page (Eisenberg, 2002). SVG, being a vector graphic, can scale to fit the web page, while bitmap images such as JPEG and GIF cannot, or at least, can’t scale cleanly. Compare the following two screenshots of the same image (Figure 4.2).

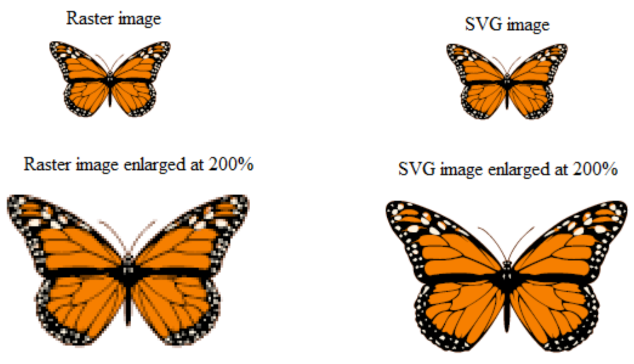


Figure 4.2 Compare enlarged raster image and SVG image

The raster image becomes very pixilated when it is scaled while there is no quality lost with the SVG image. This is because a raster image describes each and every pixel in the image, but SVG describes the image as shape elements or objects. The vector-based viewers are able to recalculate how the graphic should look based on the shape description that is found inside the SVG graphic.

SVG is well suited to playing a major role in graphics rich environments. The visualization options available in SVG graphics go beyond competing file formats. Any attribute can be animated and the available interactivity options and script bindings allow the building of fully interactive applications that do not need to hide from stand-alone offline multimedia applications. It is important to note that SVG should be used as a complementary technology and in conjunction with other established web-technologies, such as XML, XHTML, static raster graphics and movies (Chang, 2004). SVG is primarily a presentation and exchange format that can and should be generated out of other storage formats, databases and XML sources (Brodlie, 2002). SVG should be used for static illustrations, animations and interactive applications.

SVG is well suited for presenting engineering technical drawings and explaining, visualizing or simulating instruments (Su, 2006). Animations can visualize the operation of machines, technical devices or circuit diagrams. In technical drawings one can display non-graphical attributes (such as article numbers or part names) on mouse-over. In simulations, the user can interactively manage control panels, control flow or change environmental parameters.

The rich visualization and interactivity options of SVG make it particularly useful for mapping and GIS (Sheng, 2005; Huang, 2011). The available fill and stroke options, symbols and markers enable higher quality map graphics and complex symbolizations. Interactivity helps display additional non-graphical data and enables analysis functions. Basic GIS functionality can be directly implemented in SVG, while more complex GIS analysis functions can be delegated to server side GIS or spatial databases. In the latter case, SVG is only used as a presentation tool. Data acquisition and analysis functions can be directly practiced in interactive SVG applications.

SVG can also be used for smaller games and animations (Probets, 2001). A website dedicated to SVG and gaming (Alkalay, 2007) lists a number of SVG based games.

4.4 Discussion of Problems of SVG Applications

SVG integrates and leverages other W3C standard technologies already familiar to web programmers: DOM, JavaScript, CSS (Sons, 2010). Rather than having to learn entire realms of technology, programming languages, and terminology to deal with the complex and technical area of computer graphics, designers, programmers, and web professionals can leverage skills learned elsewhere.

SVG is suitable for incorporation with HTML5, web-based applications, and rich Internet applications (RIAs). The last 10 years have seen a great elevation of the status of the phrase "web-based application". Not so many years ago, people in the web community used to respond with disbelief when someone talked about wanting to create a web-based application that lived primarily in the browser. A cursory inspection of the history of HTML5 reveals that the creation of web applications was one of the primary intentions behind the development of this emerging specification. The incorporation of inline SVG into the HTML5 specification is a great advantage for web developers.

SMIL is a W3C declarative language supporting multimedia and animation for nonprogrammers. SMIL is partially incorporated into the SVG specification. Those who have had more than a cursory exposure to programming animation in JavaScript may find themselves enamoured of the ease with which certain complex animations can be authored using SVG animation (or SMIL), as well as the ability to update many objects on the screen almost concurrently. While SVG also supports scripted animation through JavaScript, SMIL brings convenience, parsimony, and elegance to the table.

SVG is supported natively by the most current versions of the five major web browsers. Additionally, it can be found in the chip sets aboard several hundred million mobile phones, with major support being offered from Nokia, Ikivo, SonyEricsson, Opera Mobile, Samsung, iPhone, and several others (Dailey, 2012).

Compared with other similar technologies such as Flash, Vector Markup Language (VML), and Silverlight, SVG has the advantages of being nonproprietary, standardized, cross platform, and interoperable with other XML languages and W3C standards.

In summary, SVG has several key advantages over other graphics formats used on the Web. These include:

1. SVG is XML-based, so it is compatible with XML, HTML4, XHTML as well as CSS, XSLT, and the DOM which means that SVG is extensible, can be

styled, scriptable, and interactive and integrates easily with other XML languages. Because the SVG source code is written in XML, it is readable by screen readers and search engines, and therefore can be “searched” or “indexed”.

2. SVG uses vector technology, not raster technology. Vector graphics exist in the world of mathematics. So SVG is a combination of geometry shapes rather than pixels. This is one reason that SVG images can be scaled without distortion or losing quality. SVG is resolution independent. SVG offers a way to do full resolution graphical elements, no matter what size screen, what zoom level, or what resolution user's device has. SVG files are generally much smaller than bitmaps, resulting in quicker download times. This makes SVG ideal for use on the Web.

3. SVG is plain text, which means developers and designers can edit SVG files using a wide variety of tools. There is no official development environment for SVG, but some vector graphics editors such as Inkscape, Adobe Illustrator, or CorelDRAW, can be used to develop an SVG application. Even some simple text file editors, such as MicroSoft NotePad, can also be used to edit a SVG file.

4. SVG is an open standard. SVG is an open, HTTP compatible standard that allows fully interactive mapping applications - without the need for applets or a round trip to the server every time the map presentation is tweaked. SVG can render on most of the modern web browsers directly without the use of a plug-in.

Apart from all the positive aspects of SVG there are unfortunately also weak aspects. One issue is that the use of SVG on the web is still limited by the lack of support in older versions of Internet Explorer. IE Version 8 does not support SVG, although there are not many people still use IE8, but it will cause problems for those who still use IE8 or earlier version. IE9 which was introduced on March 0f 2011, supports the basic SVG feature set.

The other drawback of the SVG approach is that good tools for content creation are still in their infancy. While it is trivial to create static and animated SVG graphics, tools for scripting development are not yet mature. Hence, the content creation of highly interactive content is still reserved to the more computer literate developers who are used to directly working in the source code.

There is the problem of not being able to hide the source code effectively. This can be a positive feature, but quite a few content creators are hesitant to use open standards where they cannot use code protection. While methods exist to obfuscate Javascript or disable the "View Source" function in the Adobe SVG viewer, it is usually trivial for computer literate people to still have access to the source code. This is also possible for documented binary formats, such as Flash. Quite a few programs exist to decompose .swf files and extract the individual media elements, such as graphics, movies and text. To be able to hide the source code in the future, Adobe is looking into the digital rights management.

Finally, but most importantly, SVG only supports 2D graphics. So it is necessary and both theoretically and practically important to study the possibility of apply SVG for 3D modelling in various web-based applications, incorporating the interactive features demanded by such applications. That is the main purpose of this PhD project. Here, 'interactive' means that users can actively interact with web-based systems to build, modify and dynamically view 3D models.

The next section will focus on one of the most popular libraries for 3D graphics programming-OpenGL. Through the introduction of OpenGL, understand the necessary features of a 3D graphics presentation framework.

4.5 OpenGL Theory

OpenGL is an open standard API that provides a number of functions to render 2D and 3D graphics, and is available on most modern operating systems including Windows, Mac OS X and Linux. This API consists of more than 700 distinct commands that can be used to specify the objects and operations needed to produce interactive 3D applications.

OpenGL is designed as a streamlined, hardware-independent interface to be implemented on many different hardware platforms. To achieve these qualities, no commands for performing windowing tasks or obtaining user input are included in OpenGL; instead, you must work through whatever windowing system controls the particular hardware you're using. Similarly, OpenGL does not provide high-level commands for describing models of 3D objects. Such commands might allow you to specify relatively complicated shapes such as automobiles, parts of the body, airplanes, or molecules. With OpenGL, you must build your desired model from a small set of geometric primitives—points, lines, and polygons. A sophisticated library that provides these features can certainly be built on top of OpenGL. The OpenGL Utility Library (GLU) provides many of the modelling features, such as quadric surfaces and NURBS curves and surfaces. GLU is a standard part of every OpenGL implementation.

Most implementations of OpenGL have a similar order of operations, a series of processing stages called the OpenGL rendering pipeline. A pipeline, in computing terminology, refers to a series of processing stages in which the output from one stage is fed as the input of the next stage, similar to a factory assembly line or pipe. This ordering, as shown in Figure 4.3, is not a strict rule about how OpenGL is implemented, but it provides a reliable guide for predicting what OpenGL will do.

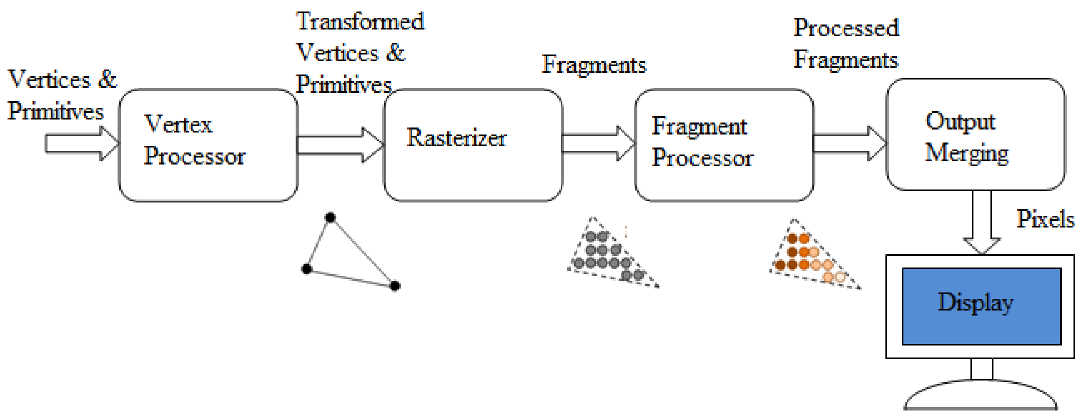


Figure 4.3 Order of operations

The OpenGL rendering pipeline consists of the following main stages:

1. Vertex processing: Process and transform individual vertices.

2. Rasterization: Convert each primitive (connected vertices) into a set of fragments. A fragment can be treated as a pixel in 3D spaces, which is aligned with the pixel grid, with attributes such as position, colour, normal and texture.

3. Fragment processing: Process individual fragments.

4. Output merging: Combine the fragments of all primitives (in 3D space) into 2D colour-pixel for the display.

In modern GPUs, the vertex processing stage and fragment processing stage are programmable. The shader programs are written in C-like high level languages such as GLSL (OpenGL Shading Language), HLSL (High-Level Shading Language for Microsoft Direct3D), or Cg (C for Graphics by NVIDIA).

4.5.1 3D Graphics Coordinate Systems

OpenGL adopts the Right-Hand Coordinate System (RHS). In the RHS, the x-axis is pointing right, y-axis is pointing up, and z-axis is pointing out of the screen. With right-hand fingers curving from the x-axis towards the y-axis, the thumb is pointing at the z-axis. RHS is counter-clockwise (CCW). The 3D Cartesian Coordinates is a RHS.

4.5.2 Primitives

OpenGL supports three classes of geometric primitives: points, line segments, and closed polygons. They are specified via vertices. Each vertex is associated with its attributes such as the position, colour, normal and texture. OpenGL provides 10 primitives as shown in Figure 4.4.

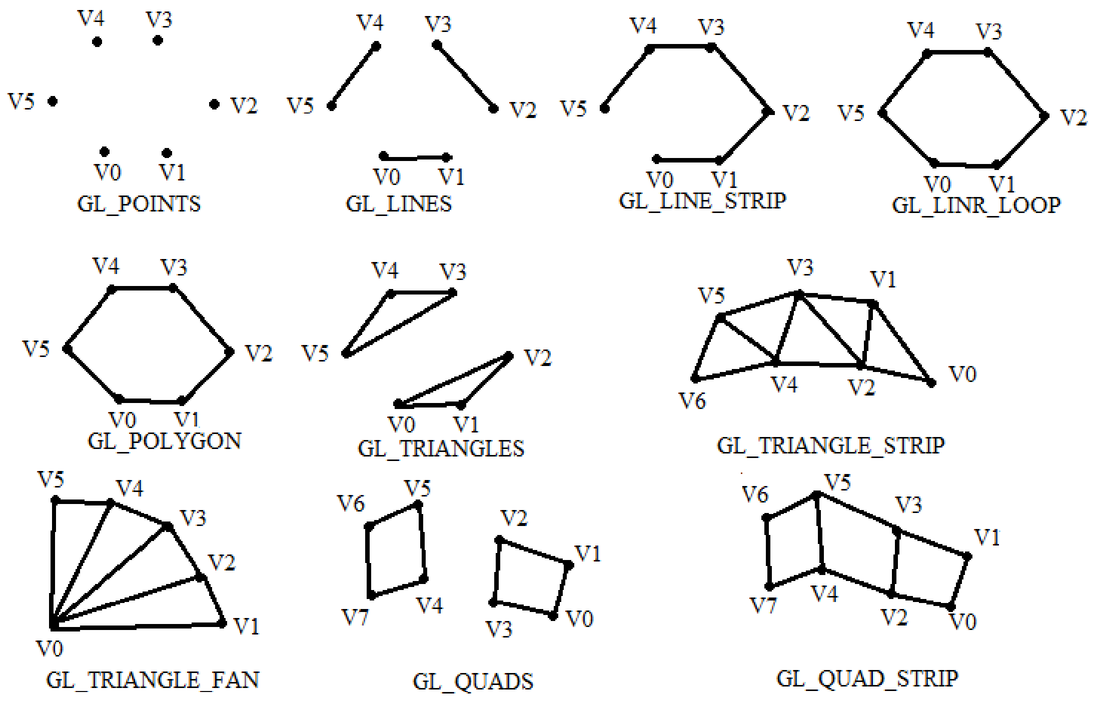


Figure 4.4 OpenGL primitives

4.5.3 Vertices

Recall that a primitive is made up of one or more vertices. A vertex, in computer graphics, has these attributes:

1. Position in 3D space V=(x, y, z): typically expressed in floating point numbers.

2. Colour: expressed in RGB (Red-Green-Blue) or RGBA (Red-Green-Blue-Alpha) components. The component values are typically normalized to the range of 0.0 and 1.0 (or 8-bit unsigned integer between 0 and 255). Alpha is used to specify the transparency, with alpha of 0 for totally transparent and alpha of 1 for opaque.

3. Vertex-Normal N=(nx, ny, nz): the normal vector is perpendicular to the surface. In computer graphics, however, we need to attach a normal vector to

each vertex, known as vertex-normal. Normals are used to differentiate the front- and back-face, and for other processing such as lighting. Right-hand rule (or counter-clockwise) is used in OpenGL. The normal is pointing outwards, indicating the outer surface (or front-face).

4. Texture T=(s, t): In computer graphics, we often wrap a 2D image to an object to make it seen realistic. A vertex can have a 2D texture coordinates (s, t), which provides a reference point to a 2D texture image.

4.5.4 Pixel vs. Fragment

Pixels refer to the dots on the display, which are aligned in a 2-dimensional grid of a certain rows and columns corresponding to the display's resolution. A pixel is 2-dimensional, with a (x, y) position and a RGB colour value (there is no alpha value for pixels). The purpose of the Graphics Rendering Pipeline is to produce the colour-value for all the pixels for displaying on the screen, given the input primitives. In order to produce the grid-aligned pixels for the display, the rasterizer of the graphics rendering pipeline, as its name implied, takes each input primitive and perform raster-scan to produce a set of grid-aligned fragments enclosed within the primitive.

A fragment is 3-dimensional, with an (x, y, z) position. The (x, y) are aligned with the 2D pixel-grid. The z-value (not grid-aligned) denotes its depth. The z-values are needed to capture the relative depth of various primitives, so that the occluded objects can be discarded (or the alpha channel of transparent objects processed) in the output-merging stage. Fragments are produced via interpolation of the vertices. Hence, a fragment has all the vertex's attributes such as colour, fragment-normal and texture coordinates.

4.6 OpenGL Applications

OpenGL aims at drawing 2D or 3D object into a frame buffer. The object is defined as a series of vertices or pixels, used to describe geometric objects and images, respectively. Then, OpenGL performs a data conversion to pixels with some processing, and these pixels can form the eventual display graphics in the frame buffer. All OpenGL interfaces are open and can be applied to various hardware platforms and operating systems. Then, users can create static and dynamic 3D colour images of high-quality which close to the ray tracing with effectively employ OpenGL. So, the application features of OpenGL are:

1. Portability. OpenGL is a software interface which is independent of the hardware platform. Intuitively, source code without modifications, can be run on different operating systems of personal computers and workstations.

2. Offline programming. The working mechanism of OpenGL is client/server mode; it is transparent to the network. So OpenGL is convenient to operate in a remote network environment.

3. Dynamic link. In Visual C++ 6.0, we can compile the dynamic link library for other procedure calls through the interface of OpenGL and the Windows system using the MFC class library.

4. Cost and efficiency. Owing to the enhancement in hardware performance and the development of the operating system, the overall performance of computation has increased over early workstations. Because OpenGL has been integrated into Windows, users either develop OpenGL application procedures in the Windows environment, or can easily transplant procedures of existing workstations onto Windows. Therefore, it is convenient to achieve interactive and high-quality 3D graphics based on Visual C++ and the OpenGL graphics library on a PC.

The next section will discuss the relationship between OpenGL and SVG, and the reason why cannot export the rendering result of OpenGL to SVG directly, and it is necessary to develop a new 3D library for integrating 3D into SVG.

4.7 OpenGL and SVG

Through the introduction of OpenGL, the necessary features of a 3D graphics presentation framework should be:

1. Defining and developing 3D Primitive Geometries.

2. Transforming 3D objects in 3D space.

3. Illuminating and shading the 3D object.

4. Adding texture to 3D object to enhance the realistic.

5. Adding 3D object to SVG file which can be rendered directly within a standard web browser.

In order to integrate all those features into SVG, there are two options:

1. Using OpenGL to process the 3D models, and export the render result to SVG directly.

2. Develop a proprietary 3D graphics processing library for SVG, using this 3D library to implement all 3D processing, and generate an SVG file according to the process result.

Table 4.2 Features supported by SVG and OpenGL

Features	SVG	OpenGL
Geometry	2D	2D, 3D
Transformation	2D	2D, 3D
File format	XML based Vector Graphic	Pixel
File size	Small	Big
Browser support	Supported by major browser without plug-in	Cannot be render on web browser
Shading method	Using filter to achieve 2D illumination	Flat shading, Gouraud shading and Phong shading
Texture mapping	Add image on 2D shape	Add texture on 3D object

(Table 4.2) shows features supported by SVG and OpenGL. OpenGL is a 2D and 3D graphical library, the render results of OpenGL are colour, depth, and depth/stencil in frame buffer; SVG is an XML-based vector image format for two-dimensional graphics. SVG images and their behaviours are defined in XML text files. There is no direct support to export OpenGL rendering result to SVG. Therefore, work needs to be done to implement the OpenGL render result in SVG. In order to integrate 3D with SVG, it is necessary to develop a new 3D library from scratch; this 3D library will fulfil all 3D model definitions, 3D transformations, and other 3D modelling features. The final result will be used to create an SVG file, and will be rendered on the web browser directly. All those works will be introduced Chapter 5 in details. This is one of the new technology contributions of this PhD project.

4.8 Summary

SVG is an XML based markup language used to describe and integrate vector graphics, raster graphics and text. The language contains ways to draw vector objects (lines, polygons), raster images and text in various colours and styles on a specified canvas area. SVG's rich visualization options and the support of interactivity make it a natural candidate for providing graphics and interactive examples in different areas.

SVG can be used in a variety of scenarios. SVG can be used for design, GIS and mapping, embedded systems, location-based services (such as traffic and weather reports, mapping and positioning, navigating etc.), animated picture messaging, multimedia messaging, animation and interactive graphics, entertainment, e-Business, and user interfaces information. Some work in these areas has been reviewed and the current state of this technology is also described and assessed.

Despite the advantages of SVG, it is still a technology which only supports 2D graphics. In order to use SVG to present 3D graphics for web-based application, there are still many problems to be solved, such as: the addition of depth information, generation of a realistic shading model, and the application of texture to 3D models.

OpenGL is a fully functional primitive-level API that allows the programmer to efficiently address and take advantage of graphics hardware. Many high-level libraries and applications make use of OpenGL due to its performance, ease of programming, extensibility, and widespread support. Since there is no direct support to export an OpenGL rendering result to SVG. Thus, it is necessary to develop a new 3D graphic library to define 3D primitives, implement 3D transformation, projection, illumination, texture mapping, and create an SVG file accordingly. The new 3D graphic library will be installed on the server side. All 3D graphics processing is carried on the server side, then the result will be sent to client side, and the 3D scene will be rendered on client’s web browser directly.

The next chapter will go on to focus on the new framework-SVG GL for web-based 3D presentation.

Chapter 5 A New Framework-SVG GL for Web-Based Graphical Presentation

5.1 Introduction

As presented in the Chapters 4, SVG is a well-known technique used for describing 2D vector graphics in XML. SVG is well suited to playing a major role in 2D graphics rich environments. It can be used for design, location-based services, animated picture messaging, multimedia messaging, animation and interactive graphics, entertainment, and graphic user interfaces. When it comes to 3D graphics applications, there are limited successful works. As far as the author is aware, the only reported work is applying JavaScript to implement the SVG GL. Most research reports the use of JavaScript to implement 3D operations for SVG (Lindsey, 2003; Tautenhahn, 2002). The benefit of using JavaScript is that it is integrated within the webpage; it provides flexibility to manipulate the object defined in SVG. The drawback is JavaScript is a scripting language; it is not efficient when it come to heavy arithmetic calculations that are the nature of 3D graphics. And it is impossible to implement texture mapping and complex illumination models. Hence, some researchers still don't believe that SVG will be suitable for 3D graphics (Peter, 2011; Tautenhahn, 2002).

However, it is important to implement 3D for web-based applications. As discussed above, SVG is developed by W3C. SVG integrates and leverages other W3C standard technologies already familiar to web programmers: DOM, JavaScript, and CSS. SVG is supported natively by the most current versions of the major web browsers, and it is resolution independent. Since SVG has many advantageous features that would be highly beneficial in the field of 3D graphics for web-based applications, it is worth further research to integrate 3D with SVG, which can benefit 3D graphics for web-based applications.

As mentioned at the beginning of Chapter 4, SVG is an XML-based vector image format for 2D graphics. SVG images and their behaviours are defined in XML text files. But the render results of the existing 3D library, such as OpenGL, are colour, depth, and depth/stencil in frame buffer, and cannot be export to SVG. So a new framework has to be developed to use SVG to present 3D graphics for web-based application. A new framework-SVG GL for 3D model creations and manipulations in a web browser is proposed and developed in this chapter. The new framework should have the following functions:

1. Defining and developing 3D models of primitive geometries.

2. Defining and developing 3D models through sweeping.

3. Creating 3D free-form models by using Bezier surface.

4. Generating 3D models through point clouds.

5. Transforming 3D models in 3D space.

6. Projecting 3D models onto 2D screen.

7. Illuminating and shading the 3D models.

8. Adding texture to 3D models to enhance their realistic.

9. Adding 3D models to SVG file that can be rendered directly in a standard web browser.

In this chapter, the fundamental structure of the new framework-SVG GL will be firstly proposed; the algorithms for geometrical transformation and projection will be secondly explained; then the different 3D modelling of primitive geometries, sweeping, Bezier surface and point clouds in the SVG GL will be developed. More advanced 3D graphics technologies, such as illumination, shading and texture mapping will be proposed in Chapter 6, and Chapter 7.

5.2 Proposition and Design of a New Framework-SVG GL for Web-Based Graphical Presentation

The structure of the new framework-SVG GL is described in Figure 5.1. There are 7 components in this new framework.

1. 3D primitive models: define a set of 3D primitives as the fundamental building block in SVG GL.

2. 3D complex models: provide more complex 3D modelling methods in SVG GL.

3. Geometrical transformations: provide methods of changing the shape and position of objects.

4. Perspective projections: transform points in 3D space to a point into 2D space.

5. Shading and illumination: determine the colour of a surface of an object based on the interaction of light and surface.

6. Texture mapping: maps an image, onto a 3D surface.

7. SVG file generator: transfers the 2D image to the viewport of SVG for final rendering.

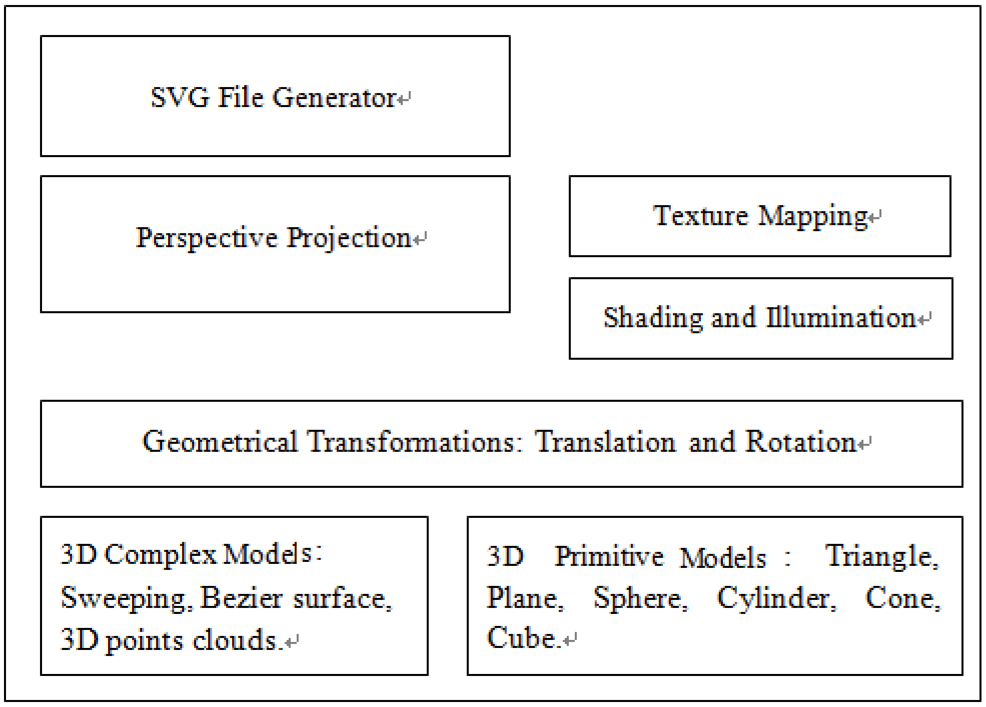

SVG File Generator
Perspective Projection
Texture Mapping
Shading and Illumination
Geometrical Transformations: Translation and Rotation
3D Complex Models: Sweeping, Bezier surface, 3D points clouds.
3D Primitive Models : Triangle, Plane, Sphere, Cylinder, Cone, Cube.

Figure 5.1 The new framework-SVG GL

In the new proposed the SVG GL, the following aspects of a 3D model are specified individually: a 3D model in the world space; a view volume of the camera or observer; a projection onto a projection plane; and a viewport on the SVG file. World space is the base reference system for the overall model, to which all other model coordinates are related. The view volume of the camera is the 3D volume seen by camera. The Viewport is a subset of the screen space where the model is to be displayed. Typically the viewport will occupy the entire screen window, or even the entire screen, but it is also possible to set up multiple smaller viewports within a single screen window. Conceptually, an object in the 3D world space is defined firstly. Then the object is transformed from the world space to the camera space. And then the content in the camera's view volume is projected onto the projection plane. Finally, the projection plane is mapped onto the viewport on the SVG file for display. Figure 5.2 shows this conceptual model of the 3D viewing process in the SVG GL.

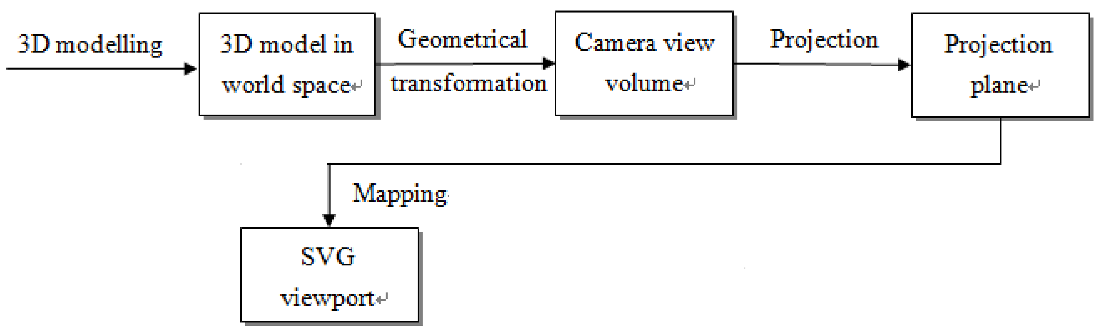


Figure 5.2 3D view processing in the SVG GL

There are four stages for 3D view processing in the SVG GL. Stage 1, 3D modelling: provides an internal mathematical representation of any 3D models that are eventually to be imaged. Stage 2, geometric transformation: transforms the 3D models from world space to camera space. Stage 3, projection: converts 3D coordinates onto a 2D projection plane. Stage 4, mapping: transfers the 2D image from the projection plane to the viewport of SVG for final rendering.

Stage 1: 3D Modelling

The 3D Modelling system needs to support the concept of a geometric coordinated system and provide some way of describing the geometry of the 3D object to be imaged in the world space. For example, a sphere in 3D space consists of the set of all points in 3D space at a fixed distance r from a central point P and can be described by the following Equation:

$$\mathrm{V} = \frac{3}{4}\pi \mathrm{r}^3 \qquad (5.1)$$

The procedure of 3D modelling in the SVG GL is shown in Figure 5.3. There are three steps in stage 1:

Step 1: A 3D model is defined by the primitives, sweeping, Bezier surface and point clouds provided in the SVG GL.

Step 2: The vertexes of the 3D model are calculated by the definition of the 3D model.

Step 3: The fragment triangles that consist the 3D model are generated based on the vertexes of the model.

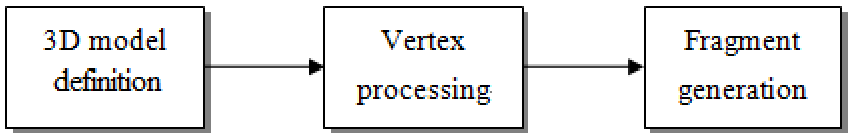


Figure 5.3 3D modelling in the SVG GL

3D primitives are the foundation for 3D modelling in the SVG GL. Primitives defined in the SVG GL are:

1. Triangle.
2. Plane.
3. Sphere.
4. Cylinder.
5. Cone.
6. Cube.

And some more complex 3D model methods are also defined, including:

1. Sweeping, including extrusion and revolution.
2. Bezier surface.
3. 3D points clouds.

The detail of primitives in the SVG GL will be discussed in Section 5.3.

Stage 2: Geometrical Transformation

After the definition of 3D model, the 3D model will be transformed from world space to camera space by geometrical transformations. In this section, the math behind geometrical transformations will be introduced first; and then the geometrical transformations implementation in the SVG GL will be developed.

The basic purpose of geometrical transformations is to provide methods of changing the shape and position of objects (Belongie, 2002), but the use of these transformations is pervasive throughout computer graphics. In fact, geometrical transformations are arguably the most fundamental mathematical tool for computer graphics.

A transformation on $\mathcal{R}^3$ is any mapping $\boldsymbol{F}: \mathcal{R}^3 \longmapsto \mathcal{R}^3$. That is, each point $\boldsymbol{P} \in \mathcal{R}^3$ is mapped to a unique point, ***F(P)***, also in $\mathcal{R}^3$, $\mathcal{R}^3$ is the usual 3D Euclidean space consisting of point (x, y, z).

Let $\boldsymbol{F}$ be a transformation. For a linear transformation, the following two conditions hold:

1. For all $a \in \boldsymbol{\mathcal{R}}$ and $\boldsymbol{P} \in \boldsymbol{\mathcal{R}^3}$, $\boldsymbol{F}(a\boldsymbol{P}) = a\boldsymbol{F}(\boldsymbol{P})$.
2. For all $\boldsymbol{P}, \boldsymbol{Q} \in \boldsymbol{\mathcal{R}^3}$, $\boldsymbol{F}(\boldsymbol{P} + \boldsymbol{Q}) = \boldsymbol{F}(\boldsymbol{P}) + \boldsymbol{F}(\boldsymbol{P})$.

A transformation can act on a single point at a time, and it can also act on arbitrary geometric objects since the geometric object can be viewed as a collection of points and, when the transformation is used to map all the points to a new location, this changes the form and position of the geometric object.

Affine transformations are the most fundamental transformations used in computer graphics (Buss, 2003). Affine transformations are transformations that preserve points, lines, and planes and parallelism (maps parallel lines to parallel lines). Also, affine transformations preserve ratios of distances between points lying on a string line, but do not preserve the angles between lines or distances between points. To be more specific, for a point, an affine transformation can be represented in the form: ***F (P) =A (P) + v***, where $\boldsymbol{A}$ is the linear transformation, and $\boldsymbol{v}$ is a vector in $\boldsymbol{\mathcal{R}^3}$. It can be seen that any affine transformation is the composition of a linear transformation and a translation. In computer graphics, the most often used affine transformations include translation, rotation, and scaling. Several affine transformations can be combined into a single overall affine transformation.

Translation and rotation are also known as rigid-body transformations (Eggert, 1997). The combination of translations and rotations cannot change the shape or volume of an object; they can only alter the object's location and orientation.

A translation is a transformation that displaces points in 3D space by a fixed distance in a given direction. A translation can be represented as: $\boldsymbol{F}(\boldsymbol{P}) = \boldsymbol{P} + \boldsymbol{d}$ where $\boldsymbol{P}$ is point in 3D space, $\boldsymbol{d}$ is a specified displacement vector in $\boldsymbol{\mathcal{R}^3}$. Translation is denoted as $\boldsymbol{T_d}$, thus $\boldsymbol{T_d}(\boldsymbol{P}) = \boldsymbol{P} + \boldsymbol{d}$.

The translation transformation can also be represented in the matrix form as:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \boldsymbol{T} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{5.2}$$

where $\boldsymbol{T}$ is called the translation matrix, and can be expressed as:

$$\boldsymbol{T} = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.3}$$

Translation has 3 degrees of freedom since the 3 components of the displacement vector can be specified arbitrarily. Equation (5.3) translates a point ***P***(x, y, z) by an offset vector $\mathbf{d} = (d_x,\ d_y,\ d_z)$, $\boldsymbol{P'}(x',\ y',\ z')$ is the coordinate of the new point.

A rotation is a transformation that rotates all points on a 3D object around the axis through a fixed angle θ in a given coordinate axis. This transformation is denoted as $\boldsymbol{R}_{\theta X/Y/Z}$, where *X, Y, Z* represents the *x, y, z* coordinate axis, θ is the rotation angle.

The rotation transformation can be represented in matrix form as:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \boldsymbol{R}_z \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{5.4}$$

where $\boldsymbol{R}_z$ is the rotation matrix for rotation around the *z*-axis, and can be expressed as:

$$\boldsymbol{R_z} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.5}$$

Equation (5.5) rotates point ***P***(x, y, z**)** around ***z***-axis by an angle θ, $\boldsymbol{P'}(x',\ y',\ z')$ is the coordinate of the new point. The rotation matrices for rotation around *x*- and *y*- coordinate axis can be derived through an identical argument. The *x* values are unchanged for rotation about *x* axis; and the *y* value are unchanged for rotation about *y*-axis. So the rotation matrices are

$$\boldsymbol{R}_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.6}$$

for rotation around the ***x***-axis; and

$$\boldsymbol{R}_y = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.7}$$

for rotation around the *y*-axis.

There are 3 degrees of freedom corresponding to the ability to rotate independently about coordinate axes. Since the matrix multiplication is not commutable, rotation about the *x*- axis by an angle θ followed by rotation about *y*-axis by an angle ϕ does not give the same result as the one that obtained by reversing the order of the rotations.

However, apart from rigid-body transformations, there is also some non-rigid body transformations used in computer graphics. Scaling is an affine non-rigid body transformation which scales points by s_x along the $\boldsymbol{x}$ axis, s_y along the $\boldsymbol{y}$ axis and $\boldsymbol{s_z}$ along $\boldsymbol{z}$ axis respectively. If $s_x = s_y = s_z = 1$**,** it is called a uniform scaling, otherwise it is called a non-uniform scaling.

A scaling transformation for independent scaling along each coordinate axes can be specified as: $x' = s_x x$, $y' = s_y$, $z' = s_z z$. These 3 Equations can be combined to express the generic scaling transformation as:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \boldsymbol{S} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{5.8}$$

where $\boldsymbol{S}$ is called the scaling matrix, it can be written as $\boldsymbol{S}(s_x, s_y, s_z)$, and can be expressed as:

$$\boldsymbol{S} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.9}$$

All these transformation matrices have an inverse. The inverse of a translation $\boldsymbol{T_d}$, is a translation in the opposite direction $\boldsymbol{T_{-d}}$. The inverse of a rotation is the same rotation with the opposite sign on the angle. The inverse of scale $\boldsymbol{S}(s_x, s_y, s_z)$ is $\boldsymbol{S}(1/s_x, 1/s_y, 1/s_z)$. For a series of matrices $\boldsymbol{M} = \boldsymbol{M_1 M_2} \cdots \boldsymbol{M_n}$, the inverse matrix is $\boldsymbol{M^{-1}} = \boldsymbol{M_n^{-1} M_{n-1}^{-1}} \cdots \boldsymbol{M_1^{-1}}$.

Any number of rotation, scaling, and translation matrices can be multiplied together. The result always has the form

$$\boldsymbol{M} = \begin{bmatrix} \boldsymbol{R} & \boldsymbol{T} \\ \boldsymbol{0}_3^T & \boldsymbol{1} \end{bmatrix} \tag{5.10}$$

The 3×3 upper-left submatrix $\boldsymbol{R}$ gives the aggregate rotation and scaling, whereas $\boldsymbol{T}$ is a 3D translation vector that gives the subsequent aggregate translation. $\boldsymbol{0_3}$ is a 3D zero vector

In the SVG GL, geometrical transformations include translation and rotation (Figure 5.4).

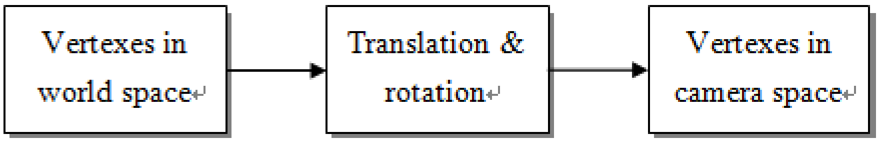


Figure 5.4 Transform from world space to camera space

Geometrical transformations are implemented in a 3DMath Library; and the following classes are defined in this Math Library:

1. Matrix3, a matrix class for 3x3 matrix operation.
2. Matrix4, a matrix class for 4x4 matrix operation.
3. Vector2, a vector class for 2D vector operation.
4. Vector3, a vector class for 3D vector operation.;
5. Vector4, a vector class for 4D vector operation.

Stage 3: Projection

After the geometrical transformations, the vertexes in camera space will be projected onto projection plane by projections transform.

In general, projections transform points in a coordinate system of dimension $\boldsymbol{n}$ into points in a coordinate system of dimension less than $\boldsymbol{n}$ (Kennedy, 2001). The projection of a 3D object is defined by straight projection rays (projectors) emanating from a center of projection, passing through each point of the object, and intersecting a projection plane to form the projection.

The class of projections used in the SVG GL is known as planar geometric projection because the projection is onto a plane rather than some curved surface. Projection onto a curved surface will cause the distortion of the 3D model according to the curved surface. Since the purpose of the SVG GL is to create realistic 3D model for web-based application, so only planar geometric projection is used in the SVG GL. Planar geometric projection can be divided into 2 basic classes: perspective and parallel (Carlbom, 1978). The distinction between them is in the relation of the center of projection to the projection plane. If the distance from the center of projection to the projection plane is finite, then the projection is perspective; if the distance is infinite, the projection is parallel.

1. Perspective projection

Perspective projection was originally discovered for applications in drawing and painting (Coxeter, 1974). An important principle in the classic theory of perspective projection is the notion of a ‘vanishing point’ - the intersection of the projections of a set of parallel lines in space onto the projections plane. In computer graphics applications, it is able to avoid all considerations of vanishing points and similar factors. Instead, an object is placed in 3D space, a projection center (camera position) is chosen, and the correct perspective transformation is mathematically calculated to create the scene as viewed from the projection centre.

Perspective projection is used to create the view when the camera or eye position is placed at a finite distance from the scene. The use of perspective means that an object will appear larger as it moves closer to the viewer. Perspective is useful for giving the viewer the sense of being ‘in’ a scene because a perspective projection shows the scene from a particular viewpoint. Perspective is heavily used in entertainment applications, where it is desired to give an immersive experience; it is particularly useful in dynamic situations in which the combination of motion and correct perspective gives a strong sense of the three-dimensionality of the scene. Perspective is also used in applications as diverse as architectural modelling to show the view from a particular viewpoint.

For simplicity, the projection center is placed at the origin looking down the negative z-axis. It is a model of image formation that projects a 3D scene towards a single point – the projection center. The image is not defined at the projection center, but rather it is defined on a plane, called the projection plane. The projection plane is perpendicular to the camera z axis.

This perspective projection model is shown in Figure 5.5. The model consists of a plane (projection plane/image plane) and a 3D point $\boldsymbol{P}$. Point $\boldsymbol{O}$ is the projection center, f is the distance between the projection plane and the projection center, and is called focal length. $\boldsymbol{P}'$ is the projection of point $\boldsymbol{P}$ on the projection plane. The line through $\boldsymbol{O}$ and perpendicular to the image plane is the optical axis.

Using the triangular mathematic operation, the relation between the coordinate of point $\boldsymbol{P}$ and point $\boldsymbol{P}'$ is:

$$\begin{bmatrix} x' \\ y' \\ z' \\ w \end{bmatrix} = M_p \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{5.11}$$

where (x, y, z, 1) is the homogenous coordinate of a point $\boldsymbol{P}$ in 3D coordinate system, (x', y', z', w) is the homogenous coordinate of perspective projection of point $\boldsymbol{P}$ in 3D coordinate system, and $(x'/w, y'/w)$ is the coordinate of perspective projection of point $\boldsymbol{P}$ on the projection plane.

M_p is the perspective projection matrix, it can be expressed as:

$$\boldsymbol{M_p} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & f & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{5.12}$$

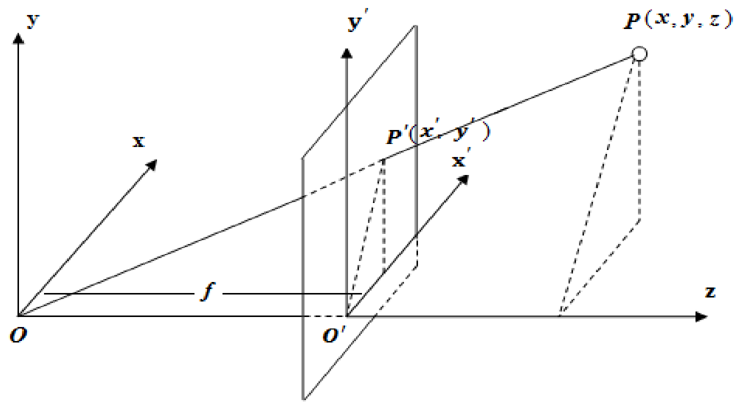


Figure 5.5 Perspective projection

Perspective projection has the following properties:

(1) The size of the projection image is inversely proportional to the distance from the object to the image plane.

(2) The smaller the $\boldsymbol{f}$ (focal length), the wider the view field.

(3) Line is preserved, but distances and angles are not preserved.

(4) Parallel lines in space project onto lines that on extension intersect at a single

point in the image plane called the vanishing point.

(5) The vanishing points of all the lines that lie on the same plane form vanishing line.

2. Parallel projection

The parallel projection mentioned is an orthographic parallel projection in which the direction of the projection is perpendicular to the projection plane (Maynard, 2005; Riley, 2006). In this type of projection, the projection plane is perpendicular to a principal axis, which is therefore the direction of the projection.

Unlike the perspective projection described earlier, orthographic projection does not cause closer objects to appear larger and distant objects to appear smaller. For this reason, orthographic projection is generally preferred for applications such as architecture or engineering applications, including CAD and CAM since the parallel projection is better at preserving relative sizes and angles.

An orthographic projection is shown in Figure 5.6. In this model, the projection ray is orthogonal to the image plane and parallel with the z axis. $\boldsymbol{P}$ is a 3D point, $\boldsymbol{P}'$ is the projection of point $\boldsymbol{P}$ on the image plane. The relation between the coordinate of point $\boldsymbol{P}$ and point $\boldsymbol{P}'$ is:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = M_o \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{5.13}$$

where (x, y, z) is the coordinate of a point $\boldsymbol{P}$ in the 3D coordinate system, and (x', y') is the orthographic projection of point $\boldsymbol{P}$ on plane $z' = 0$, M_o is the orthographic projection matrix, it can be expressed as

$$M_o = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.14}$$

An orthographic projection's properties are:

(1) Parallel lines project to parallel lines.

(2) Size does not change with the distance from the camera.

(3) Angles are not preserved.

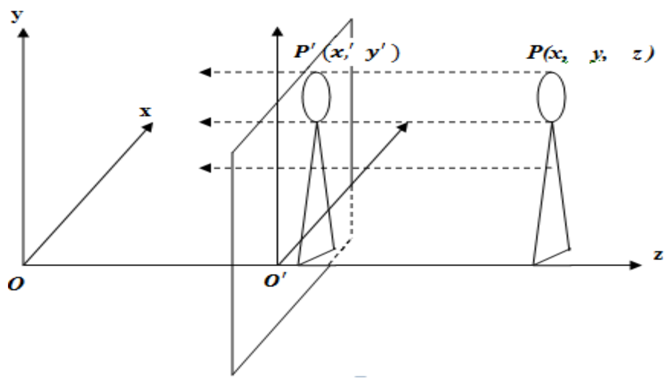


Figure 5.6 Orthographic projection

The purpose of the SVG GL is to provide a realistic interactive 3D model for web-based application, so only perspective projection is used (Figure 5.7).

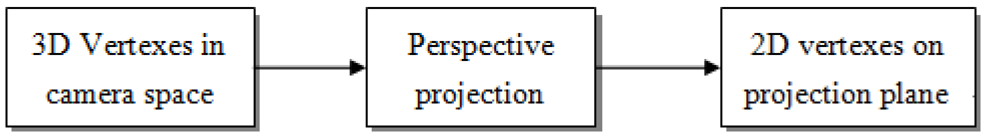


Figure 5.7 Projection from 3D camera space to 2D projection plane

Combining the transformation matrix $\boldsymbol{M}$ in Equation (5. 10) with projection matrix, the final perspective projection result can be described as:

$$\begin{bmatrix} x' \\ y' \\ z' \\ w \end{bmatrix} = \boldsymbol{M_p M} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{5.15}$$

where $\boldsymbol{M_p}$ is a projection matrix, $\boldsymbol{M}$ is the concatenated matrix of translation, rotation and scale transformation.

3. Camera View Volume

The role of the camera in a 3D computer graphics system is to provide both a point of view from which to render an image and the basic parameters of the mathematical projection that will be used to form the virtual image. The camera's position and orientation are specified as part of the scene description. It is typical for the camera to be positioned in the global coordinate system, usually with some positioning controls that correspond to the operation of a real studio camera.

Theoretically, cameras can have any projection characteristics, corresponding to the variety of lens type. However, practical 3D graphics implementations usually implement only the standard parallel or perspective projections that are common in architectural and design drafting.

In the SVG GL, the camera is implemented by a camera class. This class has the following properties:

(1) Position, the centre position of the camera.

(2) Direction, the facing direction of the camera.

(3) Focus, the focus length of the camera.

(4) Near, the distance from the camera to the near clipping plane.

(5) Far, the distance from the camera to the far clipping plane.

(6) Fov, view angle, in degrees.

Stage 4: Mapping

A viewport is a 2D rectangle on screen defining where the image will appear. Mapping is simply the process of transforming 2D scene on project plane in world space onto viewport on screen or device space (Figure 5.8). In particular, objects inside the clipping window are mapped to the viewport. The viewport is displayed in the interface window on the screen. In other words, the clipping window is used to select the part of the scene that is to be displayed. The viewport then positions the scene on the output device.

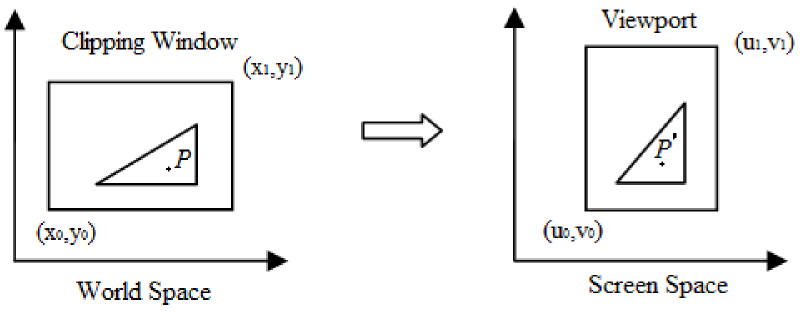


Figure 5.8 Viewport transformation

The relation between the coordinate of point ***P*** and point ***P'*** in Figure 5.8 is:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{u_1-u_0}{x_1-x_0} & 0 & -x_0\frac{u_1-u_0}{x_1-x_0}+u_o \\ 0 & \frac{v_1-v_0}{y_1-y_0} & -y_0\frac{v_1-v_0}{y_1-y_0}+v_o \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{5.16}$$

where (x, y) is the coordinate of a point $\boldsymbol{P}$ on the 2D projection plane, and (x', y') is the coordinate of point $\boldsymbol{P}'$ on the viewport,

$$\begin{bmatrix} \frac{u_1-u_0}{x_1-x_0} & 0 & -x_0\frac{u_1-u_0}{x_1-x_0}+u_o \\ 0 & \frac{v_1-v_0}{y_1-y_0} & -y_0\frac{v_1-v_0}{y_1-y_0}+v_o \\ 0 & 0 & 1 \end{bmatrix}$$ is the viewport transformation matrix.

The viewBox attribute in SVG is used to define the viewport in the SVG GL. The value of the viewBox attribute is a list of 4 numbers min-x, min-y, width and height. Min-x, min-y defines the coordinate of the lower left corner of the viewport, and the width and height define the width and height of the viewport. The viewport transformation is used to map vertexes on projection plane on the SVG viewbox (Figure 5.9)

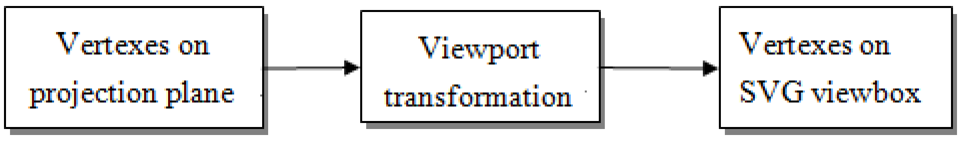


Figure 5.9 Transform from projection plane to SVG viewport

5.3 3D Modelling of Primitive Geometries in the SVG GL

A model is a representation of some features of a concrete or abstract entity. The purpose of a model of an entity is to allow people to visualize and understand the structure or behaviour of the entity, and to provide a convenient vehicle for ‘experimentation’ and prediction of the effects of inputs or changes to the model (Foley, 2013). In 3D computer graphics, real worlds can be modelled with geometric objects; and modelling is the process of developing a 3D model by using a set of points in 3D space, that are connected by various geometric data such as lines, and polygons.

There are a great variety of geometric objects in 3D world. All these geometric objects that fit well with existing graphics hardware and software have the following 3 features:

(1) The objects are described by their surfaces and can be thought of as being hollow.

(2) The objects can be specified through a set of vertices in 3D.

(3) The objects either are composed of or can be approximated by flat, convex polygons.

Generally there are four popular methods for 3D modelling: polygonal modelling, primitive modelling, NURBS (nonuniform rational B-splines) modelling, and splines & patches modelling. Polygonal modelling is a method of creating a 3D model by connecting line segments through points in a 3D space. Primitive modelling creates geometric primitive such as cube, cone, and sphere firstly, then using those primitives to create complex 3D models. NURBS modelling defines 3D model surface by curves, the curve is created by NURBS. Splines & patches modelling is similar to the NURBS modelling procedure, they depend on curved lines to identify the visible surface. A spline is a curve in 3D space defined by at least two control points. Using splines to create a model is perhaps the oldest, most traditional form of 3D modelling available.

In the SVG GL**,** Polygon modelling method is used for creating 3D models, such as cube, cone, sphere and so on. It is very common for 3D geometric shapes to be modelled firstly as a set of polygons and then mapped to a polygonal 2D to display. The basic display hardware is generally pixel based, but most computers now have special-purpose graphics hardware for processing polygons or, at least, triangles. Polygonal-based modelling is used in nearly every 3D computer graphics system (Wong, 2013). It is a central tool for the generation of interactive 3D graphics and is used for photo-realistic rendering.

Polygon modelling requires the application to either specify simple planar polygons or triangles to connect a list of vertices. For a simple polygon specified with more than 3 vertices, if the vertices do not lie in the same plane, there will be no simple way to define the interior of the object, and then the results of rasterizing the polygon are not guaranteed to be what the developer might desire. Since triangles are always flat, either the modelling system is designed to always produce triangles, or the system provides a method to divide, or triangulate an arbitrary polygon into a triangle mesh. The same procedure can be used to represent a curved object, such as a sphere that can be approximated by a small, flat polygon.

All 3D models in the SVG GL are specified through a set of vertices; and then lines or triangles are used to connect the vertices. While vertices define the shape of the object, triangles are the shapes onto which the shade, light, and texture are put. The results of the modelling process are a set of vertices that specify a group of geometric objects used by the rest of the graphics system.

Primitives are selected from a universe of possible shapes. The commonly used geometric primitives include point, line, plane, circle, triangle, and spline curves.

But the primitives defined in the SVG GL are slight different with the primitives mentioned above. Such primitives include triangle, plane, sphere, cylinder, cone, and cube. These are considered to be primitives in 3D modelling because they are the building blocks for many other shapes and forms (Watt, 1999). A primitive is instantiated by assign values to certain parameters. The SVG file created by the SVG GL for some of the primitives can be found in Appendix A.

1. Triangle: Triangle is the elementary primitive in 3D graphics, which can be used to model all other 3D objects. A triangle specified by 3 vertices that form a closed area in 3D space. The vertices of a triangle can be defined with a 3D coordinate (*x*, *y*, *z*). Casting a ray from each vertex of the triangle to the projection center, each ray intersects with the projection plane. By using Equation (5.12), a triangle in 3D world space is projected to a triangle in 2D window space, and then mapped to the SVG viewport (Figure 5.10).

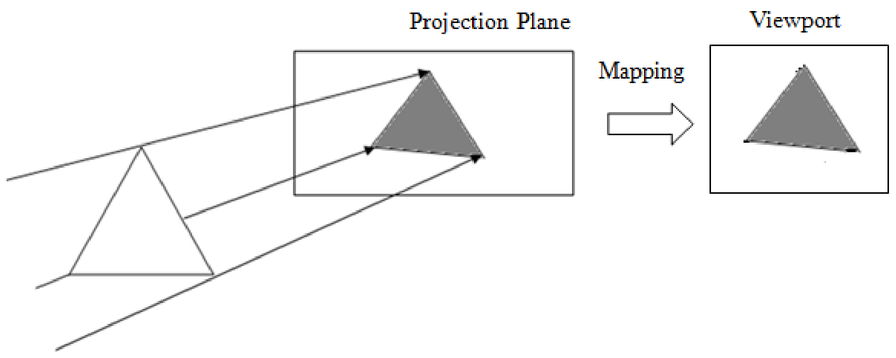


Figure 5.10 Triangle in the SVG GL

In the SVG GL, a triangle object can be created by the Triangle class constructor as Triangle ($\boldsymbol{v_0}, \boldsymbol{v_1}, \boldsymbol{v_2}$), where $\boldsymbol{v_0}, \boldsymbol{v_1}, \boldsymbol{v_2}$ are non-collinear points in 3D space that specify 3 vertices of the triangle.

2. Plane: The plane defined in the SVG GL is not the same as the one defined in geometry, it is actually a rectangle. A plane is specified by 4 vertices; and can be treated as 2 triangles that share 2 common vertices. By mapping the triangle to the SVG viewport, a 2D polygon correspondence to the plane is finally generated (Figure 5.11).

A plane object can be initialized by the Plane class constructor Plane (*Width*, *Height*), where *Width* is the width of the plane, and *Height* is the plane's height. Then the vertices accordingly are (*0, 0, 0*), (*Width, 0, 0*), (*Width, Height, 0*), and (*0, Height, 0*). It shows the default location of a plane object is the original point and *xy*-plane of the coordinate system.

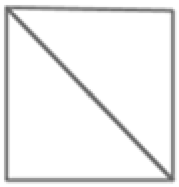


Figure 5.11 Plane in the SVG GL

3. Sphere: In geometry, a sphere can be viewed as the surface formed by rotating a circle about any diameter, a sphere can be defined by:

$$(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 = r^2 \tag{5.17}$$

where (x, y, z) is the point on the surface of a sphere, r is the radius of a sphere, and (x_0, y_0, z_0) is the center of the sphere. This equation shows that a point on the surface of a sphere is at a fixed radial distance ***r*** from its center.

Equation (5.15) can also be converted to a formula in spherical polar coordinates as

$$\begin{cases} x = x_0 + rcos\theta sin\varphi \\ y = y_0 + rcos\varphi \\ z = z_0 + rsin\theta sin\varphi \end{cases} \tag{5.18}$$

where θ is an azimuthal angle ranging from 0 to 2π, ϕ is a polar angle ranging from 0 to π, and r is the radius (Figure 5.12). The surface of a sphere with radius ***r*** can simply be covered by varying θ from 0 to 2π, angle ϕ from 0 to π. By subdividing these ranges into small enough sections, sufficient points on the surface of the sphere can be generated to form the vertices of a polyhedral approximation to the surface, replacing the curved surface with small polygon faces.

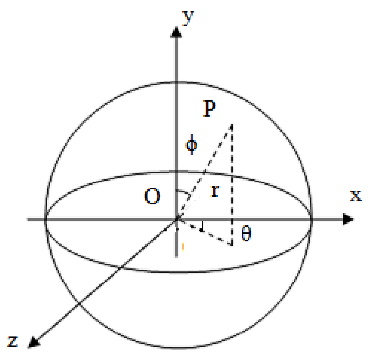


Figure 5.12 Point on the surface of a sphere

In the SVG GL, a sphere can be created by using the Sphere class constructor as Sphere(*r*, *segmentA*, *segmentP*), where ***r*** is the radius, *segmentA* is the number of sections divided along longitude direction, and *segmentP* is the number of sections along latitude direction (Figure 5.13).

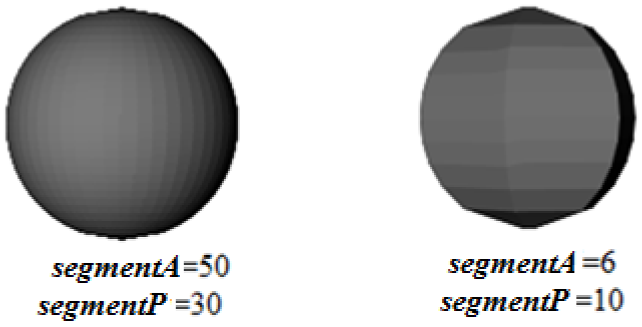


Figure 5.13 Sphere with different ***segmentsA*** and ***segmentsP***

4. Cylinder: Similar to the sphere formula defined in spherical polar coordinates, a cylinder can be defined in cylindrical polar coordinates as:

$$\begin{cases} x = r\cos\theta \\ y = r\sin\theta \quad 0 \le \theta \le 2\pi \\ z = u \qquad\quad 0 \le u \le h \end{cases} \tag{5.19}$$

where *θ* is an azimuthal angle ranging from 0 to 2π, and ***r*** is the radius of the cylinder. By generating angle *θ* from 0 to 2π, and values of ***z*** from 0 to *h* (*h* is the height of the cylinder) while holding ***r*** as a constant value (Figure 5.14), the full curved surface of the cylinder can be found. By taking sufficient subdivisions of *θ*, the curved surface of the cylinder can be approximated with small polygon surfaces.

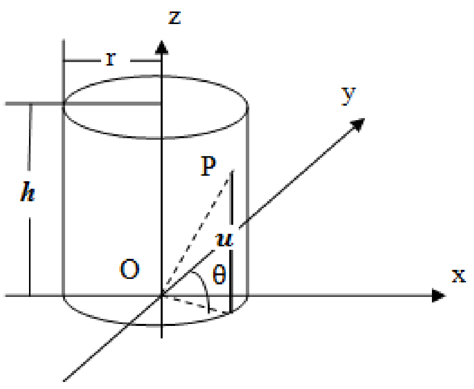


Figure 5.14 Point on the curved surface of a cylinder

Constructor Cylinder(*r, h*, *segmentA*) is used in the SVG GL to create a cylinder object, where *r* is the radius, *h* is the height of the cylinder respectively, and *segmentA* is the number of sections divided along longitude direction (Figure 5.15).

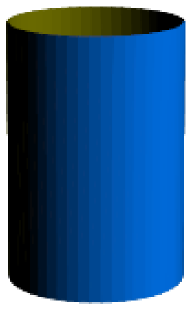


Figure 5.15 A cylinder without top and bottom face in the SVG GL

5. Cone: The parametric Equation of a cone can be defined as

$$\begin{cases} x = \frac{h-u}{h} r\cos\theta \\ y = \frac{h-u}{h} r\sin\theta \quad 0 \le \theta \le 2\pi \\ z = u \qquad\qquad\quad 0 \le u \le h \end{cases} \tag{5.20}$$

where h is the height of the cone, $\boldsymbol{r}$ is the base radius of the cone, and θ is an azimuthal angle ranging from 0 to 2π. By varying angle θ from 0 to 2π, and values of $\boldsymbol{z}$ from 0 to h, all points on the surface of the cone can be found (Figure 5.16). By taking sufficient subdivisions of θ, the curved surface of the cone can be approximated with small polygon surfaces.

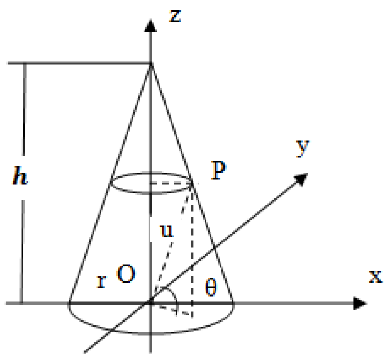


Figure 5.16 Point on the curved surface of a cone

A cone object can be defined as Cone(*r, h, segmentA*) in the SVG GL, where *r* is the base radius, *h* is the height of the cone respectively, , and *segmentA* is the number of sections divided along longitude direction (Figure 5.17).

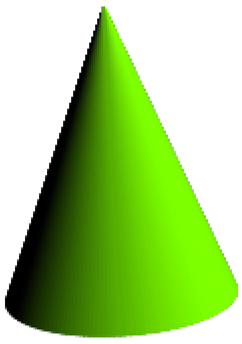


Figure 5.17 Cone in the SVG GL

6. Cube: A cube is specified by 8 vertices; 6 planes are used to connect the vertices, each plane consists with 2 triangles (Figure 5.18).

A cube object can be initialized by the Cube class constructor Cube (*Width, Height, Depth*), where *Width* is the width, *Height* is the height, and *Depth* is the depth of a cube respectively. Then the vertices accordingly are (*0, 0, 0*), (*Width, 0, 0*), (*Width, Height, 0*), (*0, Height, 0*), (*0, 0, -Depth*), (*Width, 0, -Depth*), (*Width, Height, -Depth*), and (*0, Height, -Depth*). It shows the default location of a cube object is the original point and all its 6 sides parallel to different coordinate plane respectively.

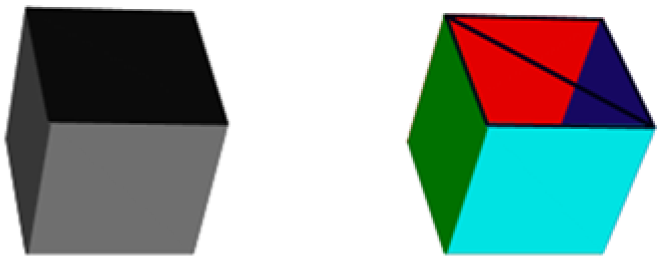


Figure 5.18 Cube rendered with different material

5.4 3D Modelling through sweeping

A sweep object is generated when a space curve ***C***(***s***) is transformed by a transformation rule ***T***(***t***) (Martin, 1989). Curve ***C***(***s***) is referred to as the profile curve. The surface of the sweep object is swept by the profile curve when that curve is transformed by the transformation rule ***T***(***t***). The expression of the surface is simply the product.

Two typical sweep objects are discussed here: extrusion, and revolution. The different between these two types is determined by the transformation rule ***T(t)***

5.4.1 Extrusion

Extrusion is a kind of sweeping, it is defined by translating a space curve along a linear trajectory normal to the plane of the shape to create 3D object (Shum. 2001). The trajectory curve can be specified by a vector $\boldsymbol{t}$= ($\boldsymbol{t_x}, \boldsymbol{t_y}, \boldsymbol{t_z}$). When a space curve **C**(***s***) is translated along a vector ***v***, then the transformation matrix $\boldsymbol{E}(\boldsymbol{t})$ is given by

$$\boldsymbol{E}(\boldsymbol{t}) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.21}$$

If a space curve is expressed by **C**(***s***), where $\mathbf{0} \leq \boldsymbol{s} \leq \mathbf{1}$, then the surface of extrusion has the form **C**(***s, t***) = **C**(***s***)***E***(***t***), The surface of the extrusion can be covered by varying ***s*** from 0 to 1. By subdividing these ranges into small enough sections, sufficient points on the surface of the extrusion object can be generated, replacing the surface with small polygon faces.

Figure 5.19 shows that an elliptic tube is created by extruding an ellipse on the *xy*-plane along ***y***-axis.

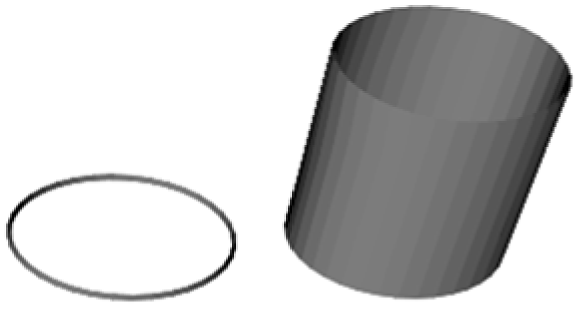


Figure 5.19 Extruding a ellipse along ***y***-axis

5.4.2 Revolution

Revolution is a special case of sweeping. It is obtained when a space curve **C**(***s***) is rotated about an axis in space (Kent, 1992; Blundell, 2008). The rotation angle can be 2π or less. A general rotation in 3D is fully specified by the axis of rotation and the rotation angle. If the rotation angle is $\boldsymbol{\theta}$ and rotation axis is ***r***, then the rotation matrix ***R***(***θ***) is given by

$$\begin{bmatrix} r_x^2+\cos\theta(1-r_x^2) & r_xr_y(1-\cos\theta)-r_z\sin\theta & r_xr_z(1-\cos\theta)-r_y\sin\theta \\ r_xr_y(1-\cos\theta)+r_z\sin\theta & r_y^2+\cos\theta(1-r_y^2) & r_yr_z(1-\cos\theta)-r_x\sin\theta \\ r_xr_z(1-\cos\theta)-r_y\sin\theta & r_yr_z(1-\cos\theta)+r_x\sin\theta & r_z^2+\cos\theta(1-r_z^2) \end{bmatrix} \tag{5.22}$$

If a space curve is expressed by $\mathbf{C}(\boldsymbol{s})$, then the surface of revolution has the form $\boldsymbol{C}(\boldsymbol{s},\ \boldsymbol{\theta})=\boldsymbol{C}(\boldsymbol{u})\boldsymbol{R}(\boldsymbol{\theta})$, where $\mathbf{0} \le \boldsymbol{s} \le \mathbf{1}$ and $0 \le \theta \le 2\pi$. The surface of the revolution can be covered by varying $\boldsymbol{s}$ from 0 to 1, angle $\boldsymbol{\theta}$ from 0 to 2π. By subdividing these ranges into small enough sections, sufficient points on the surface of the revolution object can be generated, replacing the curved surface with small polygon faces.

Figure 5.20 shows a chess pieces model is created by revolving the outline on the left side around $\boldsymbol{y}$-axis.

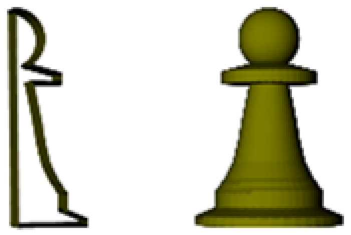


Figure 5.20 A chess pieces generated by revolution

5.5 Bezier Surface

At the lowest level, the smooth surfaces of geometric objects are approximated by triangles. However, many useful surfaces can be described mathematically by a small number of parameters such as a few control points. Saving a few control points for a surface requires much less storage than saving hundreds triangles.

Bezier surface is a typical type of freeform surface. A Bezier surface is defined by a set of control points (Farin, 1996; Gálvez, 2007). The Equation of a Bezier surface defined by *m+1* rows and *n+1* columns of control points is:

$$\boldsymbol{p}(\boldsymbol{s},\boldsymbol{t}) = \sum_{i=0}^{m}\sum_{j=0}^{n}\boldsymbol{B}_{m,i}(\boldsymbol{s})\boldsymbol{B}_{n,j}(\boldsymbol{t})\boldsymbol{p}_{ij} \tag{5.23}$$

where $\boldsymbol{p}_{ij}$ are the set of control points, $\boldsymbol{B}_{m,i}(\boldsymbol{s})$ and $\boldsymbol{B}_{n,j}(\boldsymbol{t})$ are the *i-th* and *j-th* Bezier basis function in the $\boldsymbol{u}$ and $\boldsymbol{v}$ directions, and are defined as follows:

$$\begin{cases} \boldsymbol{B_{m,i}(s) = \frac{m!}{i!(m-i)!} s^i (1-s)^{m-i}} \\ \boldsymbol{B_{n,j}(t) = \frac{n!}{j!(n-j)!} t^j (1-t)^{n-j}} \end{cases} \tag{5.24}$$

A Bezier curve is defined by ***n+1*** control points; it passes through the two extreme points, and uses the interior points to determine its shape. Similarly, a Bezier surface is defined by a grid of (***m+1***)×(***n+1***) controls, it is anchored at the four corner points and uses the other grid point to determine its shape. Closed surface can be formed by setting the last control equal to the first. A curved surface can be created by varying ***s*** from 0 to 1, ***t*** from 0 to 1. By subdividing these ranges into small enough sections, sufficient points on the surface can be generated; replacing the curved surface with small polygon faces (Figure 5.21).

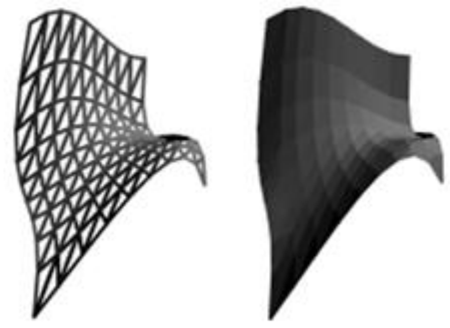
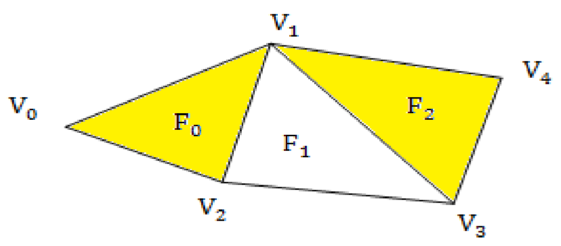


Figure 5.21 Curved surface generated by Bezier surface

5.6 3D Modelling Through 3D Point Clouds

A 3D point cloud is a set of data points in a 3D coordinate system, these points are usually defined by *x, y,* and *z* coordinates, and are used to represent the external surface of an object (Golovinskiy, 2009). These data sets are often very large. They can be used to display objects directly if the graphics system supports point primitives.

Since the smallest primitive in the SVG GL graphics is the triangle, it cannot display point clouds. In order to generate 3D object from point clouds, more structure information have to be added.



Vertex List	Face List
$V_0 = (x_0, y_0, z_0)$	$F_0 = (V_0, V_2, V_1)$
$V_1 = (x_1, y_1, z_1)$	$F_1 = (V_2, V_3, V_1)$
$V_2 = (x_2, y_2, z_2)$	$F_2 = (V_3, V_4, V_1)$
$V_3 = (x_3, y_3, z_3)$	
$V_4 = (x_4, y_4, z_4)$	

Figure 5.22 Structured point clouds

A simple 3D surface is shown in Figure 5.22 consisting of 5 vertices. Besides the vertex-list, there is also a face-list to specify how to connect the vertices by triangle face. The point cloud with face information is called a *structured point cloud*. By using the face-list, a set of triangle meshes will be generated to form a 3D object from the point clouds.

5.7 Summary

Since the render results of the existing 3D library cannot be export to SVG. A new framework has to be developed to use SVG to present 3D model for web-based application.

A new framework-SVG GL was proposed is this chapter. There are four stages in the new framework-SVG GL. Stage 1, 3D modelling: provides an internal mathematical representation of any 3D models that are eventually to be imaged. Stage 2, geometric transformation: transforms the 3D models from world space to camera space. Stage 3, projection: converts 3D coordinates onto a 2D projection plane. Stage 4, mapping: transfers the 2D image from the projection plane to the viewport of SVG for final rendering.

By using the new proposed framework, an object in the 3D world space is defined first. Then the object is transformed from the world space to the camera space. And then the content in the camera's view volume is projected onto the projection plane. Finally, the projection plane is mapped onto the viewport on the SVG file for display.

In order to implement all the 3D graphics operations in the SVG GL:

1. A 3DMath Library in the SVG GL is developed to implement geometric transformations, including translation, rotation, and scaling.

2. A perspective projection is developed in the SVG GL to project 3D models onto 2D projection plane.

3. A viewport transformation is also defined in the SVG GL to transfer the 2D image from projection plane to viewport in SVG.

4. A set of primitives are developed in the SVG GL as the foundation for 3D modelling in the SVG GL, Such primitives include triangle, plane, sphere, cylinder, cone, and cube. And some more complex 3D model methods are also developed, including: extrusion, revolution, Bezier surface and point clouds.

Chapter 6 New Algorithms for Shading in the SVG GL

6.1 Introduction

With the technologies discussed in Chapters 5, a 3D model can be built and rendered. But the result is not really promising since it looks flat and fails to show 3D nature of the object. This appearance is a consequence of the assumption that each surface is lit such that it appears to a viewer in a single colour (Figure 6.1).

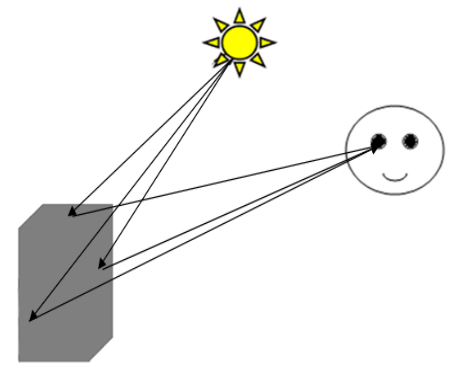


Figure 6.1 Lighting surfaces with single colour

To produce a 3D model that looks more realistically, the model has to be lit and shaded. Figure 6.2 shows two versions of same object (a cuboid), (a) without lighting and (b) with lighting. It can be seen that the unlit cuboid looks no different from a uniformly coloured polygon. The lit cuboid also shows the gradations of colour give the cuboid the appearance of being 3D.

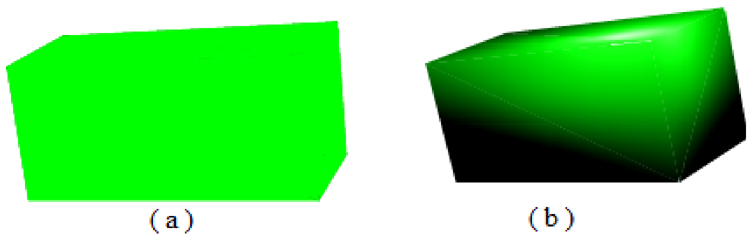


Figure 6.2 A lit and unlit cuboid

The basis of the calculation for lighting or shading objects is the interaction of light and surfaces of the objects in an environment. The technique for determining the colour of a surface of an object at a given point based on the interaction of light and surface is called an illuminating or lighting model (Strauss, 1990; Tabellion, 2004; Ritschel, 2012). The factors that govern the illuminating model determine the visual representation of the 3D object. Modelling only lighting directly from a light source is called local illuminating or direct illuminating model. In such model, the calculation for shading assigned to a point on a surface depend only on the material properties of the surface, the local geometry of the surface, and the locations and properties of the light source, independent from the shading of all other surfaces. A lighting model that handles inter reflection-the light that is reflected from other surfaces to the current surface is called global illuminating model. A global illuminating model is more comprehensive, more physically correct, and produces more realistic images. But it is also more computationally expensive. For the purpose of efficient processing only local illuminating model is used in the SVG GL. Once the illuminating models are defined, a shading model will be used to apply the illuminating model on the 3D object. A shading model is a broader framework that determines how an illuminating model is used and what parameters it receives. For instance, the illuminating model may be used for every pixel covered by an object or just for its vertices.

In this chapter, the lighting filter in existing SVG is introduced firstly, and the problem for using this method in the SVG GL is discussed; then the illumination model in 3D graphic is discussed; finally new Gouraud shading and new shading algorithms in the SVG GL are proposed and developed. In both shading algorithm, the areas are used to interpolate the intensity of colour or normal.

6.2 Discussion of the Existing Algorithms in SVG and Its Problems

Lighting is done in SVG with some specific filter effects. Filters can make the difference between an appealing image with sizzle and one that is dull and ordinary (Figure 6.3). SVG has its own set of filter effects that allow the user to combine several of these effects and apply the filter to the graphic.

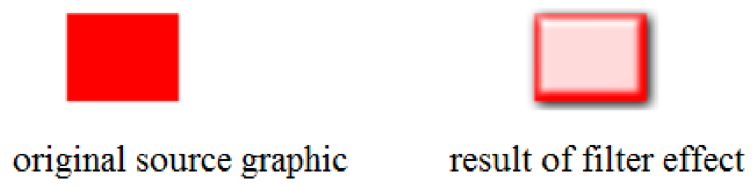


Figure 6.3 Compare ordinary image with the result of filter effect

Lighting effect can be added to SVG with the ‘feDiffuseLighting’ and ‘feSpecularLighting’ filter effects, and the details of the lighting effect can be controlled through one of three filter effects: fePointLight, feDistantLight, and feSpotLight.

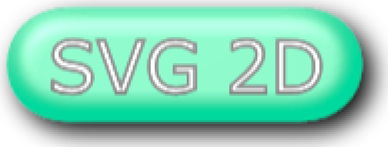
6.2.1 Filter Element

A filter effect consists of a series of graphics operations that are applied to a given source graphic to produce a modified graphical result. The result of the filter effect is rendered to the target device instead of the original source graphic.

Filter effects are defined by <filter> elements. By setting the value of the filter attributes on the given element, a filter effect can be applied to a graphics element or a container element. <filter> element uses an id attribute to uniquely identify it. Filters are defined within <def> elements and then are referenced by graphics elements by their ids. The syntax declaration of <filter> element is shown:

```
<filter
    filterUnits="units to define filter effect region"
    primitiveUnits="units to define primitive filter subregion"
                              x="x-axis co-ordinate"
    y="y-axis co-ordinate"
    width="length"
    height="length"
    filterRes="numbers for filter region"
    xlink:href="reference to another filter" >
</filter>
```

In Figure 6.4, the 3D effect is produced by using SVG filter. Although it looks like 3D, it is actually 2D graphic. This method cannot be used on 3D model for shading, especially when the 3D model is transformed in 3D space.



```
<defs>
<filter id="MyFilter" filterUnits="userSpaceOnUse" x="0" y="0"
width="300"
height="120">
<feGaussianBlur in="SourceAlpha" stdDeviation="4" result="blur"/>
<feOffset in="blur" dx="4" dy="4" result="offsetBlur"/>
<feSpecularLighting in="blur" surfaceScale="5"
specularConstant=".75"
specularExponent="20" lighting-colour="#bbbbbb"
result="specOut">
<fePointLight x="-5000" y="-10000" z="20000"/>
</feSpecularLighting>
<feComposite in="specOut" in2="SourceAlpha" operator="in"
result="specOut"/>
<feComposite in="SourceGraphic" in2="specOut"
operator="arithmetic"
k1="0" k2="1" k3="1" k4="0" result="litPaint"/>
<feMerge>
<feMergeNode in="offsetBlur"/>
<feMergeNode in="litPaint"/>
</feMerge>
</filter>
</defs>
```

Figure 6.4 An example of a filter effect and the SVG file

6.2.2 Lighting Filters

SVG lighting is accessed through the use of feDiffuseLighting or feSpecularLighting filters, that establish its calculations based on the appropriate component of the Phong lighting model

While diffuse light is light that hits a surface and gets scattered equally in all directions, specular light refers to a bright spot of light that gets reflected in a particular direction.

This feDiffuseLighting filter lights an image using the alpha channel as a bump map. The resulting image is an RGBA opaque image based on the light colour with alpha = 1.0 everywhere. The lighting calculation follows the standard diffuse component of the Phong lighting model. The resulting image depends on the light colour, light position and surface geometry of the input bump map.

In feSpecularLighting filter, the lighting calculation follows the standard specular component of the Phong lighting model. The resulting image depends on the light colour, light position and surface geometry of the input bump map. The light source can be defined and controlled by the following 3 filter effects.

1. fePointLight

fePointLight establishes a specific point as the main light source when applying feDiffuseLighting or feSpecularLighting filter.

The x, y, and z attributes here determine the location of the light source in the coordinate system on the appropriate axis. Z will adjust the perceived size of the point of light by determining its location from the point to the user; a higher value here results in a larger point of light that is "closer" to the user(Figure 6.5).

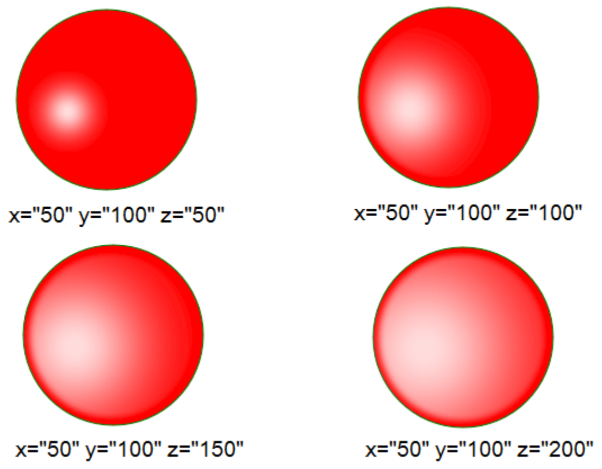


Figure 6.5 The effect of z attribute on light effect

2. feDistantLight

feDistantLight defines a distant light source.

The azimuth attribute within feDistantLight defines the clockwise direction angle in degrees for the light source on the XY plane.

The elevation attribute within feDistantLight defines the direction angle in degrees of the light source from the XY plane towards the z axis (Figure 6.6).

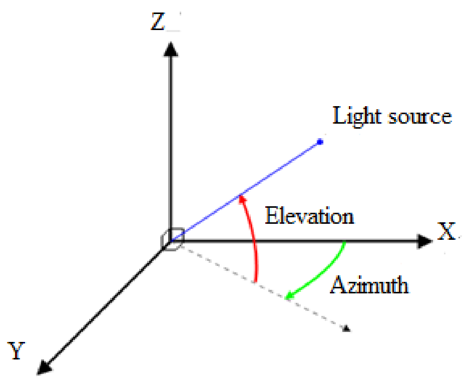


Figure 6.6 Azimuth and elevation in XYZ coordinate

The effects of Azimuth and elevation value on the final result are shown in Figure 6.7.

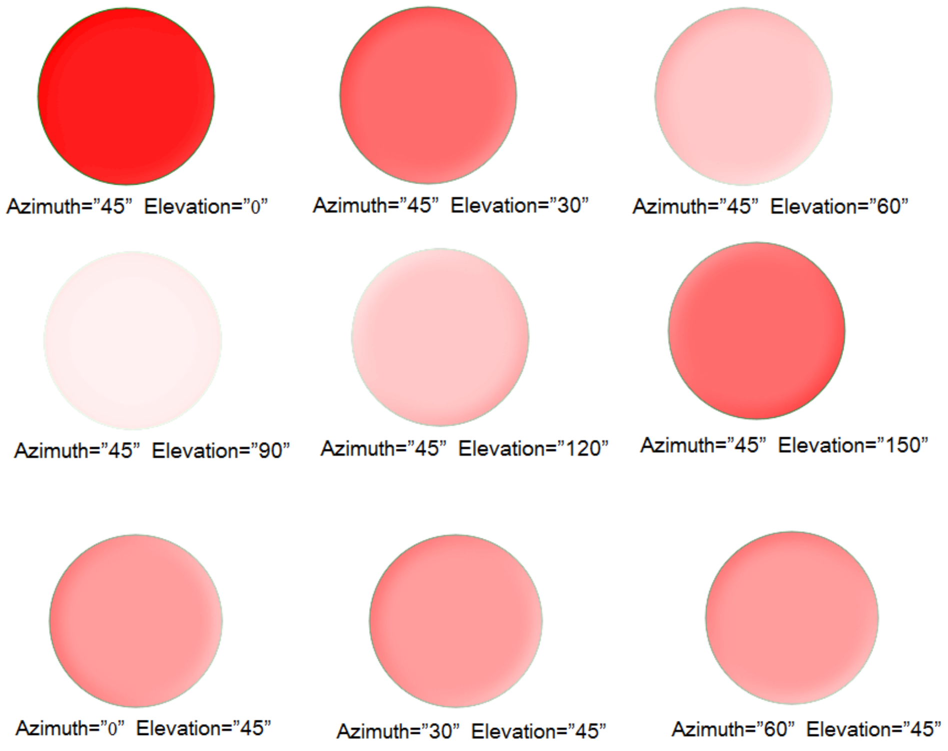

Azimuth="45" Elevation="0"
Azimuth="45" Elevation="30"
Azimuth="45" Elevation="60"
Azimuth="45" Elevation="90"
Azimuth="45" Elevation="120"
Azimuth="45" Elevation="150"
Azimuth="0" Elevation="45"
Azimuth="30" Elevation="45"
Azimuth="60" Elevation="45"

Figure 6.7 The effect of azimuth and elevation attribute on light effect

3. feSpotLight

feSpotLight defines a spot light as a light source.

The x, y, and z values establish the location of the light source along the appropriate axis within the coordinate system.

The pointsAtX, pointsAtY and pointsAtZ attributes define the point at which the light source is pointing.

The limitingConeAngle restricts the area to which light is projected by disallowing light to render outside of it. This value sets the angle in degrees between the spot light axis and cone. A higher value here results in a less restricted area (Figure 6.8).

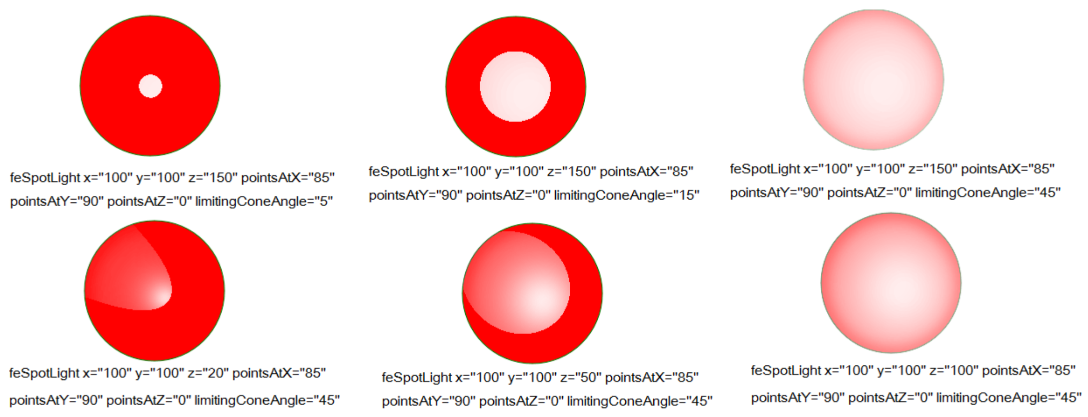

feSpotLight x="100" y="100" z="150" pointsAtX="85" pointsAtY="90" pointsAtZ="0" limitingConeAngle="5"
feSpotLight x="100" y="100" z="150" pointsAtX="85" pointsAtY="90" pointsAtZ="0" limitingConeAngle="15"
feSpotLight x="100" y="100" z="150" pointsAtX="85" pointsAtY="90" pointsAtZ="0" limitingConeAngle="45"
feSpotLight x="100" y="100" z="20" pointsAtX="85" pointsAtY="90" pointsAtZ="0" limitingConeAngle="45"
feSpotLight x="100" y="100" z="50" pointsAtX="85" pointsAtY="90" pointsAtZ="0" limitingConeAngle="45"
feSpotLight x="100" y="100" z="100" pointsAtX="85" pointsAtY="90" pointsAtZ="0" limitingConeAngle="45"

Figure 6.8 The effect of limitingConeAngle and x, y, z attribute on light effect

Although filter effect in SVG can be used to generate appealing image and sometimes they can add some 3D effects on the final image, they can only be used for 2D image processing. They cannot be used to add lighting effect on 3D object, especially when a 3D object is transformed in 3D space, it is impossible to use filter to create physical correct shading result. In order to use advanced shading effect such as Gouraud shading and Phong shading in the SVG GL, new algorithm have to be developed.

Before propose the new Gouraud shading algorithm and Phong shading algorithm in the SVG GL, the illumination model and shading methods in 3D computer graphic will be discussed firstly.

6.3 Illuminating Model

In computer graphics, illuminating is the process used to simulate the interactions of light and the surfaces of an object. From a physical perspective, there are two independent parts involved in this procedure: the light source in the scene, and reflection models that deals with the interaction between material of the surface and light. The properties of the light source determine the properties of light received by the surface of an object; the reflection models define how the light is reflected from the surface of the objects. These reflection models can be classified into 3 different types: ambient reflection model, diffuse reflection model, and specular reflection model.

6.3.1 Light Source

There are three basic types of light sources: point light, spot light, and distant light.

1. Point light

Point light can be defined as a point in space from which light emitted uniformly in all directions. The intensity of illumination received from a point source is proportional to the inverse square of the distance between the source and the surface (Wright, 2004). So the intensity of light at the point $\boldsymbol{P}$ on the surface coming from the point light is given by

$$\boldsymbol{I}(\boldsymbol{P}) = \frac{1}{|\boldsymbol{P}-\boldsymbol{P}_0|^2}\boldsymbol{I}(\boldsymbol{P}_0) \tag{6.1}$$

where $\boldsymbol{P}_0$is the position vector of point light source, $\boldsymbol{I}(\boldsymbol{P}_0)$ is the intensity of the light at the light point, and $|\boldsymbol{P}-\boldsymbol{P}_0|$is the distance between point $\boldsymbol{P}$ and $\boldsymbol{P}_0$(Figure 6.9).

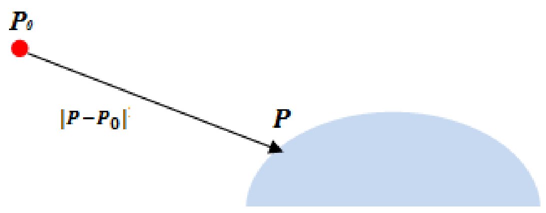


Figure 6.9 Intensity of light at the point $\boldsymbol{P}$ on the surface

2. Spot light

Spot light can be seen as a subset of a point light. In contrast to a point light, a spot light has a direction, in which it spreads its light in the form of a cone. A simple spot light can be constructed from a point light by limiting the angles at which light emit from the source.

A more realistic spot lights are characterized by the distribution of light within the cone. Usually the intensity of light is smaller where is closer to the boundary of the cone than at the center of it. Consider a point $\boldsymbol{P}$ on a surface that is illuminated by a spot light. Let $\boldsymbol{L}$ be a vector that points from point $\boldsymbol{P}$ to point $\boldsymbol{P}_0$where the spot light is located, $\boldsymbol{L}_s$ be the direction of the spot light. The intensity of light at point $\boldsymbol{P}$ is computed by

$$\boldsymbol{I(P)} = \frac{\mathbf{1}}{|\boldsymbol{P}-\boldsymbol{P_0}|^2}\boldsymbol{I(P_0)cos^e\theta} \tag{6.2}$$

where $\boldsymbol{P_0}$is the position vector of point light, $\boldsymbol{I(P_0)}$ is the intensity of the light at the light point, and $\boldsymbol{\theta}$ is the angle between $\boldsymbol{L}$ and $\boldsymbol{L_s}$(Figure 6.10). The light's intensity is highest in the center of the cone. It's attenuated toward the edges of the cone by the cosine of the angle $\boldsymbol{\theta}$, raised to the power of the spot exponent $\boldsymbol{e}$. Thus, higher spot exponents result in a more focused light source. And $\boldsymbol{e}$ determines how rapidly the light intensity drops off.

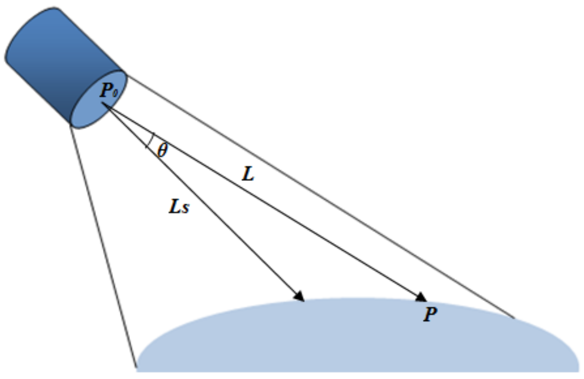


Figure 6.10 Related parameters in spot light

3. Distant light

A distant light, also known as an infinite light, is the light source radiates in a single direction from infinitely far away. Since the light source is far from the surface, the light from the light source strikes all objects that are in close proximity to one another at the same angle. Because the illuminated object is much smaller compared with its distance to the light source, so the intensity of light can be considered to be constant, which means the variation of intensity caused by the distance can be neglected. Therefore distant light is only defined by a direction. The intensity of light at point **P** is

$$\boldsymbol{I(P)} = I \tag{6.3}$$

where *I* is the intensity of the distant light (Figure 6.11).

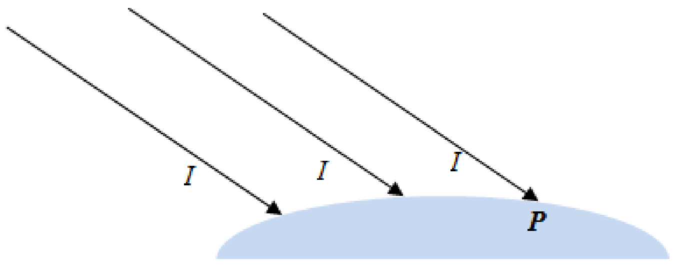


Figure 6.11 Intensity of distant light can be considered to be constant

6.3.2 Ambient Reflection

Ambient light is the low intensity light that arises from the many reflections of light on all surfaces in an environment (Cook, 1981; Zhang, 2009). Ambient light comes from every direction with equal intensity, thus illuminates all objects in the scene equally from all directions. The ambient reflection can be expressed as:

$$\boldsymbol{I} = \boldsymbol{I_a K_a} \tag{6.4}$$

where $\boldsymbol{I}$ is the intensity of reflected light from a surface; $\boldsymbol{I_a}$is the intensity of ambient light, assumed to be constant for all objects in the scene; $\boldsymbol{K_a}$ is object's ambient reflection coefficient, ranges from 0 to 1. The ambient reflection coefficient is a material property. Equation (6.4) shows that intensity $\boldsymbol{I}$ is not affected by the position or orientation of the object in the scene, and independent of the viewer's position.

6.3.3 Diffuse Reflection

A diffuse surface is one for which part of the light incident on a point on the surface is scattered in random direction (Blinn, 1977; Wolff, 1996, 1998). A perfectly diffuse surface reflects the light equally in all directions. This is called diffuse reflection, also known as Lambertian reflection (Angel, 2003), and because light is reflected uniformly in every direction, the appearance of the diffuse reflection appears the same to all views (Figure 6.12).

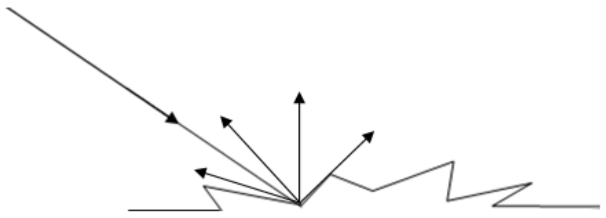


Figure 6.12 Light is reflected uniformly in every direction

The diffuse reflection is modelled by Equation (6.5):

$$\boldsymbol{I} = \boldsymbol{I_d K_d} \boldsymbol{cos}(\theta) \qquad (6.5)$$

where $\boldsymbol{I}$ is the intensity of light reflected from a surface; $\boldsymbol{I_d}$is the intensity of the light source; $\boldsymbol{K_d}$ is the object's diffuse reflection coefficient, ranges from 0 to 1, θ is the angle between the surface normal ***N*** and the light source direction vector ***L*** (Figure 6.13), θ have to be between 0^{o} and 90^{o} if the light source is to have any direct effect on the point being shaded.

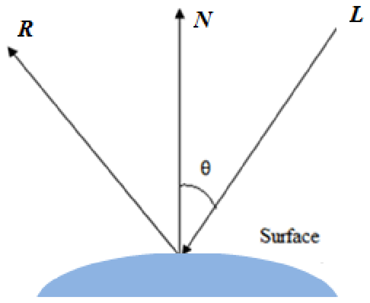


Figure 6.13 The intensity of diffuse reflection is related to the angle between the surface normal ***N*** and the light source direction vector ***L***

Assuming that the vectors ***N*** and ***L*** have been normalized, Equation (6.5) can be rewritten as

$$\boldsymbol{I} = \boldsymbol{I_d K_d}(\boldsymbol{N} \cdot \boldsymbol{L}) \qquad (6.6)$$

The intensity of diffuse reflection depends on $\boldsymbol{\theta}$---the angle between surface normal and the direction of the light source and independent with the position of the viewer.

6.3.4 Specular Reflection

If only ambient and diffuse reflections are employed, the final image will be shaded and will looks like in3D, but all the surfaces will look dull, somewhat like chalk (Figure 6.14). What are missed are the highlights that reflected from shiny objects. In addition to ambient and diffuse reflections, surfaces tend to reflect light strongly along the path given by the reflection of the incident direction across the surface normal. It results in the appearance of a shiny highlight on a surface called specular reflection (Boivin, 2001; Seulin, 2002; Comninos, 2005). The visibility of specular reflection on a surface depends on the position of the viewer.

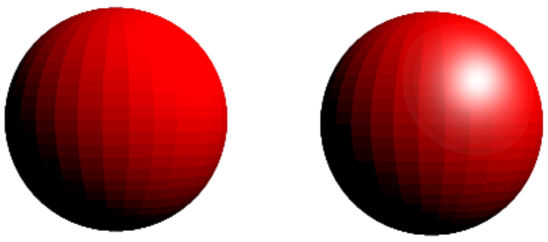


Figure 6.14 Without specular reflection, surfaces look dull, like chalk

The specular component of the illumination model can be given as:

$$\boldsymbol{I} = \boldsymbol{W}(\theta)\boldsymbol{I_p}\boldsymbol{cos}^n(\phi) \tag{6.7}$$

where $\boldsymbol{I}$ is the intensity of light reflected form a surface; $\boldsymbol{I_p}$is the intensity of the light source; n is the specular reflection exponent, the higher the power of n the smaller and brighter the specular highlight (Figure 6.15).

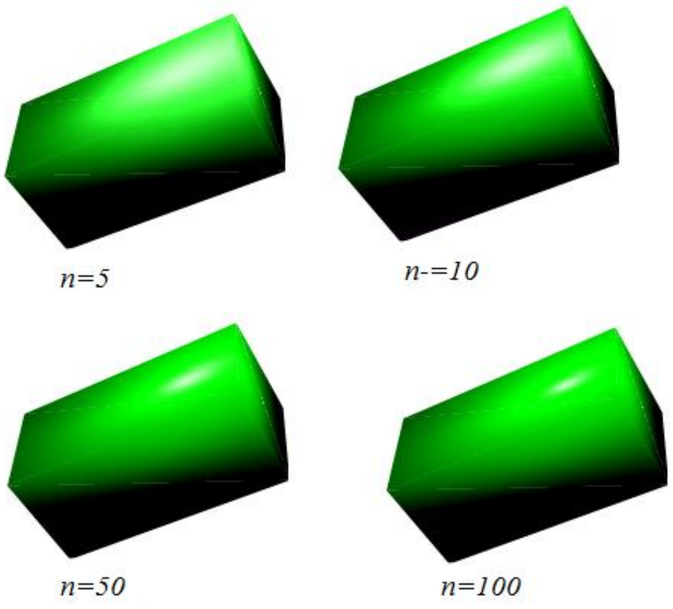


Figure 6.15 Higher the power of $\boldsymbol{n}$ the smaller and brighter the specular highlight

$\boldsymbol{W}(\theta)$ is the fraction striking the surface that is specularly reflected, it is often set as a constant referred to the object's specular-reflection coefficient; $\boldsymbol{\theta}$ is the angle between surface normal $\boldsymbol{N}$ and light source direction $\boldsymbol{L}$; ϕ is the angle between the viewer $\boldsymbol{V}$ and the reflected ray $\boldsymbol{R}$ (Figure 6.16).

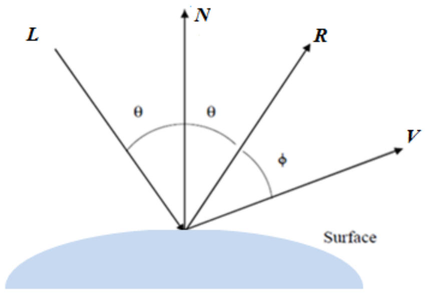


Figure 6.16 The intensity of specular reflection is related to the angle between the viewer ***V*** and the reflected ray ***R***

The intensity of specular reflection not only depends on θ- the angle between surface normal ***N*** and light source direction ***L***, but also depends on ϕ- the angle between the viewer ***V*** and the reflected ray ***R***, this means the specular reflection is affected by the position of viewer.

Light is additive. The reflected model can be achieved by add ambient, diffuse, and specular light together. So the basic illumination model is:

$$\boldsymbol{I} = \boldsymbol{I_a K_a} + \boldsymbol{I_d K_d}\, \boldsymbol{cos}(\theta) + \boldsymbol{W}(\theta)\boldsymbol{I_p}\boldsymbol{cos}^n(\phi) \tag{6.8}$$

6.4 Discussion of Existing Shading Methods

Illuminating model determines the colour of a point on the surface of an object. Shading model determines where the lighting model is applied (Schlick, 1994; Pharr, 2004).A surface can be shaded by calculating the surface normal at each visible point and applying the desired illuminating model at the point. Unfortunately, the amount of computation required for this kind of shading model is too big (Phong, 1975).

The computation can be significantly reduced if the surfaces are approximated with flat polygons, such as triangle. When a triangle is rendered, information known at each vertex is interpolated across the face of the triangle, and then the results can be used to render the triangle. The most common forms of shading model are: Flat shading, Gouraud shading, and Phong shading.

6.4.1 Flat Shading

Flat shading, also known as constant shading applies an illuminating model once to determine a single intensity value used to shade an entire polygon, and each pixel on the polygon is assigned the same intensity (Nicolae, 2004). For a polygon, the colour is determined only for a single pixel based on the normal vector of the polygon. All other pixels on the polygon are given the same colour (Figure 6.17).

Flat shading is the simplest shading method, and applies only one illumination calculation for each primitive, so the performance is more efficiency. It is usually used for high speed rendering where more advanced shading techniques are too computationally expensive. The disadvantage of flat shading is that it gives low-polygon models a faceted look.

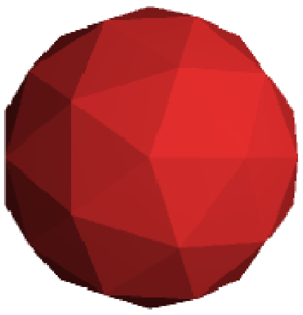


Figure 6.17 All pixels on the same polygon are given the same colour in Flat shading

6.4.2 Gouraud Shading

Gouraud shading is a colour intensity interpolation method (Gouraud, 1971). In Goraud shading, the illuminating equation is used at each vertex of the polygon. Given a normal at each vertex of the polygon, the colour at each vertex is determined from the illuminating equation. The linear interpolation of the colour at each vertex is performed to generate the colour for each pixel on the edges of the polygon. Similarly, the linear interpolation is performed across each scan line to generate colour for each pixel in the polygon (Figure 6.18).

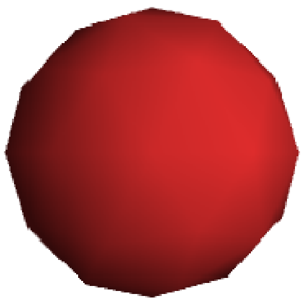


Figure 6.18 Linear interpolation of the colour is performed to generate the colour in Gouraud shading

Gouraud shading is a very simple and effective method of adding a curved feel to a polygon that would otherwise appear flat. However, for large polygons, it can miss specular highlights or at least miss the brightest part of the specular highlight if this falls in the middle of a polygon.

6.4.3 Phong Shading

Phong shading is a normal vector interpolation shading method (Phong, 1975; Bishop, 1986). In Phong shading, the illuminating equation is used at each pixel in the polygon. Given a normal at each vertex of the polygon, the linear interpolation of the normal at each vertex is performed to generate normal for the pixels on the edges of the polygon. Similarly, linear interpolation is performed across each scan line to generate normal for each pixel in the polygon. Then the illuminating equation is used (Figure 6.19).

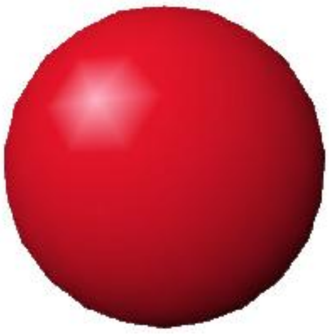


Figure 6.19 Linear interpolation of the normal is performed to calculate the colour in Phong shading

Phong shading overcomes some of the disadvantages of Gouraud shading and specular highlights can be successfully incorporated in the scheme. Phong Shading interpolation phase is three times as expensive as Gouraud Shading, so it significantly increase the computation cost. The other disadvantage of Phong shading is that all the information about the colours and directions of lights needs to be kept until the final rendering stage so that lighting can be calculated at every pixel in the final image.

Although Gouraud shading and Phong shading have been around for years, but due to excessive computation cost, no one has used them in SVG for shading on 3D model. In this PhD project, new Gouraud shading and Phong shading are proposed and developed. Instead of using linear interpolation, an area interpolation is used to generate colour or normal for each pixel inside a polygon.

The render results show that the new algorithms can be used to create ideal shading for the 3D model in the SVG GL.

6.5 Flat Shading, New Gouraud Shading and Phong Shading Algorithms in the SVG GL

6.5.1 Flat Shading in the SVG GL

In the SVG GL, Flat shading uses only one colour per triangle. The illuminating models discussed in Section 6.3 can be used to calculate the desired colour for a triangle. Most of the calculations involve the determination of required vectors and dot products.

1. Normal vector

For smooth surfaces, the normal vector to the surface exists at every point and gives the local orientation of the surface. The calculation of normal vector depends on how the surface is represented.

Given 3 non-collinear points-$\mathbf{P_0}$, $\mathbf{P_1}$, $\mathbf{P_2}$, they are sufficient to determine a triangle or a plane uniquely (Figure 6.20). The normal vector can be found by using the cross product

$$\boldsymbol{N} = (\boldsymbol{P_2} - \boldsymbol{P_0}) \times (\boldsymbol{P_1} - \boldsymbol{P_0}) \tag{6.9}$$

A special care have to be taken about the order of the vectors in the cross product: reversing the order changes the surface from outward to inward, and that reversal can affect the lighting calculation.

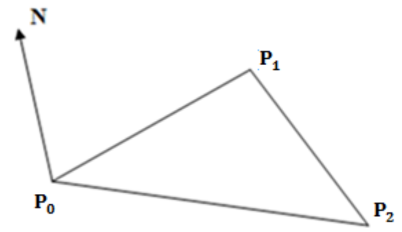


Figure 6.20 Triangle surface normal

The curved surface can be approximated by triangle mesh, and the normal vector of each triangle can be calculated by Equation (6.9).

2. Reflection vector

Once the normal at a point on the surface is calculated, the reflection vector can be computed by using this normal and the direction vector of the light source (Figure 6.21). Calculating reflection normal $\boldsymbol{R}$ required mirroring light source direction vector $\boldsymbol{L}$ about surface normal $\boldsymbol{N}$. Assuming $\boldsymbol{L}$ and $\boldsymbol{N}$ are normalized, the projection of $\boldsymbol{L}$ onto $\boldsymbol{N}$ is $\boldsymbol{Ncos}\theta$, $\boldsymbol{R} = \boldsymbol{Ncos}\theta + \boldsymbol{S}$, where $\boldsymbol{S} = \boldsymbol{Ncos}\theta - \boldsymbol{L}$. Therefore, $\boldsymbol{R} = 2\boldsymbol{Ncos}\theta - \boldsymbol{L}$, substitution $\boldsymbol{cos}\theta = \boldsymbol{N} \cdot \boldsymbol{L}$ yields

$$\boldsymbol{R} = 2\boldsymbol{N}(\boldsymbol{N} \cdot \boldsymbol{L}) - \boldsymbol{L} \tag{6.10}$$

If the light source is at infinity, $\boldsymbol{N} \cdot \boldsymbol{L}$ is constant for a given polygon. For curved surfaces or for a light source not at infinity, $\boldsymbol{N} \cdot \boldsymbol{L}$ varies across the surface.

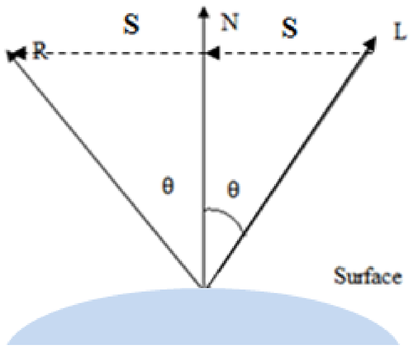


Figure 6.21 Calculation of reflection vector

Once the related vectors are calculated, the triangle colour can be decided by using the following Equation.

$$\boldsymbol{I} = \boldsymbol{I_a K_a} + \boldsymbol{I_d K_d}\, \boldsymbol{cos}(\theta) \tag{6.11}$$

As mentioned in Chapter 5, SVG elements can be painted with uniform single colour, so this colour can be used as the filling colour for the triangle. Figure 6.22 shows a 3D box rendered by using Flat shading. Since each side of the box has the same normal vector, so it is rendered with the same colour.

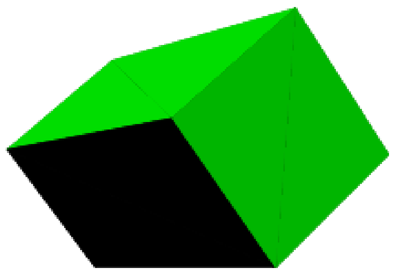


Figure 6.22 Flat shading 3D Box

Flat shading is easy to be implemented and often used for high speed render where advanced shading techniques are too computationally expensive. Since

Flat shading only applies one colour per triangle, the render result is not very realistic, and colour contrast artifacts between polygons are clearly visible, resulting in ‘facetted objects’. For more smooth transitions between triangles, and more realistic rendering, more advanced shading methods---Gouraud shading and Phong shading need to be applied.

6.5.2 A New Gouraud Shading Algorithm in the SVG GL

Gouraud shading is a colour intensity interpolation method based on the illumination of vertex. These colour values are first calculated for each vertex of a triangle, and then interpolation is done between the three vertexes to obtain a gradient.

In the SVG GL, a new Gouraud shading algorithm is proposed and developed to calculate the colour intensity of any point inside a triangle. Instead of using linear interpolation, an area interpolation is used to generate colour for each pixel in the polygon.

Gouraud shading requires that the normal be known for each vertex of the triangle. Then the colour intensity of each vertex will be computed by using the vertex normal with any desired illumination model. Finally each triangle is shaded by area interpolation on three vertex intensities.

$$\boldsymbol{I} = \alpha \boldsymbol{I_0} + \beta \boldsymbol{I_1} + \gamma \boldsymbol{I_2} \tag{6.12}$$

where $\boldsymbol{I_0}, \boldsymbol{I_1}, \boldsymbol{I_2}$ are colour intensity of each triangle vertex, $\boldsymbol{I}$ is the colour intensity of any point $\boldsymbol{P}$ inside the triangle.

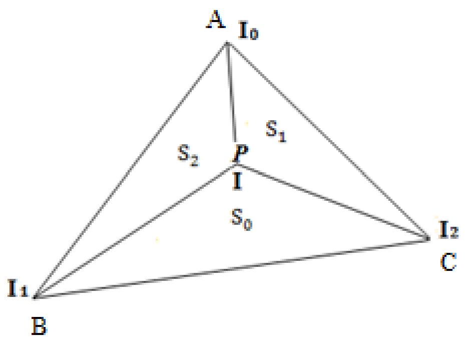


Figure 6.23 Intensity interpolation based on triangle area

The interpolation parameters α, β, and γ can be calculated in terms of the area interpolation. Figure 6.23 shows a triangle, a point $\boldsymbol{P}$ inside the triangle divides the triangles into three subtriangles. The areas of these three small triangles are S_0, S_1, and S_2, and so the area of the entire triangle is equal to $S_0+S_1+S_2$. Parameters α, β, and γ are proportional to the three areas S_0, S_1, and S_2.

$$\alpha = \frac{S_0}{S_0+S_1+S_2}, \quad \beta = \frac{S_1}{S_0+S_1+S_2}, \quad \gamma = \frac{S_2}{S_0+S_1+S_{2}{}_0} \tag{6.13}$$

The colour intesity of the point $\boldsymbol{P}$ inside triangle can be calculated by combining Equation (6.12) and Equation (6.13) together. Then the intensity can be used to fill the triangle. Since different point inside the triangle has different intensity, so the triangle can not be filled with a sigle colour.

The area S_0 can be calculated by:

$$S_0 = \sqrt{d(d-a)(d-b)(d-c)} \tag{6.14}$$

where

$$d = \frac{a+b+c}{2} \tag{6.15}$$

and $a = |\boldsymbol{P} - \boldsymbol{A}|, b = |\boldsymbol{P} - \boldsymbol{B}|, c = |\boldsymbol{P} - \boldsymbol{C}|$.

The area of S_1, S_2 can be calculated by the similar equation.

This new Gouraud shading algorithm defines an SVG pattern for each triangle, and then calculates the colour of each vertex of the triangle by applying the vertex's normal to Equation (6.11), and then the colour of the point inside triangle will be calculated by Equation (6.12), finally the colour will be used to fill the pattern, and the pattern is used to fill the triangle to achieve the Gouraud shading.

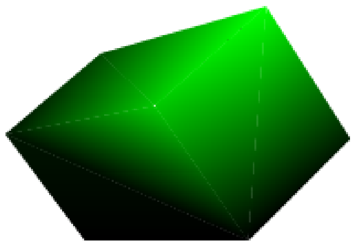


Figure 6.24 Gouraud shading 3D Box

Figure 6.24 shows a 3D box rendered by using the new Gouraud shading algorithm. By comparing Figure 6.22 and Figure 6.24, it shows that new Gouraud shading algorithm produces more smooth and realistic rendering result than Flat shading. Each triangle is shaded by gradient colour instead of a single colour. Since the colour of each point inside the triangle need to be interpolated by Equation (6.12), and a pattern need to be generated for each triangle, so the new Gouraud shading algorithm needs more computational time, and the final SVG file is bigger than using Flat shading.

Table 6.1 Gouraud shading render rate for linear interpolation and area interpolation

Web browser	IE	Firefox	Chrome	Safari	Opera
Linear Interpolation Render Rate (seconds/frame)	0.27	0.30	0.25	0.27	0.31
Area Interpolation Render Rate (seconds/frame)	0.11	0.12	0.11	0.12	0.12

The average number of second per frame which was achieved with linear interpolation and area interpolation on different web browser is shown in Table 6.1. As can be seen in the table, the render time required for area interpolation is less than linear interpolation, and can improve the performance of the Gouraud shading algorithm.

6.5.3 A New Phong Shading Algorithm in the SVG GL

Phong shading is a normal vector interpolation shading method. The normal vectors are first calculated for each vertex of a triangle, and then interpolation is done between the vertexes to obtain a normal vector for a point inside the triangle. Finally each triangle is shaded with colour intensity of the point computed by using the normal with any desired illumination model.

When specular lights are involved, Phong shading produces more realistic result than Gouraud shading, since the specular highlights are completely missed or distorted by Gouraud shading for polygons whose areas are greater than the highlight areas. In spite of this, most of graphics application softwares do not perform Phong shading due to its computational expense. The cost comes from the interpolation of normal and the evaluation of an illuminating model at every pixel. Although Phong shading is computationally expensive, it is still necessary to implement it in the SVG GL. Especially for applications that have a high rendering quality requirement, but relative low rendering speed requirement.

In the SVG GL graphics, new Phong shading is proposed and developed to calculate the normal of any point inside a triangle. Instead of using linear interpolation, an area interpolation is used to calculate normal for each pixel in the polygon.

Phong shading requires that the normal be known for each vertex of the triangle. Then the normal vector of a point $\boldsymbol{P}$ inside the triangle can be interpolation on three vertex intensities.

$$\boldsymbol{N} = \alpha\boldsymbol{N_0} + \beta\boldsymbol{N_1} + \gamma\boldsymbol{N_2} \tag{6.16}$$

where $\boldsymbol{N_0}, \boldsymbol{N_1}, \boldsymbol{N_2}$ are normal vectors of each triangle vertex, $\boldsymbol{N}$ is the normal vector of point $\boldsymbol{P}$.

Figure 6.25 shows a triangle, a point $\boldsymbol{P}$ inside the triangle divides the triangles into three subtriangles. The areas of these three small triangles are S_0, S_1, and S_2, and so the area of the entire triangle is equal to $S_0+S_1+S_2$. So parameters α, β, and γ can be calculated by Equation (6.13).

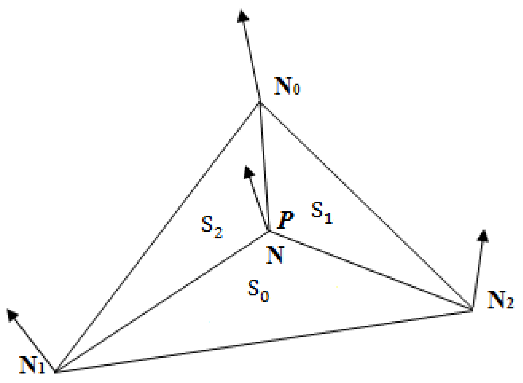


Figure 6.25 Normal vector interpolation based on triangle area

In the SVG GL graphics, the new Phong shading algorithm defines an SVG pattern for each triangle, and then interpolate the normal of each point $\boldsymbol{P}$ inside the triangle by Equation (6.16), and then the colour of the point will be calculated by applying the normal vector to Equation (6.8), finally the colour will be used to fill the pattern, and the pattern is used to fill the triangle to achieve the Phong shading.

Figure 6.26 shows a box is rendered by the new Phong shading algorithm introduced in this section. A specular highlight is added to achieve more realistic results. Although this method can be used to achieve Phong shading, it is obvious that it needs more computational time, but the final SVG file size is similar as the Gouraud shading.

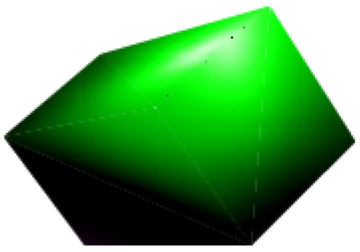


Figure 6.26 Phong shading 3D Box

Table 6.2 Phong shading render rate for linear interpolation and area interpolation

Web browser	IE	Firefox	Chrome	Safari	Opera
Linear Interpolation Render rate (seconds/frame)	0.51	0.50	0.55	0.52	0.50
AreaLinear Interpolation Render rate (seconds/frame)	0.15	0.16	0.22	0.17	0.19

The average number of second per frame which was achieved with linear interpolation and area interpolation on different web browser is shown in Table 6.2. As can be seen in the table, the render time required for area interpolation is far less than linear interpolation, and can significantly improve the performance of the Gouraud shading algorithm.

6.6 Summary

In this chapter, SVG filter is discussed first. Although filter effect in SVG can be used to generate appealing image and sometimes they can add some 3D effects on the final image, they can only be used for 2D image processing. They cannot be used to add lighting effect on 3D object, especially when a 3D object is transformed in 3D space, it is impossible to use filter to create physical correct shading result.

Then different illuminating models and shading methods are introduced in this chapter. Although Gouraud shading and Phong shading have been around for years, but due to excessive computation cost, no one has used them in SVG for shading on 3D model.

In this chapter, new Gouraud shading and Phong shading are proposed and developed. Instead of using linear interpolation, an area interpolation is used to generate colour or normal for each pixel inside a polygon. The render results show that the new algorithms can be used to create ideal shading for the 3D model in the SVG GL.

Chapter 7 New Algorithms for Texture Mapping in the SVG GL

7.1 Introduction

As detail of 3D model becomes finer and more intricate, 3D modelling with polygons or other geometric primitives becomes less practical. An alternative is to map an image, either digitized or synthesized, onto a surface. This approach is known as texture mapping. Texture mapping is one of the most successful techniques in high quality image synthesis (Carey, 1985; Heckbert, 1986; Haeberli, 1993). Its use can enhance the visual realism while only a relatively small increase in computation.

Textures can be one, two, or three dimensional. For example, a 1D texture might be used to create a pattern for colouring a curve (Lefebvre, 2003). A 2D texture is mapped to the surface of a shape or polygon. This process is akin to applying patterned paper to a plain white box. It can be used to render complicated shapes like trees, clouds, or people, with a single polygon (Elinas, 2000; Harris, 2003). A 3D texture, also called solid texture, is basically the equivalent of carving the object out of a block of material (Dischler, 2001; Pietroni, 2007). It places the texture onto the object coherently, not producing discontinuities of texture where two faces meet. 3D texture can be used to simulate the wood grain on a cube to avoid discontinuities of grain along the edges of the cube (Heeger, 1995).

Since the use of surfaces is so important in computer graphics, mapping 2D texture to surface is by far the most common use of texture mapping. The new proposed the SVG GL is a polygon based 3D modelling method; the 3D object created by the SVG GL is approximated by multiple triangles, so only 2D texture mapping that will be implemented in the SVG GL.

There are lots of different textures mapping algorithms in 3D computer graphics. But they are all pixel-based, that means when a primitive is rendered, texture parameters for each image pixel are determined, and used to address the appropriate texture pixels. In the SVG GL, the elementary primitive is triangle. So the existing texture mapping algorithm cannot be used in the SVG GL. New texture mapping algorithms have to be proposed and developed.

7.2 Texture Mapping in SVG

SVG is a language for describing 2D graphics in XML. SVG pattern is used to fill a shape with a pattern made up from images. This pattern can be made up from SVG images (shapes) or from bitmap images.

A pattern is used to fill or stroke an object using a pre-defined graphic object that can be replicated at fixed intervals in *x* and *y* to cover the areas to be painted. Patterns are defined using a 'pattern' element and then referenced by properties 'fill' and 'stroke' on a given graphics element to indicate that the given element shall be filled or stroked with the referenced pattern.

Attributes '*x*', '*y*', '*width*', '*height*' and '*patternUnits*' define a reference rectangle somewhere on the infinite canvas. The reference rectangle has its top/left at (*x, y*) and its bottom/right at (*x* + *width*, *y* + *height*).

Here is a simple SVG fill pattern example:

```
<defs>
   <pattern id="pattern1" x="10" y="10" width="20" height="20"
                  patternUnits="userSpaceOnUse" >
  <circle cx="10" cy="10" r="10" style="stroke: none; fill: #0000ff" />
  </pattern>
</defs>
<rect x="10" y="10" width="100" height="100" style="stroke:
#000000; fill: url(#pattern1);" />
```

The result is shown in Figure 7.1.

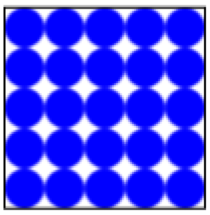


Figure 7.1 SVG pattern

Although an SVG pattern can be use to display a 2D image, to use it to wrap 2D texture on 3D object, it still has problems. First, not all 3D objects surfaces are flat, so using SVG pattern directly on curved 3D surface will cause unexpected distort (Figure 7.2); second, when 3D objects are transformed in 3D space, using the SVG pattern directly cannot create transformed texture accordingly, so the final texture mapping is incorrect (Figure 7.3). In order to use SVG pattern for wrapping texture on 3D objects in the SVG GL, there is still more works to be done, new algorithms have to be proposed. In this PhD project, the pattern based image transformed texture mapping algorithms for different 3D objects in the SVG GL are proposed, and discussed from Section 7.5 to Section 7.12.

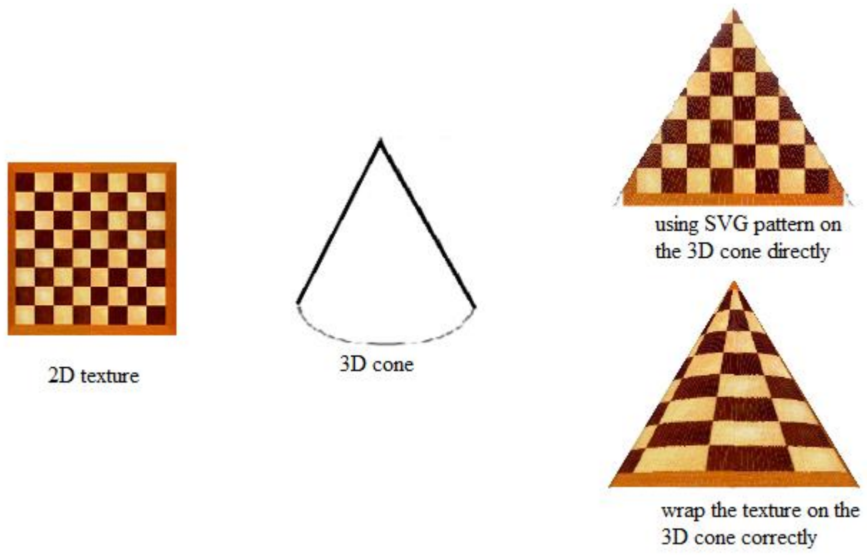

2D texture
3D cone
using SVG pattern on the 3D cone directly
wrap the texture on the 3D cone correctly

Figure 7.2 Compare using SVG pattern directly on an 3D cone with the correct texture mapping on the 3D cone

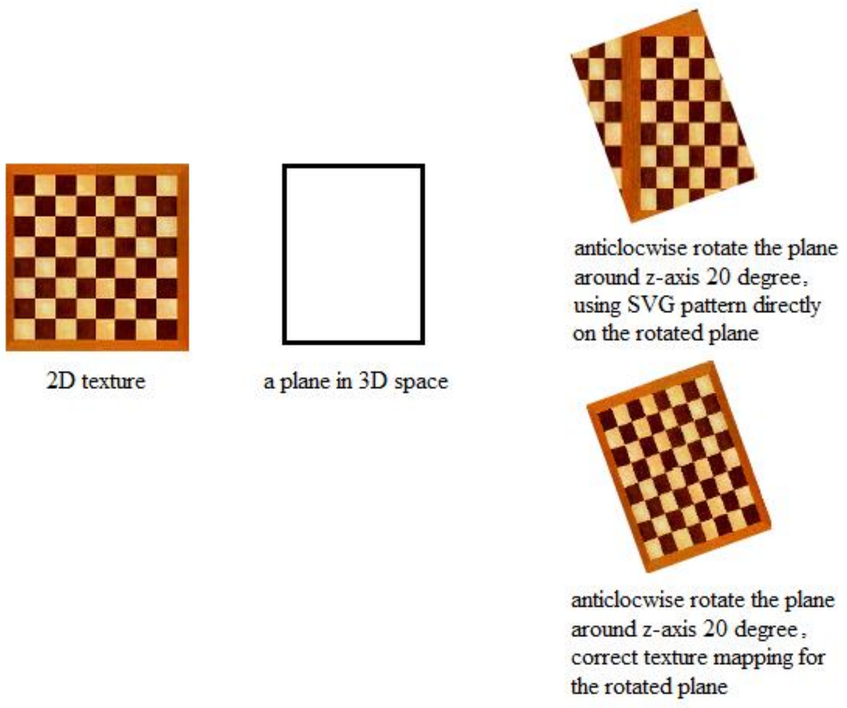

2D texture
a plane in 3D space
anticlocwise rotate the plane around z-axis 20 degree, using SVG pattern directly on the rotated plane
anticlocwise rotate the plane around z-axis 20 degree, correct texture mapping for the rotated plane

Figure 7.3 Compare using SVG pattern directly on a rotated plane with the correct texture mapping on the rotated plane.

7.3 Texture Mapping

Before discussing texture mapping, there are 3 coordinate spaces need to be defined. Screen space, is a 2D space where the final image is displayed; object space, is a 3D space where the objects upon which the textures will be mapped is defined; texture space, is a 2D space which the position of the texture is located.

In computer graphics, texture mapping can be referred as a transformation from texture space to screen space (Oliveira, 2000). This transformation can be split into two phase (Figure 7.4). The first is the surface parameterization that establishes the one-to-one correspondence of points from texture space to object space, and then followed by the standard geometrical and projection transformations that affect the mapping from object space to screen space.

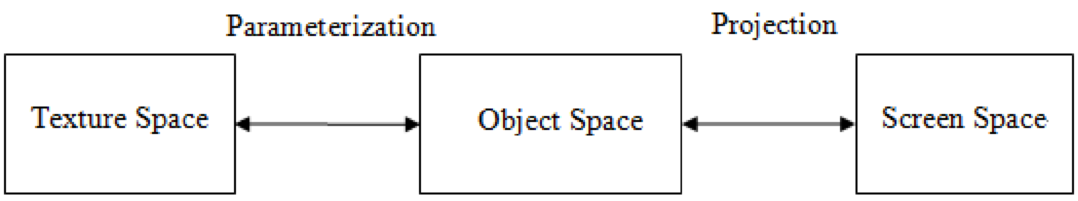


Figure 7.4 Texture space to screen space transformation

The mapping between texture space and screen space has to be evaluated for each pixel to be shaded. Generally there are two major types of implementations: forward texture mapping and backward texture mapping.

Forward texture mapping, also called texture order, scans the data in texture space and maps from texture space to screen space (Chen, 1999; Deng, 2002). In forward texture mapping, each coordinate pair (u, v) on texture space is mapped to point (x, y) on screen space. Firstly, the coordinate (u, v) will be mapped to a point on a 3D surface in object space by parameterization; then the point on 3D surface will be projected to 2D screen. So coordinate (u, v) on texture space is mapped to point (x, y) on screen space via a pair of function:

$$\begin{cases} x = \boldsymbol{X}(u, v) \\ y = \boldsymbol{Y}(u, v) \end{cases} \tag{7.1}$$

Forward texture mapping is performed with only one operation per pixel in the texture space. The application of this forward mapping algorithm to a texture will result in the kinds of situation shown in Figure 7.5. The output image on the screen space will be left with ‘holes’ (pixels with unknown values) where the output is scaled up compared with the input texture, and multiple pixel overlaps where the output is scaled down with respect to the input.

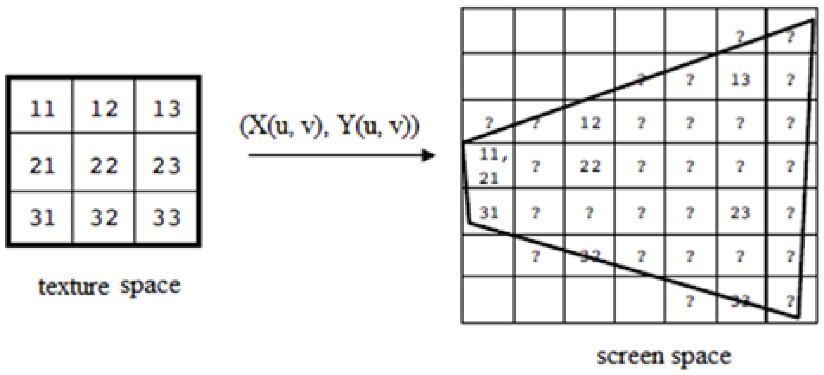


Figure 7.5 Forward mapping leaves holes and overlaps

One solution is to add mapped samples into a screen space accumulator buffer with a filter function. Forward texture mapping is preferable only when the texture-to –screen mapping is difficult to invert, or when the texture image have to be read sequentially.

Backward texture mapping, also called screen order or inverse mapping, it scans the pixels in screen space and uses the mapping from screen space to texture space (Wei, 2008; Chen, 2010). The problems with the forward texture mapping can be solved by backward texture mapping. Instead of sending each input pixel to an output pixel, backward mapping looks at each output pixel and determine what input pixels map to it. In backward mapping, the coordinate (x, y) on 2D screen will be map to a point on a 3D surface in object space firstly; then the point on 3D surface will be mapped to 2D texture by parameterization. By inverting the forward mapping function $\boldsymbol{X}(u, v)$, $\boldsymbol{Y}(u, v)$, the backward mapping function can be defined as:

$$\begin{cases} u = \boldsymbol{U}(\boldsymbol{X}(u,v),\boldsymbol{Y}(u,v)) \\ v = \boldsymbol{V}(\boldsymbol{X}(u,v),\boldsymbol{Y}(u,v)) \end{cases} \tag{7.2}$$

Each pixel in screen space is inverse-transformed to texture space and the textel value there is read. Backward texture mapping is preferred when the screen has to be written sequentially, the mapping is invertible, and the texture is random access.

As mentioned in previous section, texture mapping consists of a transformation of 2D texture space to a 3D object surface via parameterization, and then a projection of that surface onto 2D screen space. 2D mappings are central to each of these transformations.

Conceptually, a small area of the texture maps to the area of the surface of a 3D object, corresponding to pixels in the final image. Colour values can be used, either to modify the colour of the surface that might have been determined by a lighting model, or to assign a colour to the surface based on only the texture value.

On closer examination of the texture mapping procedure, there are still a number of difficulties.

1. Parameterization, the map from texture space to object space has to be determined. A texture is usually defined over a rectangle region in 2D texture space, but most surfaces in 3D object space are not flat. The mapping from this rectangle to an arbitrary region in 3D space may be a complex function or may have undesirable properties; there is often no single best method of assigning texture space to object space.

2. Although SVG supports 2D raster images, texture mapping in the SVG GL can be very tricky. 2D image can be displayed by using <Pattern> elements in SVG. But originally, they can only be used with 2D shape, in order to use them as texture for 3D objects; the 2 problems mentioned in section 7.2 have to be solved

7.4 Texture Mapping in the SVG GL

Due to the problem with existing SVG pattern for texture mapping in the SVG GL, and the traditional texture mapping also cannot be applied to the SVG GL, the new texture mapping algorithms are proposed and developed in this project. The new algorithms are based on SVG pattern and transformation of the 2D texture according to the geometric transformation of 3D model. The procedure of this algorithm is shown in Figure 7.6. There are four steps:

Step1: Parameterization, mapping the 2D texture to 3D model by the parameterization equation.

Step 2: Transformation, transforming the 2D texture according to the geometric transformation of the 3D model.

Step 3: SVG pattern creation, generating SVG pattern based on the transformed texture.

Step 4: SVG pattern application, applying the SVG pattern onto the 3D model.

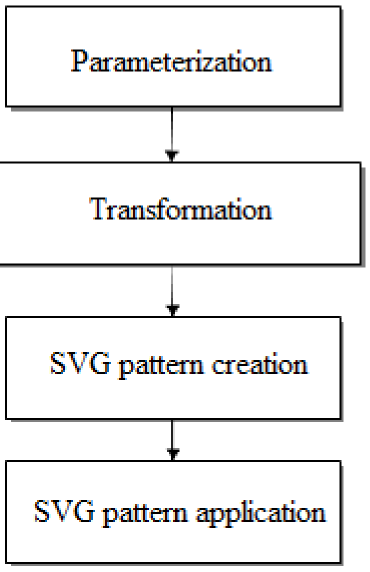


Figure 7.6 The procedure of the pattern based image transformed texture mapping algorithms

The new texture mapping algorithm is a mapping that transform a 2D texture in texture space into a 2D screen space. It maps a source point (u, v) in texture space to a destination point (x, y) in screen space according to the geometrical and projection transformation of the 3D object.

The elementary primitive in the SVG GL graphics is triangle, so the new texture mapping algorithm with a triangle will be proposed and developed firstly; then texture mapping algorithm for different 3D models in the SVG GL will be proposed and developed from Section 7.6 to Section 7.12, namely: plane, cylinder, sphere, cone, and complex 3D models in the SVG GL.

7.5 A New Texture Mapping Algorithm for a Triangle in the SVG GL

A new texture mapping algorithm for a triangle is proposed and developed in this section. This new algorithm is based on the algorithm proposed in Section 7.4. The parameterization equation from a 2D texture to a triangle is used at the first step.

A triangle can be defined by three non-collinear point ***A*** (x_0, y_0, z_0), ***B*** (x_1, y_1, z_1), and ***C*** (x_2, y_2, z_2). The corresponding points in 2D texture space are ***D*** (u_0, v_0), ***E*** (u_1, v_1), and ***F*** (u_2, v_2) (Figure 7.7). The complete transformation from texture space to object space is:

$$\begin{cases} x = a \cdot u + b \cdot v + c \\ y = d \cdot u + e \cdot v + f \\ z = g \cdot u + h \cdot v + i \end{cases} \tag{7.3}$$

The unknown coefficients can be derived from the solution of a linear equations developed by putting the coordinates of points ***A***, ***B***, ***C***, ***D***, ***E***, and ***F*** into this Equation. And the results are:

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} u_0 & v_0 & 1 \\ u_1 & v_1 & 1 \\ u_2 & v_2 & 1 \end{bmatrix}^{-1} \begin{bmatrix} x_A \\ x_B \\ x_C \end{bmatrix} \tag{7.4}$$

$$\begin{bmatrix} d \\ e \\ f \end{bmatrix} = \begin{bmatrix} u_0 & v_0 & 1 \\ u_1 & v_1 & 1 \\ u_2 & v_2 & 1 \end{bmatrix}^{-1} \begin{bmatrix} y_A \\ y_B \\ y_C \end{bmatrix} \tag{7.5}$$

$$\begin{bmatrix} g \\ h \\ i \end{bmatrix} = \begin{bmatrix} u_0 & v_0 & 1 \\ u_1 & v_1 & 1 \\ u_2 & v_2 & 1 \end{bmatrix}^{-1} \begin{bmatrix} z_A \\ z_B \\ z_C \end{bmatrix} \tag{7.6}$$

Then the intermediate data will be transformed according to the geometrical and projection transformation of the plane by the following Equation.

$$\boldsymbol{P}' = \boldsymbol{M}_p \boldsymbol{M}_G \boldsymbol{P} \tag{7.7}$$

where $\boldsymbol{P}$ is the intermediate data generated from the parameterization, $\boldsymbol{P}'$ is the transformed intermediate data. $\boldsymbol{M}_G$ and $\boldsymbol{M}_p$ are the geometrical transformation and projection matrices according to the transformation and projection of the triangle. After the original texture is transformed according to Equation (7.7), the SVG pattern is generated from the transformed texture. Finally the texture is wrapped to the triangle by applying the pattern to the triangle.

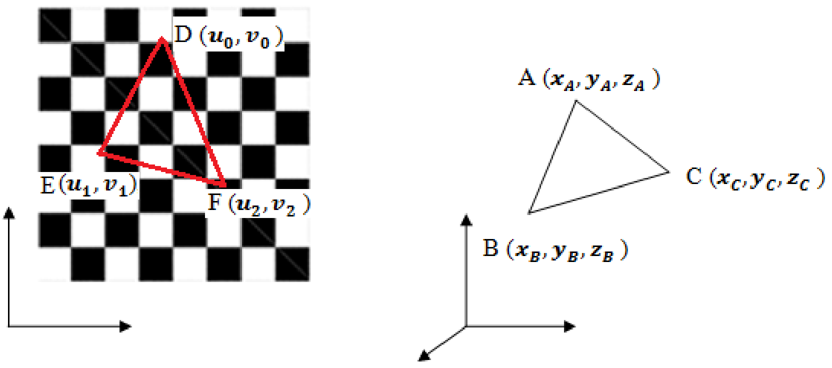


Figure 7.7 Parameterization of a triangle

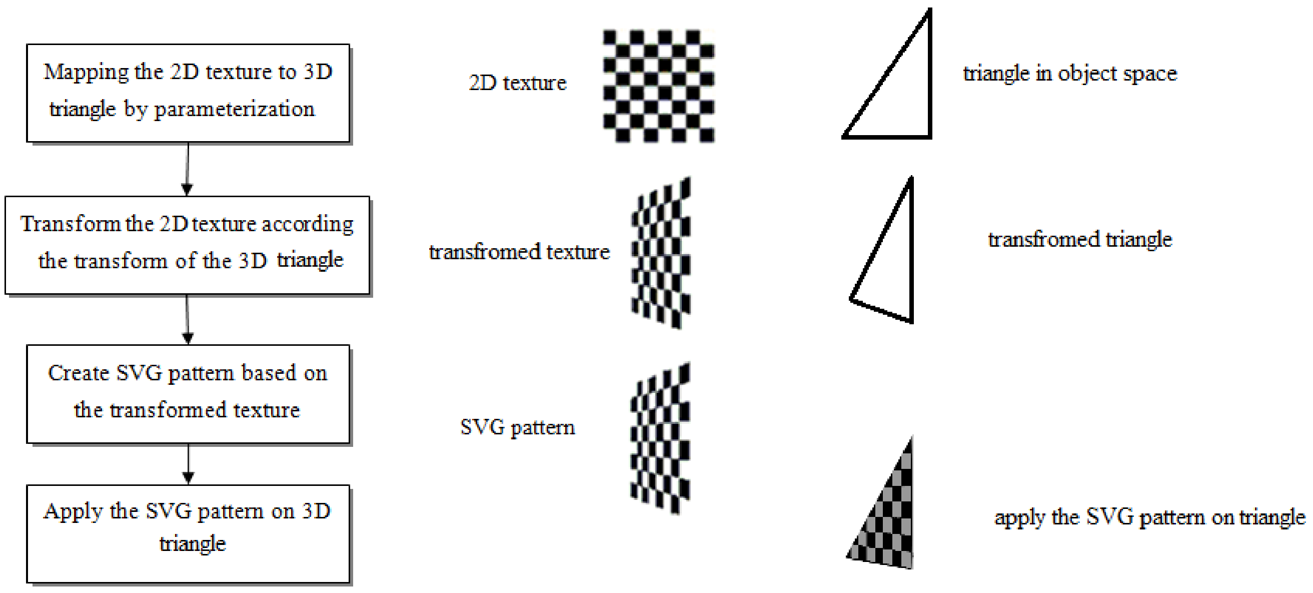
The texture mapping algorithm for a triangle can be described in Figure 7.8.

1. A triangle is defined in object space with coordinates (*x*, *y*, *z*), and the corresponding texture is defined in texture space with coordinates (*u*, *v*). The texture is parameterized by Equation (7.3) to generate the intermediate data.

2. The intermediate data will be transformed according to the geometrical and projection transformation of the triangle by Equation (7.7).

3. The SVG pattern is generated from the transformed texture.

4. Finally the texture is wrapped onto the triangle by adding the pattern to the triangle.



Mapping the 2D texture to 3D triangle by parameterization
2D texture
triangle in object space
Transform the 2D texture according the transform of the 3D triangle
transfromed texture
transfromed triangle
Create SVG pattern based on the transformed texture
SVG pattern
Apply the SVG pattern on 3D triangle
apply the SVG pattern on triangle

Figure 7.8 New texture mapping algorithm for a triangle in the SVG GL

Figure 7.9 (a) shows the texture is wrapped on a triangle by using the new texture mapping algorithm. The triangle is rotated around x-axis, y-axis, and z-axis respectively. Figure 7.9 (b) shows the texture by applying SVG pattern directly (without transform according to the triangle) on the triangle, and this triangle is also rotated around x-axis, y-axis, and z-axis respectively.

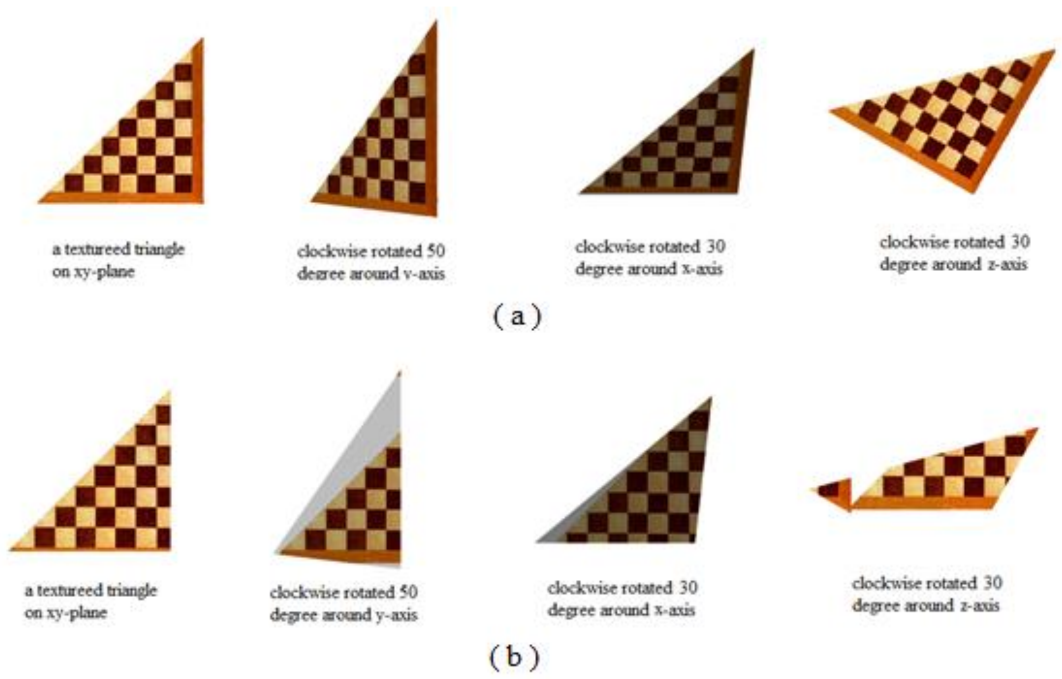

a textureed triangle on xy-plane
clockwise rotated 50 degree around y-axis
clockwise rotated 30 degree around x-axis
clockwise rotated 30 degree around z-axis
(a)
a textureed triangle on xy-plane
clockwise rotated 50 degree around y-axis
clockwise rotated 30 degree around x-axis
clockwise rotated 30 degree around z-axis
(b)

Figure 7.9 Compare texture mapping create by new algorithm for a triangle and by using SVG pattern directly

By comparing Figure 7.9 (a) and Figure 7.9 (b), it shows applying SVG pattern directly on the triangle will create distort when the triangle is transformed in 3D space. But the new texture mapping algorithm for a triangle can generate realistic texture for the triangle even the triangle is transformed in 3D space.

7.6 A New Texture Mapping Algorithm for a Plane in the SVG GL

A new texture mapping algorithm for a plane is proposed and developed in this section. This new algorithm is based on the algorithm proposed in Section 7.4. The parameterization equation from a 2D texture to a plane will be used at the first step.

The parameterization of a plane can be derived from the parameterization of triangle, by subdividing the plane into 2 triangles and generating a parameterization for the separate triangles by Equation (7.3).

The new texture mapping algorithm for a plane can be described in Figure 7.10.

1. A plane is defined in object space with coordinates (*x*, *y*, *z*), and the corresponding texture is defined in texture space with coordinates (*u*, *v*). The plane is divided to 2 triangles, and then the texture is parameterized by Equation (7.3) to generate the intermediate data.

2. The intermediate data will be transformed according to the geometrical and

projection transformation of the plane by Equation (7.7).

3. The SVG pattern is generated from the transformed texture.

4. Finally the texture is wrapped onto the plane by adding the pattern to the plane.

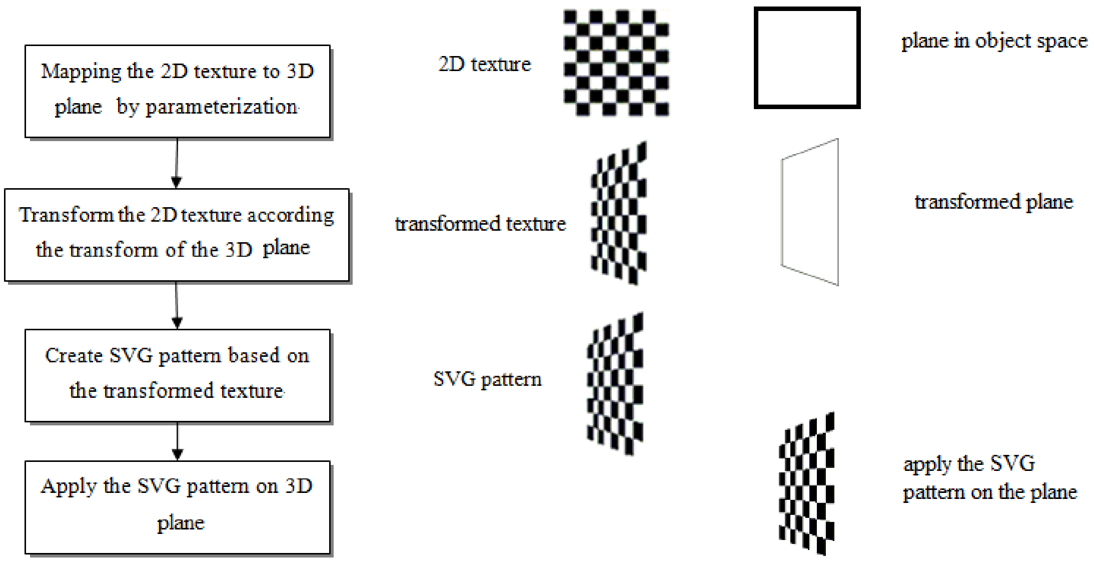
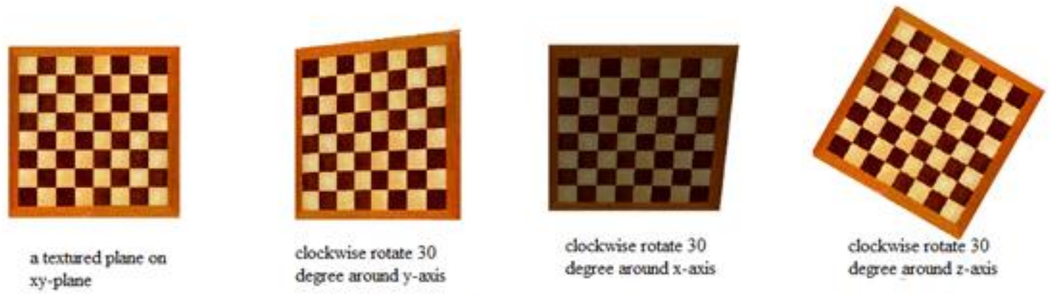
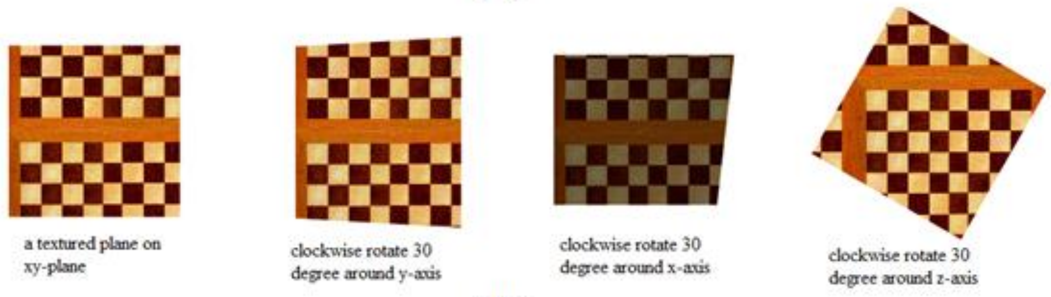

Mapping the 2D texture to 3D plane by parameterization
Transform the 2D texture according the transform of the 3D plane
Create SVG pattern based on the transformed texture
Apply the SVG pattern on 3D plane
2D texture
plane in object space
transformed texture
transformed plane
SVG pattern
apply the SVG pattern on the plane

Figure 7.10 New texture mapping algorithm for a plane in the SVG GL

Figure 7.11(a) shows the texture is wrapped on a plane by using the new texture mapping algorithm. The plane is rotated around *x*-axis, *y*-axis, and *z*-axis respectively. Figure 7.11(b) shows the texture applying SVG pattern directly (without transform according to the plane) on the plane, and this plane is also rotated around x-axis, y-axis, and z-axis respectively.


a textured plane on xy-plane
clockwise rotate 30 degree around y-axis
clockwise rotate 30 degree around x-axis
clockwise rotate 30 degree around z-axis

(a)


a textured plane on xy-plane
clockwise rotate 30 degree around y-axis
clockwise rotate 30 degree around x-axis
clockwise rotate 30 degree around z-axis

(b)

Figure 7.11 Compare texture mapping create by the new algorithm for a plane and by using SVG pattern directly

By comparing Figure 7.11(a) and Figure 7.11(b), it shows applying SVG pattern directly on the plane will create distort when the plane is transformed in 3D space. But the new texture mapping algorithm for a plane can generate realistic texture for the plane even the plane is transformed in 3D space.

7.7 A New Texture Mapping Algorithm for a Sphere in the SVG GL

A new texture mapping algorithm for a sphere is proposed and developed in this section. This new algorithm is based on the algorithm proposed in Section 7.4. The parameterization equation from a 2D texture to a sphere will be used at the first step.

The implicit definition of a sphere around point (x_0, y_0, z_0) with radius *r* is:

$$(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 = r^2 \tag{7.8}$$

An appropriate parameterization can be derived using a spherical coordinates:

$$\begin{cases} x = x_0 + rcos\theta sin\varphi \\ y = y_0 + rcos\varphi \\ z = z_0 + rsin\theta sin\varphi \end{cases} \tag{7.9}$$

The spherical coordinate θ represents the heading angle that covers the range $[0, 360^o]$, and φ represents the azimuth angle that covers the range $[-90^o, 90^o]$, thus, the appropriate choice for assigning texture map coordinates would be

$$\begin{cases} u = \frac{\theta}{360} \\ v = \frac{2\varphi+180}{360} \end{cases} \tag{7.10}$$

The complete transformation from texture space to object space is:

$$\begin{cases} x(u,v) = x_0 + r \cdot cos(360 \cdot u) \cdot sin[180 \cdot (v-0.5)] \\ y(u,v) = y_0 + r \cdot cos[180 \cdot (v-0.5)] \\ z(u,v) = z_0 + r \cdot sin(360 \cdot u) \cdot sin[180 \cdot (v-0.5)] \end{cases} \tag{7.11}$$

The new texture mapping algorithm for a sphere can be described in Figure 7.12.

1. A sphere is defined in object space with coordinates (*x*, *y*, *z*), and the corresponding texture is defined in texture space with coordinates (*u*, *v*). The texture is parameterized by Equation (7.11) to generate the intermediate data.

2. The intermediate data will be transformed according to the geometrical and projection transformation of the sphere by Equation (7.7).

3. The SVG pattern is generated from the transformed texture.

4. Finally the texture is wrapped onto the sphere by adding the pattern to the sphere.

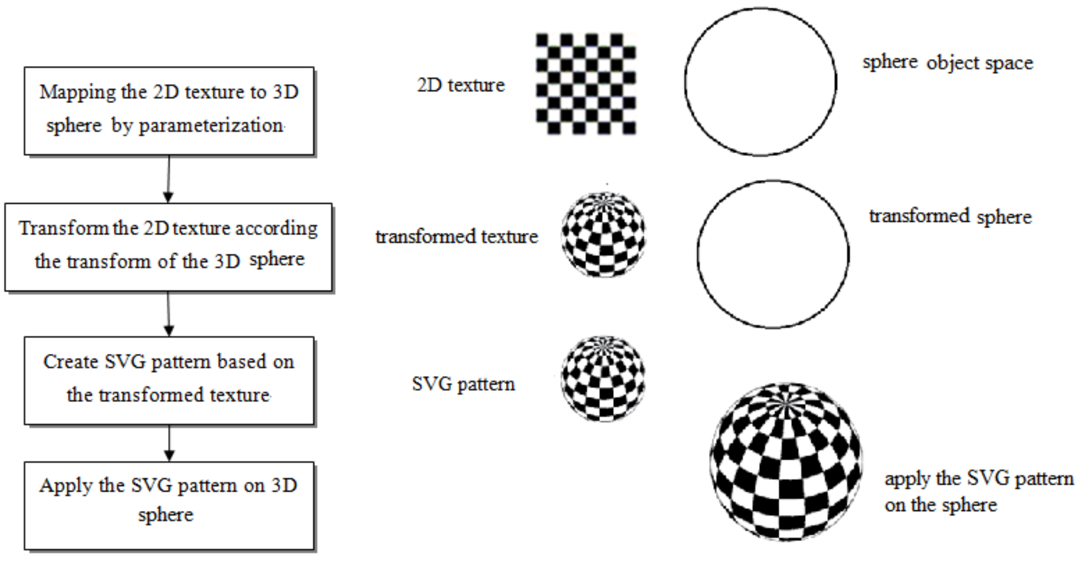

Mapping the 2D texture to 3D sphere by parameterization
Transform the 2D texture according the transform of the 3D sphere
Create SVG pattern based on the transformed texture
Apply the SVG pattern on 3D sphere
2D texture
sphere object space
transformed texture
transformed sphere
SVG pattern
apply the SVG pattern on the sphere

Figure 7.12 New texture mapping algorithm for a sphere in the SVG GL

The proposed algorithm is used to generate texture for a sphere, and the result is compared with the texture mapping created by using SVG pattern directly (without transform according to the sphere). Figure 7.13(a) shows the texture is wrapped onto a sphere by using the new texture mapping algorithm for a sphere. The sphere is rotated around *x*-axis, *y*-axis, and *z*-axis respectively. Figure 7.13(b) shows the texture applying SVG pattern directly on the sphere, and this sphere is also rotated around *x*-axis, *y*-axis, and *z*-axis respectively.

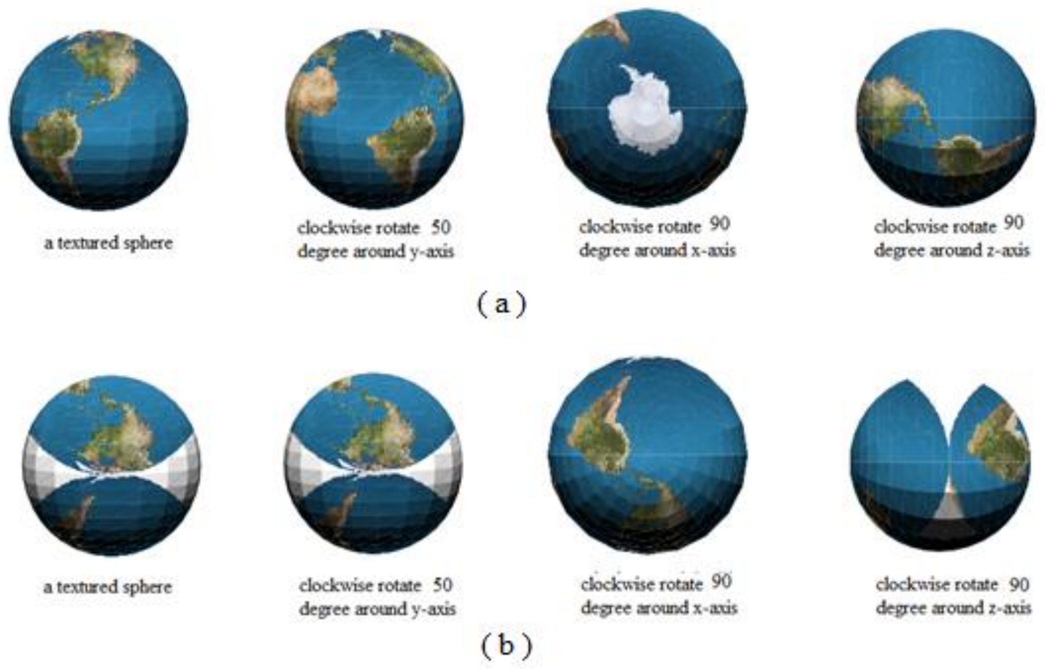

a textured sphere
clockwise rotate 50 degree around y-axis
clockwise rotate 90 degree around x-axis
clockwise rotate 90 degree around z-axis
(a)
a textured sphere
clockwise rotate 50 degree around y-axis
clockwise rotate 90 degree around x-axis
clockwise rotate 90 degree around z-axis
(b)

Figure 7.13 Compare texture mapping create by new algorithm for a sphere and by using SVG pattern directly

By comparing Figure 7.13(a) and Figure 7.13(b), it shows applying SVG pattern directly on the sphere will create distort when the sphere is transformed in 3D space. But the new texture mapping algorithm for a sphere can generate realistic texture for the sphere even the sphere is transformed in 3D space.

7.8 A New Texture Mapping Algorithm for a Cylinder in the SVG GL

A new texture mapping algorithm for a cylinder is proposed and developed in this section. This new algorithm is based on the algorithm proposed in Section 7.4. The parameterization equation from a 2D texture to a cylinder will be used at the first step.

A cylinder of height H centered at the origin and located around the ***y*** axis has the following implicit Equation:

$$x^2 + z^2 = r^2, 0 \leq y \leq H \tag{7.12}$$

The same cylinder can be conveniently expressed by cylindrical coordinates ($\theta \in [0, 360^o], h \in [0, \mathrm{H}]$):

$$\begin{cases} x(\theta, h) = r \cdot cos\theta \\ y(\theta, h) = h \\ z(\theta, h) = r \cdot sin\theta \end{cases} \tag{7.13}$$

One of the most natural choices for assigning texture space to the cylinder would be to use

$$\begin{cases} u = \frac{\theta}{360} \\ v = \frac{h}{H} \end{cases} \tag{7.14}$$

This lets u vary linearly from 0 to 1 as θ varies from 0 to 360^o and lets v vary from 0 to 1 as h varies from 0 to H. The complete transformation from texture space to object space is:

$$\begin{cases} x(u, v) = r \cdot cos(360 \cdot u) \\ y(u, v) = Hv \\ z(u, v) = r \cdot sin(360 \cdot u) \end{cases} \tag{7.15}$$

This has the effects of pasting the texture onto the cylinder without any distortion beyond being scaled to cover the cylinder; the right and left boundaries meet at the front of the cylinder along the line where *x=0* and *z=r*.

The new texture mapping algorithm for a cylinder can be described in Figure 7.14.

1. A cylinder is defined in object space with coordinates (*x*, *y*, *z*), and the corresponding texture is defined in texture space with coordinates (*u*, *v*). The texture is parameterized by Equation (7.15) to generate the intermediate data.

2. The intermediate data will be transformed according to the geometrical and projection transformation of the cylinder by Equation (7.7).

3. The SVG pattern is generated from the transformed texture.

4. Finally the texture is wrapped onto the cylinder by adding the pattern to the cylinder.

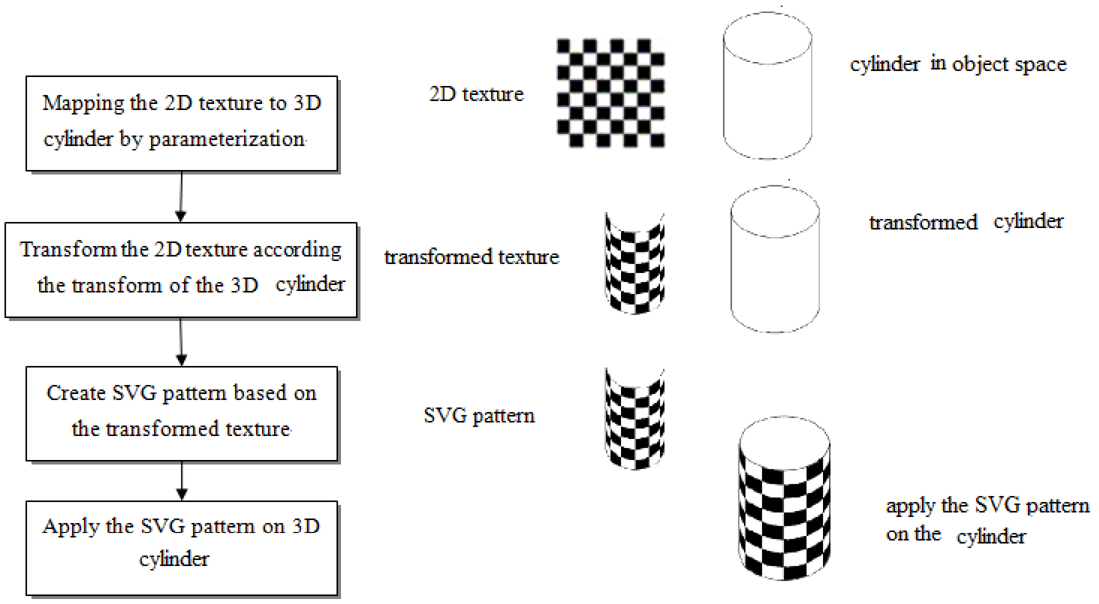

Mapping the 2D texture to 3D cylinder by parameterization
Transform the 2D texture according the transform of the 3D cylinder
Create SVG pattern based on the transformed texture
Apply the SVG pattern on 3D cylinder
2D texture
transformed texture
SVG pattern
cylinder in object space
transformed cylinder
apply the SVG pattern on the cylinder

Figure 7.14 New texture mapping method for a cylinder in the SVG GL

The proposed algorithm is used to generate texture for a cylinder, and the result is compared with the texture mapping created by using SVG pattern directly (without transform according to the cylinder). Figure 7.15(a) shows the texture is wrapped on a cylinder by using the new texture mapping algorithm. The cylinder is rotated around *x*-axis, *y*-axis, and *z*-axis respectively. Figure 7.15(b) shows the texture applying SVG pattern directly on the cylinder, and this cylinder is also rotated around *x*-axis, *y*-axis, and *z*-axis respectively.

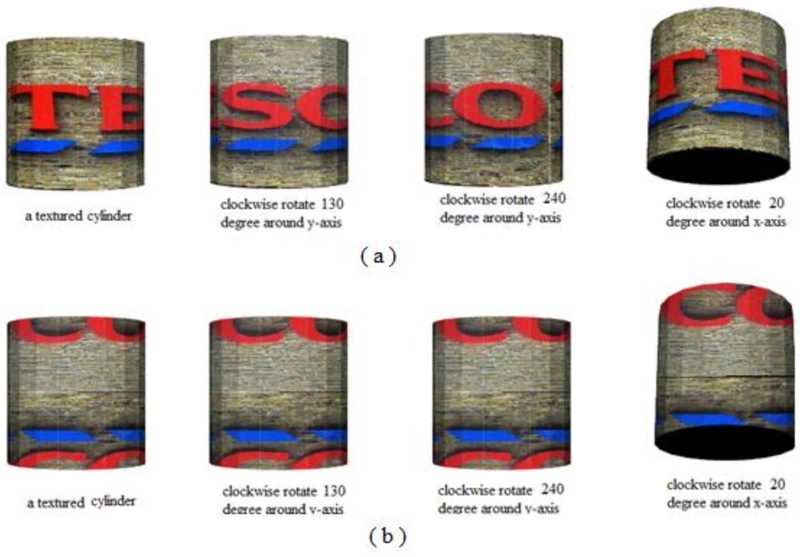

a textured cylinder
clockwise rotate 130 degree around y-axis
clockwise rotate 240 degree around y-axis
clockwise rotate 20 degree around x-axis
(a)
a textured cylinder
clockwise rotate 130 degree around y-axis
clockwise rotate 240 degree around y-axis
clockwise rotate 20 degree around x-axis
(b)

Figure 7.15 Compare texture mapping create by new algorithm for a cylinder and by using SVG pattern directly

By comparing the Figure 7.15(a) and Figure 7.15(b), it shows applying SVG pattern directly on the cylinder will create distort when the cylinder is transformed in 3D space. But the new texture mapping algorithm for a cylinder can generate realistic texture for the cylinder even the cylinder is transformed in 3D space.

7.9 A New Texture Mapping Algorithm for a Cone in the SVG GL

A new texture mapping algorithm for a cone is proposed and developed in this section. This new algorithm is based on the algorithm proposed in Section 7.4. The parameterization equation from a 2D texture to a cone will be used at the first step.

A cone of height H centered at the origin and located around the ***y*** axis has the following parametric equation of a cone can be defined as

$$\begin{cases} x = \frac{H-h}{H} rcos\theta \\ y = h \qquad\qquad 0 \le h \le H \\ z = \frac{H-h}{H} rsin\theta \;\; 0 \le \theta \le 2\pi \end{cases} \tag{7.16}$$

One of the most natural choices for assigning texture space to the cylinder would be to use

$$\begin{cases} u = \frac{\theta}{360} \\ v = \frac{h}{H} \end{cases} \tag{7.17}$$

This lets u vary linearly from 0 to 1 as θ varies from 0 to 360^{o} and lets v vary from 0 to 1 as h varies from 0 to H. The complete transformation from texture space to object space is:

$$\begin{cases} x = \frac{H-h}{H} rcos(360 \cdot u) \\ y = Hv \\ z = \frac{H-Hv}{H} rsin(360 \cdot u) \end{cases} \tag{7.18}$$

This has the effects of pasting the texture onto the cone without any distortion beyond being scaled to cover the cone; the right and left boundaries meet at the front of the cylinder along the line where $x=0$ and $z=r$.

The new texture mapping algorithm for a cone can be described in Figure 7.16.

1. A cone is defined in object space with coordinates (x, y, z), and the corresponding texture is defined in texture space with coordinates (u, v). The

texture is parameterized by Equation (7.18) to generate the intermediate data.

2. The intermediate data will be transformed according to the geometrical and projection transformation of the cone by Equation (7.7).

3. The SVG pattern is generated from the transformed texture.

4. Finally the texture is wrapped onto the cone by adding the pattern to the cone.

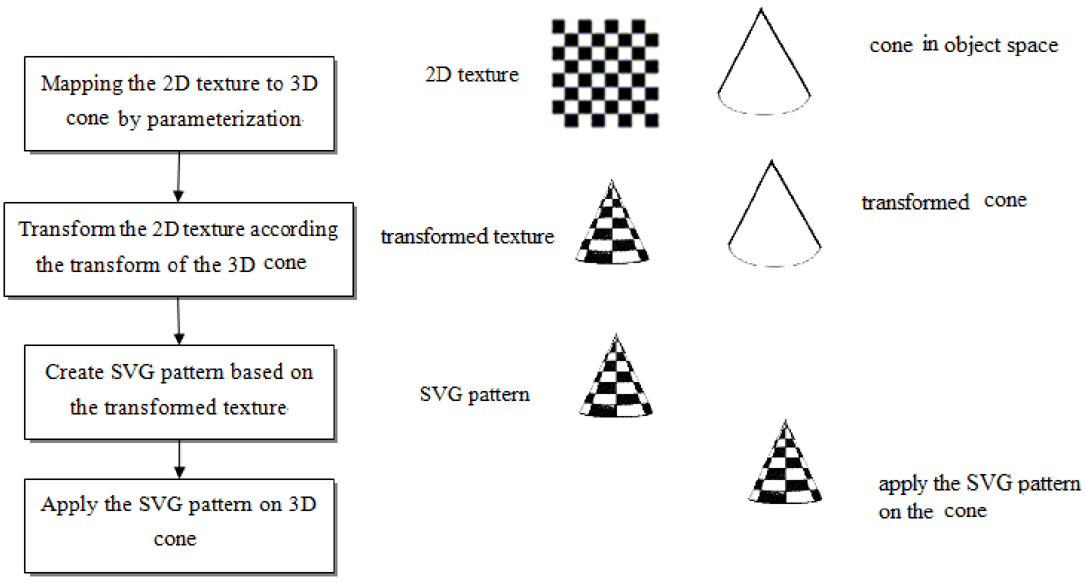

Mapping the 2D texture to 3D cone by parameterization
Transform the 2D texture according the transform of the 3D cone
Create SVG pattern based on the transformed texture
Apply the SVG pattern on 3D cone
2D texture
cone in object space
transformed texture
transformed cone
SVG pattern
apply the SVG pattern on the cone

Figure 7.16 New texture mapping method for a cone in the SVG GL

The proposed algorithm is used to generate texture for a cone, and the result is compared with the texture mapping created by using SVG pattern directly (without transform according to the cone). Figure 7.17(a) shows the texture is wrapped on a cone by using the new texture mapping algorithm for a cone. The cone is rotated around *x*-axis, y-axis, and *z*-axis respectively. Figure 7.17(b) shows the texture by applying SVG pattern directly on the cone, and this cone is also rotated around *x*-axis, *y*-axis, and *z*-axis respectively.

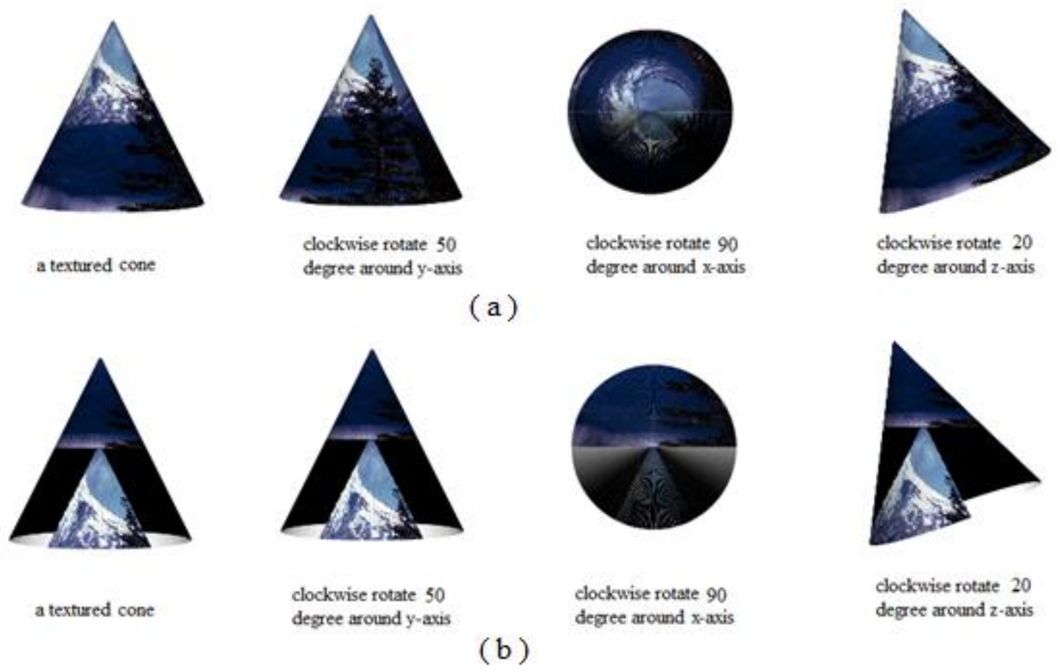

a textured cone
clockwise rotate 50 degree around y-axis
clockwise rotate 90 degree around x-axis
clockwise rotate 20 degree around z-axis
(a)
a textured cone
clockwise rotate 50 degree around y-axis
clockwise rotate 90 degree around x-axis
clockwise rotate 20 degree around z-axis
(b)

Figure 7.17 Compare texture mapping create by new algorithm for a cone and by using SVG pattern directly

By comparing Figure 7.17(a) and Figure 7.17(b), it shows applying SVG pattern directly on the cone will create distort when the cone is transformed in 3D space. But the new texture mapping algorithm for a cone can generate realistic texture for the cone even the cone is transformed in 3D space.

7.10 A New Texture Mapping Algorithm for a Bezier Surface in the SVG GL

A new texture mapping algorithm for a Bezier surface is proposed and developed in this section. This new algorithm is based on the algorithm proposed in Section 7.4. The parameterization equation from a 2D texture to a Bezier surface will be used at the first step.

A Bezier surface is defined by a set of control points (Farin, 1996). The Equation of a Bezier surface defined by *m+1* rows and *n+1* columns of control points is:

$$p(s,t) = \sum_{i=0}^{m} \sum_{j=0}^{n} B_{m,i}(s) B_{n,j}(t) p_{ij} \tag{7.19}$$

The parametric Equation of a Bezier surface can be defined as

$$\begin{cases} x = \sum_{i=0}^{m} \sum_{j=0}^{n} B_{m,i}(s) B_{n,j}(t) p_{ij} \\ y = \sum_{i=0}^{m} \sum_{j=0}^{n} B_{m,i}(s) B_{n,j}(t) p_{ij} \\ z = \sum_{i=0}^{m} \sum_{j=0}^{n} B_{m,i}(s) B_{n,j}(t) p_{ij} \end{cases} \tag{7.20}$$

One of the most natural choices for assigning texture space to the Bezier surface would be to use

$$\begin{cases} u = s \\ v = t \end{cases} \tag{7.21}$$

The complete transformation from texture space to object space is:

$$\begin{cases} x = \sum_{i=0}^{m} \sum_{j=0}^{n} B_{m,i}(u) B_{n,j}(v) p_{ij} \\ y = \sum_{i=0}^{m} \sum_{j=0}^{n} B_{m,i}(u) B_{n,j}(v) p_{ij} \\ z = \sum_{i=0}^{m} \sum_{j=0}^{n} B_{m,i}(u) B_{n,j}(v) p_{ij} \end{cases} \tag{7.22}$$

This has the effects of pasting the texture onto the Bezier surface without any distortion beyond being scaled to cover the Bezier surface.

The new texture mapping algorithm for a Bezier surface can be described in Figure 7.18.

1. A Bezier surface is defined in object space with coordinates (x, y, z), and the corresponding texture is defined in texture space with coordinates (u, v). The texture is parameterized by Equation (7.22) to generate the intermediate data.

2. The intermediate data will be transformed according to the geometrical and projection transformation of the Bezier surface by Equation (7.7).

3. The SVG pattern is generated from the transformed texture.

4. Finally the texture is wrapped onto the Bezier surface by adding the pattern to the Bezier surface.

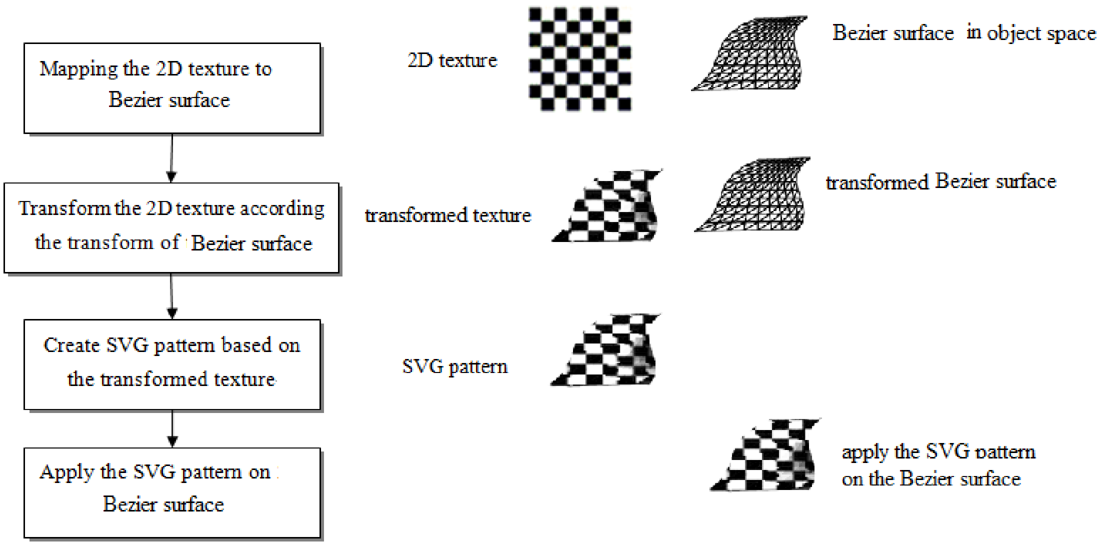

Mapping the 2D texture to Bezier surface
Transform the 2D texture according the transform of Bezier surface
Create SVG pattern based on the transformed texture
Apply the SVG pattern on Bezier surface
2D texture
Bezier surface in object space
transformed texture
transformed Bezier surface
SVG pattern
apply the SVG pattern on the Bezier surface

Figure 7.18 New texture mapping method for a Bezier surface in the SVG GL

The proposed algorithm is used to generate texture for a Bezier surface, and the result is compared with the texture mapping created by using SVG pattern directly (without transform according to the Bezier surface). Figure 7.19(a) shows the texture is wrapped on a Bezier surface by using the new texture mapping algorithm. The Bezier surface is rotated around *x*-axis, *y*-axis, and *z*-axis respectively. Figure 7.19(b) shows the texture by applying SVG pattern directly on the Bezier surface, and this Bezier surface is also rotated around *x*-axis, *y*-axis, and *z*-axis respectively.

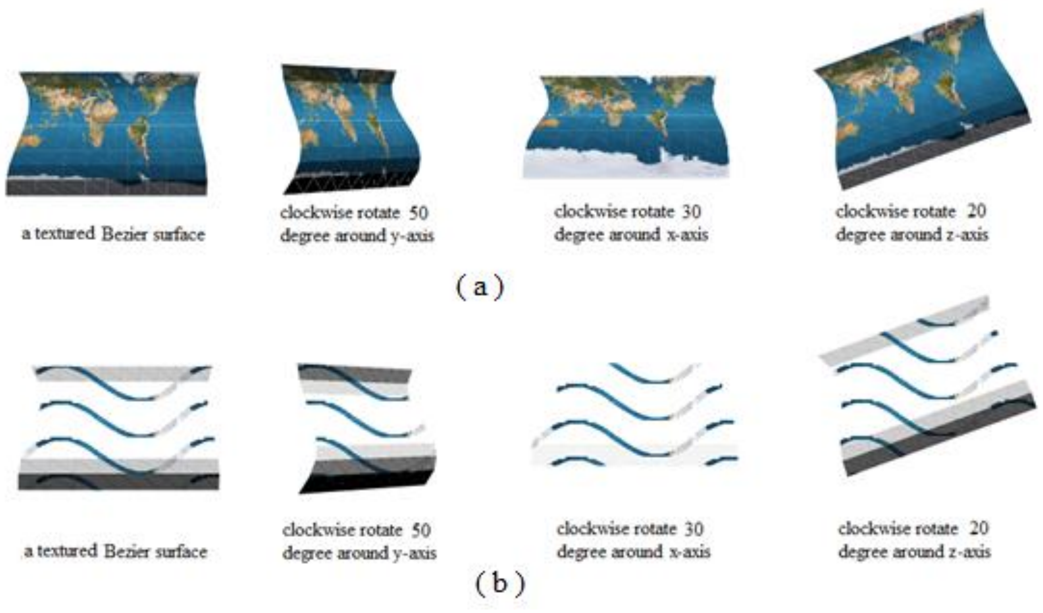

a textured Bezier surface
clockwise rotate 50 degree around y-axis
clockwise rotate 30 degree around x-axis
clockwise rotate 20 degree around z-axis
(a)
a textured Bezier surface
clockwise rotate 50 degree around y-axis
clockwise rotate 30 degree around x-axis
clockwise rotate 20 degree around z-axis
(b)

Figure 7.19 Compare texture mapping create by new algorithm for a Bezier surface and by using SVG pattern directly

By comparing Figure 7.19(a) and Figure 7.19(b), it shows applying SVG pattern directly on the Bezier surface will create distort when the cone is transformed in 3D space. But the new texture mapping algorithm for a Bezier surface can generate realistic texture for the Bezier surface even the Bezier surface is transformed in 3D space.

7.11 A New Texture Mapping Algorithm for an Extrusion in the SVG GL

A new texture mapping algorithm for an extrusion surface is proposed and developed in this section. This new algorithm is based on the algorithm proposed in Section 7.4. The parameterization equation from a 2D texture to an extrusion surface will be used at the first step.

If a space curve is expressed by **C**(s), where $\mathbf{0 \leq s \leq 1}$, the transformation matrix is $\boldsymbol{E}(t)$, then the surface of extrusion has the form:

$$\mathbf{C}(s, t) = \mathbf{C}(s)\mathbf{E}(t) \tag{7.23}$$

The parametric Equation of an extrusion surface can be defined as

$$\begin{cases} x = \mathbf{C_x}(s)\mathbf{E_x}(t) \\ y = \mathbf{C_y}(s)\mathbf{E_y}(t) \\ z = \mathbf{C_z}(s)\mathbf{E_z}(t) \end{cases} \tag{7.24}$$

One of the most natural choices for assigning a point (u, v) in texture space to a point (s, t) on the extrusion surface would be to use

$$\begin{cases} u = s \\ v = t \end{cases} \tag{7.25}$$

The complete transformation from texture space to object space is:

$$\begin{cases} x = \mathbf{C_x}(u)\mathbf{E_x}(v) \\ y = \mathbf{C_y}(u)\mathbf{E_y}(v) \\ z = \mathbf{C_z}(u)\mathbf{E_z}(v) \end{cases} \tag{7.26}$$

This has the effects of pasting the texture onto the extrusion surface without any distortion to cover the extrusion surface.

The new texture mapping algorithm for an extrusion surface can be described in Figure 7.20.

1. An extrusion surface is defined in object space with coordinates (x, y, z), and the corresponding texture is defined in texture space with coordinates (u, v). The

texture is parameterized by Equation (7.26) to generate the intermediate data.

2. The intermediate data will be transformed according to the geometrical and projection transformation of the extrusion surface by Equation (7.7).

3. The SVG pattern is generated from the transformed texture.

4. Finally the texture is wrapped onto the extrusion surface by adding the pattern to the extrusion surface.

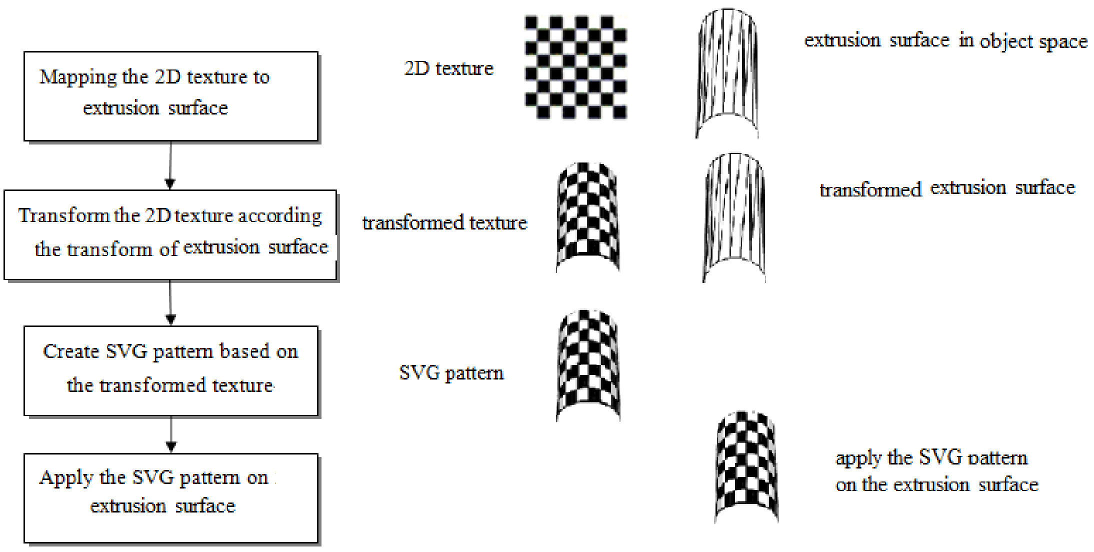

Mapping the 2D texture to extrusion surface
Transform the 2D texture according the transform of extrusion surface
Create SVG pattern based on the transformed texture
Apply the SVG pattern on extrusion surface
2D texture
extrusion surface in object space
transformed texture
transformed extrusion surface
SVG pattern
apply the SVG pattern on the extrusion surface

Figure 7.20 New texture mapping method for an extrusion surface in the SVG GL

The proposed algorithm is used to generate texture for an extrusion surface, and the result is compared with the texture mapping created by using SVG pattern directly (without transform according to the extrusion surface). Figure 7.21(a) shows the texture is wrapped on an extrusion surface by using the new algorithm. The extrusion surface is rotated around *x*-axis, *y*-axis, and *z*-axis respectively. Figure 7.21(b) shows the texture by applying SVG pattern directly on the extrusion surface, and this extrusion surface is also rotated around *x*-axis, *y*-axis, and z-axis respectively.

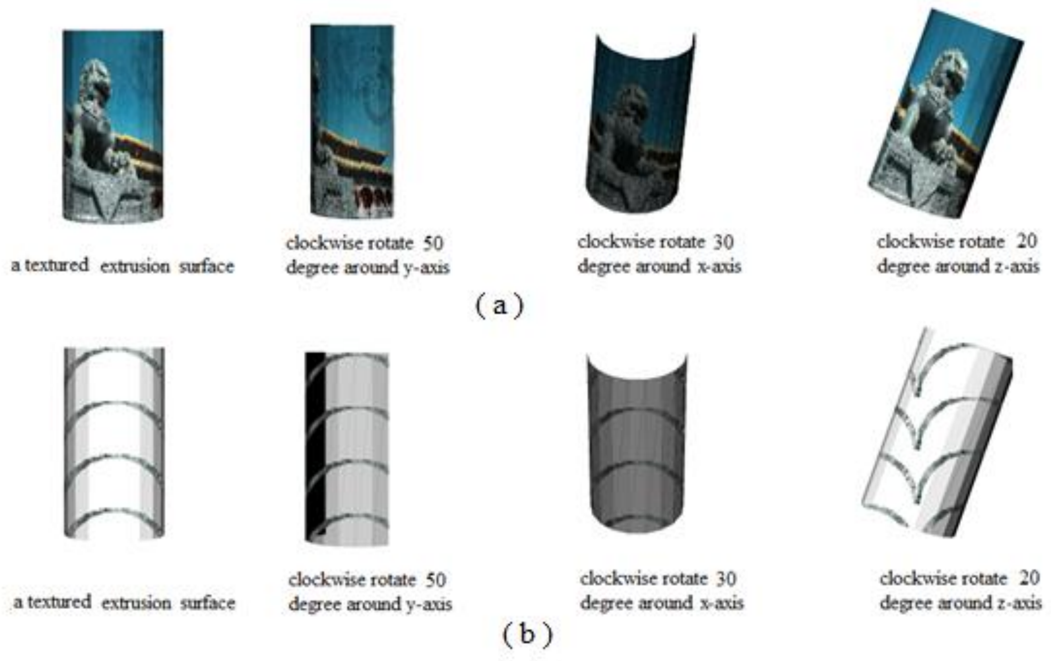

a textured extrusion surface
clockwise rotate 50 degree around y-axis
clockwise rotate 30 degree around x-axis
clockwise rotate 20 degree around z-axis
(a)
a textured extrusion surface
clockwise rotate 50 degree around y-axis
clockwise rotate 30 degree around x-axis
clockwise rotate 20 degree around z-axis
(b)

Figure 7.21 Compare texture mapping create by new algorithm for an extrusion surface and by using SVG pattern directly

By comparing Figure 7.21(a) and Figure 7.21(b), it shows applying SVG pattern directly on the extrusion surface will create distort when the cone is transformed in 3D space. But the new texture mapping algorithm for an extrusion surface can generate realistic texture for the extrusion surface even the extrusion surface is transformed in 3D space.

7.12 A New Texture Mapping Algorithm for a Revolution in the SVG GL

A new texture mapping algorithm for a revolution surface is proposed and developed in this section. This new algorithm is based on the algorithm proposed in Section 7.4. The parameterization equation from a 2D texture to a revolution surface will be used at the first step

If a space curve is expressed by $\mathbf{C}(s)$, where $\mathbf{0} \leq s \leq \mathbf{1}$, $\mathbf{C}(s)$ is rotated about an axis in space, the rotation matrix $R(t)$, then the surface of revolution surface has the form:

$$\boldsymbol{C}(s, t)=\boldsymbol{C}(s)\boldsymbol{R}(t) \tag{7.27}$$

The parametric Equation of a revolution surface can be defined as

$$\begin{cases} x = \mathbf{C}_{\mathbf{x}}(s)\mathbf{R}_{\mathbf{x}}(t) \\ y = \mathbf{C}_{\mathbf{y}}(s)\mathbf{R}_{\mathbf{y}}(t) \\ z = \mathbf{C}_{\mathbf{z}}(s)\mathbf{R}_{\mathbf{z}}(t) \end{cases} \tag{7.28}$$

One of the most natural choices for assigning a point (u, v) in texture space to a point (s, t) on the extrusion surface would be to use

$$\begin{cases} u = s \\ v = t \end{cases} \tag{7.29}$$

The complete transformation from texture space to object space is:

$$\begin{cases} x = \mathbf{C}_{\mathbf{x}}(u)\mathbf{R}_{\mathbf{x}}(v) \\ y = \mathbf{C}_{\mathbf{y}}(u)\mathbf{R}_{\mathbf{y}}(v) \\ z = \mathbf{C}_{\mathbf{z}}(u)\mathbf{R}_{\mathbf{z}}(v) \end{cases} \tag{7.30}$$

This has the effects of pasting the texture onto the revolution surface without any distortion to cover the revolution surface.

The new texture mapping algorithm for a revolution surface can be described in Figure 7.22.

1. A revolution surface is defined in object space with coordinates (x, y, z), and the corresponding texture is defined in texture with space coordinates (u, v). The texture is parameterized by Equation (7.30) to generate the intermediate data.

2. The intermediate data will be transformed according to the geometrical and projection transformation of the revolution surface by Equation (7.7).

3. The SVG pattern is generated from the transformed texture.

4. Finally the texture is wrapped onto the revolution surface by adding the pattern to the revolution surface.

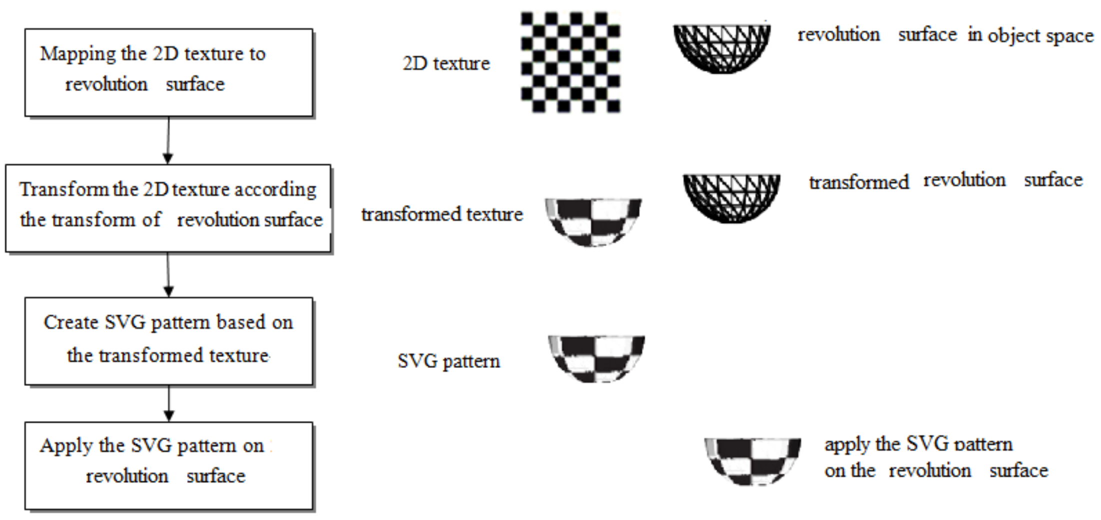

Mapping the 2D texture to revolution surface
2D texture
revolution surface in object space
Transform the 2D texture according the transform of revolution surface
transformed texture
transformed revolution surface
Create SVG pattern based on the transformed texture
SVG pattern
Apply the SVG pattern on revolution surface
apply the SVG pattern on the revolution surface

Figure 7.22 New texture mapping method for a revolution surface in the SVG GL

The proposed algorithm is used to generate texture for a revolution surface, and the result is compared with the texture mapping created by using SVG pattern directly (without transform according to the revolution surface). Figure 7.23(a) shows the texture is wrapped on a revolution surface by using the pattern based image transformed algorithm. The revolution surface is rotated around *x*-axis, *y*-axis, and *z*-axis respectively. Figure 7.23(b) shows the texture by applying SVG pattern directly on the revolution surface, and this revolution surface is also rotated around *x*-axis, *y*-axis, and *z*-axis respectively.

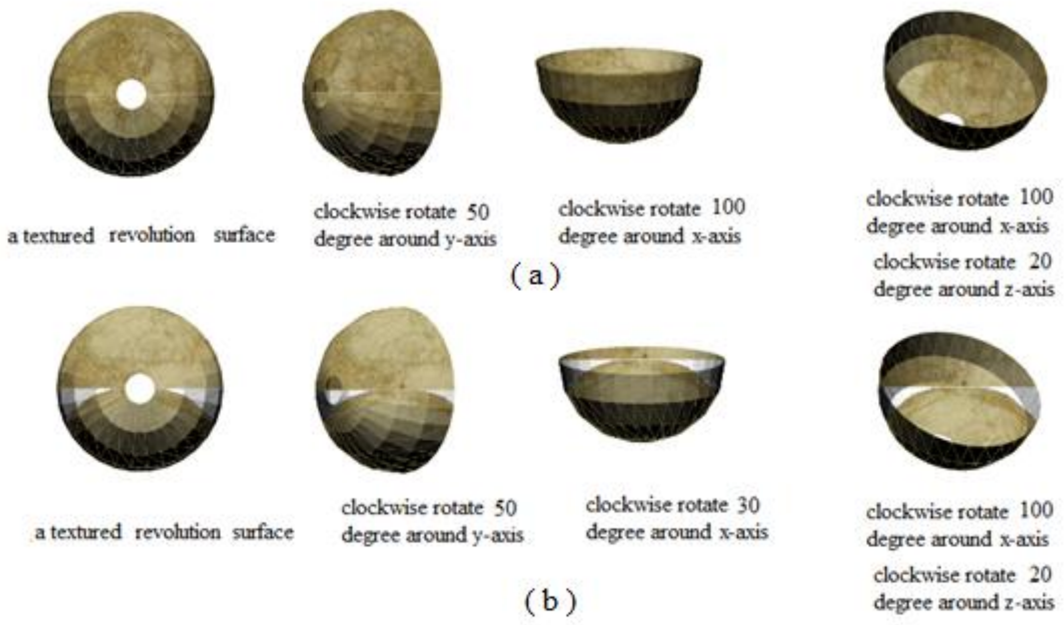

a textured revolution surface
clockwise rotate 50 degree around y-axis
clockwise rotate 100 degree around x-axis
clockwise rotate 100 degree around x-axis
clockwise rotate 20 degree around z-axis
(a)
a textured revolution surface
clockwise rotate 50 degree around y-axis
clockwise rotate 30 degree around x-axis
clockwise rotate 100 degree around x-axis
clockwise rotate 20 degree around z-axis
(b)

Figure 7.23 Compare texture mapping create by new algorithm for a revolution surface and by using SVG pattern directly

By comparing Figure 7.23(a) and Figure 7.23(b), it shows applying SVG pattern directly on the revolution surface will create distort when the cone is transformed in 3D space. But the new texture mapping algorithm for revolution surface can generate realistic texture for a revolution surface even the revolution surface is transformed in 3D space.

7.13 Summary

SVG patter is discussed first in this chapter. Although SVG pattern can be use to display 2D image, to use it to wrap 2D texture on 3D object, it still has problems.

1. Not all 3D objects surfaces are flat, so using SVG pattern directly on curved 3D surface will cause unexpected distort;

2. When 3D objects are transformed in 3D space, using the SVG pattern directly cannot create transformed texture accordingly, so the final texture mapping is incorrect.

Then existing texture mapping algorithms are also discussed. Since the existing texture mapping algorithms are pixel based, but the elementary primitive in the SVG GL is triangle. The existing texture mapping algorithms cannot be used in the SVG GL.

Due to the reason mentioned above, new texture mapping algorithms in the SVG GL are proposed and developed in this chapter. The new algorithms are based on SVG pattern and transform the 2D texture according the transformation of the 3D model. The basic idea behind this method is to transform the original texture according to the transformation of the 3D object, and then generate the pattern that will be added to the screen image. A set of new texture mapping algorithms for different 3D models in the SVG GL are proposed in this chapter, include plane, sphere, cylinder, cone, Bezier surface, extrusion surface and revolution surface. And it also proves that the proposed algorithm can generate realistic texture for the 3D object even the object is transformed in 3D space.

Chapter 8 Design and Development of the Software Environment for Validation the Proposed Framework and Algorithms

8.1 Introduction

In this PhD project a new framework-SVG GL based on SVG technology is proposed for publishing 3D for web-based applications over Internet, that can be viewed on standard web browsers (except for IE which will need a plug-in) without having to install any plug-ins. New Gouraud shading algorithm and new Phong shading algorithm in the SVG GL are proposed and developed. And a set of new texture mapping algorithms are developed for different 3D primitives and 3D complex models, including triangle, plane, sphere, cylinder, cone, extrusion, revolution, etc.

To validate the proposed framework and algorithms, a dedicated test software system- SVG 3D graphics library (S3GL) has to be developed. After a close analysis of samples collected during the problem definition stage the analyst found that all the hardware and software requirements needed for development and implementation of the S3GL are readily available in the market.

8.2 System Validation

This PhD project is mainly exploratory with some experimental validation work through a self designed and developed software system- S3GL. S3GL is developed based on the proposed framework.

S3GL will be validated firstly, to prove it can be used to create desired 3D scene. Then four 3D test applications are implemented to validate the new framework proposed in Chapter 5, the Gouraud shading and Phong shading algorithms proposed in Chapter 6, and the texture mapping algorithms developed in Chapter 7. The primary purpose of the test applications is to validate the framework, algorithms and test the performance of S3GL.

S3GL is developed following system development stages for smooth developing and running 3D models for standard web browsers. After an information gathering process from existing methods for 3D modelling for web-based applications by systematically reviewing the published literature, the system analyst saw that the new S3GL is indeed needed for generating 3D models for web-based applications.

S3GL is developed using visual C# programming language, it will help to develop 3D models for different web-based applications to realize their maximum potential in addition to their competence in the different fields.

8.3 System Requirement Analysis

8.3.1 Problem Definition

The Goals of this project are to propose a new framework-SVG GL based on SVG technology for publishing 3D for web-based applications over Internet, that can be viewed on standard web browsers; develop new Gouraud shading and Phong shading algorithms; and develop a set of new texture mapping algorithms to enhance the realism of the final rendering result. For those goals, S3GL needs to be able to perform the following operations:

1. Defining and developing 3D primitive geometries.
2. Defining and developing 3D objects through sweeping.
3. Creating free form surface by using Bezier surface.
4. Generating 3D objects through point clouds.
5. Transforming 3D objects in 3D space.
6. Projecting the 3D objects to 2D screen.
7. Illuminating and shading the 3D objects.
8. Adding texture to 3D objects to enhance the realism.
9. Adding 3D objects to an SVG file which can be rendered directly onto a standard web browser.

3D primitive geometries includes: triangle, plane, sphere, cylinder, cone, and cube. Sweeping includes extrusion and revolution. 3D objects can be rotated around the *x*, *y* and *z* axes, and translated along the *x*, *y* and *z* axes. The new texture mapping algorithms are used to add texture onto the 3D objects.

In addition to all the functions described above, the system should also be:

1. Be user-friendly to develop and use.
2. Improve the performance of 3D modelling for web-based application.
3. Reduce the file size of the 3D objects.

8.3.2 The Software System- S3GL

The Objectives of the software system- S3GL are:

(1) To create 3D objects from 3D primitive geometries, extrusion, revolution, Bezier surface, and point clouds.

(2) To integrate different shading methods such as Flat shading, Gouraud shading and Phong shading.

(3) To add texture mapping to enhance the realism of the 3D scene.

(4) To implement S3GL and validate it through development and evaluation of typical 3D web-based applications.

S3GL will cover defining, creating, transforming, rendering 3D scene to SVG file, and finally render the 3D scene on standard web browser. Moreover, special effects such as Gouraud shading, Phong shading, and texture mapping will be automated by S3GL also, and will be efficiently handled by S3GL.

To help S3GL smoothly carry out its intended purpose to meet the needs of web-based applications, the following components will be used in S3GL, each of these components relate directly to classes that are used by S3GL.

1. Scene

The scene is the entire composition of 3D objects in a 3D space. It is like a stage with three axes—*x*, *y*, and *z*. Each 3D object that the user wants to be visible should be added to the scene. If user doesn't add objects to the scene, they will not appear on the web browser.

2. Camera

As a real camera, that is somewhere in 3D space recording activity inside the scene. The camera defines the point of view from which viewer is viewing the scene. Camera in 3D space can usually do more than real camera. The camera is able to ignore objects that are not in a certain defined range. This is done for performance reasons.

3. 3D objects

A shape in 3D space is called a 3D object. A 3D object can be placed anywhere in 3D space and rotated over each of the three axes. S3GL has a set of primitive shapes. Such primitives include triangle, plane, sphere, cylinder, cone, and cube. However, working with more complex 3D objects is possible, as S3GL allows user to create 3D objects by extrusion, sweeping, Bezier surface and point clouds. User can also import 3D objects into application by using text file.

4. Material

A material is the colour, or texture that is printed on a 3D object. When a 3D object doesn't have a material applied, it will be invisible. There are a variety of materials available to be used. For example, a very simple material is a colour; a more advanced example of a material can be a raster image

5. Light

The only available light in S3GL is a point light. This is a point somewhere in 3D space that defines the origin of a light source. Each shader in S3GL requires a point light. S3GL does not provide other types of lights such as spot light and directional light.

The advantages of S3GL include:

(1) Enables easy and fast creating 3D objects.

(2) Provides multiple 3D objects definition methods, including primitives, sweeping, Bezier surface, and point clouds.

(3) Implements efficient 3D transformation methods.

(4) Implements perspective projection methods.

(5) Adds different shading methods.

(6) Implements texture mapping methods.

(7) Reduce the final file size of 3D scene.

(8) Render 3D scene on standard web browser without the need of any plug-ins (except for IE).

8.4 System Design

S3GL was designed in Microsoft Visual Studio 2010. The system design phase describes the functional capabilities of the system.

8.4.1 Development Environment

C# is a .NET, general-purpose, object-oriented programming language. The advantages of C# over other languages such as C++, Java, and JavaScript are:

1. It can be used to develop both window-based and web applications.

2. C# being a .NET language, supports language interoperability, i.e. C# can access code written in any .NET compliant language and can also inherit the

classes written in these languages.

3. It can access to all the .NET Framework class libraries, which are quite extensive. While these libraries might support specific features better than other programming language.

4. It is a compiled computer programming language, the source code will be compiled once when the program first run. This means it is much more efficient than an interpreted programming language.

5. It has native garbage-collection.

Due to all the advantages of C#, the system is developed by using C#, and Microsoft Visual Studio 2010 is used as the development environment for both S3GL development and web application test.

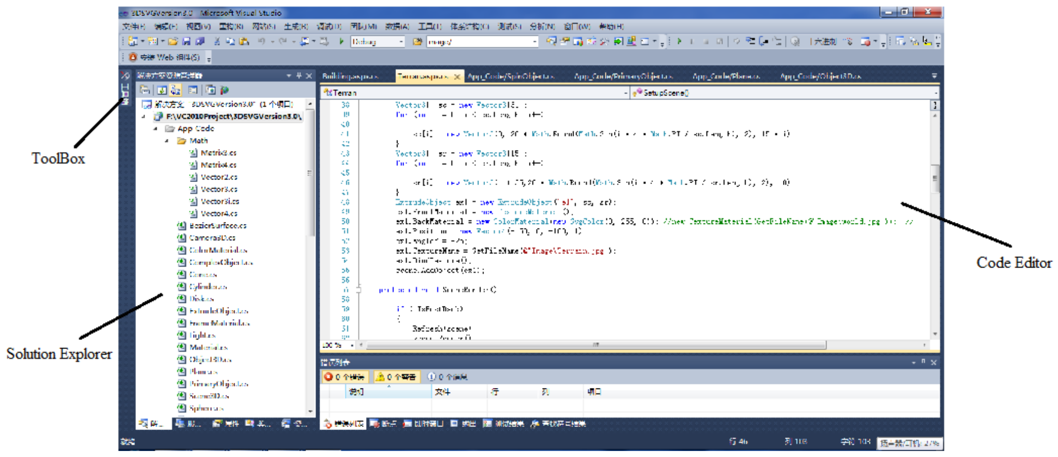


Figure 8.1 Visual Studio 2010 integrate development environment

Microsoft Visual Studio is an integrated development environment (IDE) from Microsoft. It is used to develop computer programs for Microsoft Windows, as well as web sites, web applications and web services. Visual Studio uses Microsoft software development platforms such as Windows API, Windows Forms, Windows Presentation Foundation, Windows Store and Microsoft Silverlight.

The Visual Studio IDE is shown in Figure 8.1. It includes a code editor, a toolbox, and a solution explore. The code editor is where developers write code that makes everything in the application work; the toolbox is a palette of developer objects, or controls, that are placed on the forms or web pages; the solution explorer is a section that is used to view and modify the content of the project. The integrated debugger works both as a source-level debugger and a machine-level debugger. Other built-in tools include a forms designer for building GUI applications, web designer, class designer, and database schema designer.

Visual Studio supports different programming languages and allows the code editor and debugger to support nearly any programming language, provided a language-specific service exists. Built-in languages include C, C++ and C++/CLI (via Visual C++), VB.NET, C#, and F#.

8.4.2 Classes Design and Definition

The followings are the designs of the main classes that shall be used to store the data in S3GL:

1. Triangle

A triangle is the simplest shape among the primitives. There are totally 4 arguments (Table 8.1) to be passed to the Triangle constructor.

Table 8.1 Triangle's argument

Argument	Data type	Default value	Description
id	string		Defines a unique id to distinguish from other objects
va	point	(0,0,0)	Sets the coordinate of triangle vertex
vb	point	(0,1,0)	Sets the coordinate of triangle vertex
vc	point	(1,0,0)	Sets the coordinate of triangle vertex

2. Plane

A plane looks like a rectangle if it does not rotate over the *x*-axis or *y*-axis. There are totally 3 arguments (Table 8.2) to be passed to the Plane constructor.

Table 8.2 Plane’s argument

Argument	Data type	Default value	Description
id	string		Defines a unique id to distinguish from other objects
width	int	0	Sets the desired width of the plane
height	int	0	Sets the desired height of the plane

3. Sphere

There are 4 arguments need to be set for a Sphere constructor. They are listed and described in Table 8.3.

Table 8.3 Sphere’s argument

Argument	Data type	Default value	Description
id	string		Defines a unique id to distinguish from other objects
radius	int	10	Sets the radius of the sphere
segmentsA	int	10	Sets the number of segments horizontally
segmentsP	int	10	Sets the number of segments vertically

4. Cylinder

There are more arguments for creating a cylinder. Table 8.4 show all the arguments need to be passed to a Cylinder constructor.

Table 8.4 Cylinder's argument

Argument	Data type	Default value	Description
id	string		Defines a unique id to distinguish from other objects
top radius	int	10	Sets the desired top radius of the cylinder
bottom radius	int	10	Sets the desired bottom radius of the cylinder
height	int	10	Sets the desired height of the cylinder
segmentsA	int	0	Sets the number of segments horizontally

5. Cone

The following arguments are available in the Cone constructor (Table 8.5).

Table 8.5 Cone's argument

Argument	Data type	Default value	Description
id	string		Defines a unique Id to distinguish from other objects
radius	int	10	Sets the desired radius of the cone
height	int	10	Sets the desired height of the cone
segmentsA	int	0	Sets the number of segments horizontally

The arguments for Cone constructor are similar with those for Cylinder constructor. Since cone always has a converged top, the radius in the cone constructor is set for the bottom of the cone.

6. Cube

Instantiate a cube is similar to instantiate the previously discussed primitives. Table 8.6 shows all arguments for instantiating a cube.

Table 8.6 Cube’s argument

Argument	**Data type**	**Default value**	**Description**
id	string		Defines a unique id to distinguish from other objects
width	int	10	Sets the desired width of the cube
depth	int	10	Sets the desired depth of the cube
height	int	10	Sets the desired height of the cube

7. Complexobject

ComplexObject is created from primitives introduced above. Table 8.7 shows all arguments for a constructor of ComplexObject. An AddObject() function is provided for ComplexObject to add primitives. Different primitives put together to build a complex object.

Table 8.7 Complexobject’s argument

Argument	**Data type**	**Default value**	**Description**
id	string		Defines a unique id to distinguish from other objects
primitiveobjects	List<primitive>	null	Primitives used to create a Complexobject

8. Other 3D objects in S3GL

Besides the 3D objects discussed above, there are other 3D objects in the S3GL. ExtrusionObject creates 3D objects by extruding a 2D shape along a given route; RevolutionObject creates 3D objects by revolving a 2D shape around coordinate axis; and BezierSurface is used to create free form surface in 3D space.

ExtrusionObject takes 4 argument, they are shown in Table 8.8

Table 8.8 ExtrusionObject's argument

Argument	Data type	Default value	Description
id	string		Defines a unique id to distinguish from other objects
contour	Point list	null	Sets the contour of the 2D shape
route	Point list	null	Sets the extruding route

The arguments for a RevolutionObject constructor are shown in Table 8.9.

Table 8.9 RevolutionObject's argument

Argument	Data type	Default value	Description
id	string		Defines a unique id to distinguish from other objects
contour	point list	null	Sets the contour of the 2D shape
axis	chart	'y'	Sets the coordinate axis revolving around
segmentsA	int	0	Sets the number of segments around the revolving direction

Table 8.10 shows all the arguments for instantiating a Bezier Surface.

Table 8.10 BezierSurface's argument

Argument	Data type	Default value	Description
id	string		Defines a unique id to distinguish from other objects
control points	Point list	null	Sets the set of the control points
segmentsA	int	0	Sets the number of segments horizontally
segmentsP	int	0	Sets the number of segments vertiaclly

In the S3GL bicubics patches are used for the Bezier surface, which means the order of the Bezier surface is 3, and 16 control points in total are used to define a Bezier surface

9. Materials

There are 3 types of materials in the S3GL: FrameMaterial, ColourMaterial, and TextureMaterial.

The different between FrameMaterial and ColourMaterial is: FrameMaterial connects the points by lines, but ColourMaterial connects the points by triangles. TextureMaterial creates a material that is made of a bitmap, and then the material is wrapping onto the surface of a 3D model to achieve more realistic effect than FrameMaterial and ColourMaterial.

The constructor of FrameMaterial has 2 arguments shown in Table 8.11.

Table 8.11 FrameMaterial's argument

Argument	Data type	Default value	Description
colour	Colour	(0,0,0)	Defines the colour of the frame
thickness	int	1	Defines the thickness of the frame

The constructor of ColourMaterial has only one argument (Table 8.12).

Table 8.12 ColourMaterial's argument

Argument	Data type	Default value	Description
colour	Colour	(0,0,0)	Defines the colour of the surface

The constructor of TextureMaterial also has only one argument (Table 8.13).

Table 8.13 TextureMaterial's argument

Argument	Data type	Default value	Description
filename	string	null	Defines the filename of the specified texture

10. Light

A point light source is defined in S3GL; Table 8.14 shows the arguments for a Light's constructor.

Table 8.14 Light's argument

Argument	Data type	Default value	Description
direction	vector	(1,0,0)	The direction vector of the light
position	vector	(0,0,100)	The position vector of the light
colour	Colour	(1,1,1)	The colour of the light
ambient	vector	(1,1,1)	Ambient component of the light
diffuse	vector	(0,0,0)	Diffuse component of the light
specular	int	1	Specular coefficient of the light

11. Camera

The following arguments are needed to initiate a camera in S3GL (Table 8.15).

Table 8.15 Camera's argument

Argument	Data type	Default value	Description
direction	vector	(1,0,0)	The direction vector of the camera
position	vector	(0,0,0)	The position vector of the camera
focus	int	100	The focus length of the camera
near	int	100	The near surface of the camera
far	int	1000	The far surface of the camera

12. Scene

The scene in the S3GL is a canvas on which all 3D objects are rendered. The following arguments are needed to initiate a scene (Table 8.16).

Table 8.16 Scene argument

Argument	Data type	Default value	Description
width	int	500	The width of the scene
height	int	500	The height of the scene
bcolour	Colour	(1,1,1)	The background colour of the scene
filename	string	null	The final SVG file name
camera	Camera	null	The camera in the scene
light	Light	null	The light source in the scene
objects	List<object>	null	All objects in the scene

Except for all the classes mentioned above, a math library was also developed in the S3GL. 2D vector, 3D vector, 3x3 matrixes, and 4x4 matrixes are developed in the 3DMath library, all related vector, and matrix operations are implemented.

8.5 System Implementation

8.5.1 Web Server

S3GL is developed by using C# based on ASP.NET technology. ASP.NET is an open source server-side web application framework designed for web development to produce dynamic web pages. It was developed by Microsoft to allow programmers to build dynamic web sites, web applications and web services.

ASP.NET is a server-side web application framework designed for web development to produce dynamic web pages. Since ASP.NET is run on server-side, the web hosting provider must configure its servers appropriately to execute the necessary source code. In addition to providing connectivity, ASP.NET web hosting providers also provide the technological basis for the ASP.NET creative process.

ASP.NET is a cross-platform technology; the application based on ASP.NET can run on almost all different platform, includes web server that has the .NET Framework and Internet Information Services (IIS) installed, and non-Microsoft sever that installed MONO (Delahunty, 2005) platform. The web server that supports ASP.NET includes:

1. IIS, Internet Information Services, is a free component bundled with Windows System.
2. Apache, a classic web application server, ASP.NET can run on Apache that has MONO installed.
3. XSP, a server with an independent standard. It is written in C#, and can be used to run ASP.NET application.
4. Nginx, a high-performance HTTP server that supports ASP.NET and applications.
5. Jexus, is based on .NET/MONO, supports ASP.NET and applications.

Since S3GL is based on ASP.NET, so all applications developed by using S3GL can run on almost all different platforms.

8.5.2 Support Platform

The application developed based on S3GL is running on the web server (Figure 8.2). The client side sends request for a S3GL web page to the web server, the S3GL application will run on the web server, and the render result will be sent back to the client side as a normal SVG file. The client side manipulation on the 3D scene will be sent to the server side, after processing by S3GL on the server, the result will be sent back to client side again. This means S3GL has no special requirement for the client side, all client side platform that support normal SVG can view the 3D model rendered by S3GL.

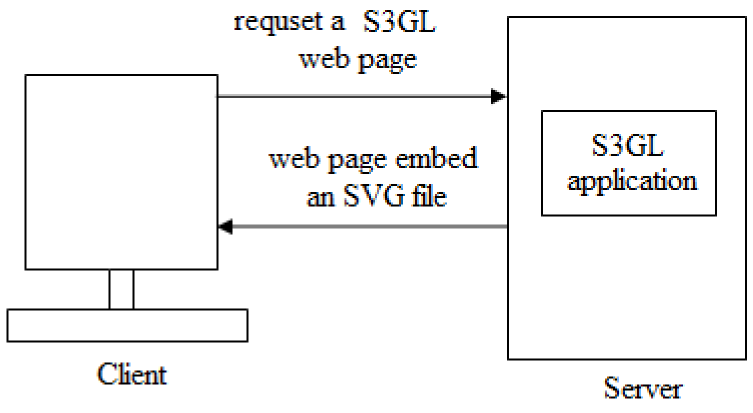


Figure 8.2 S3GL running as a server side application

Platforms on the client side that S3GL can be used are summarized in Table 8.17. The data in the table is based on the latest versions of the respective web browsers as of the writing of this thesis.

All modern browsers support rendering SVG. Internet Explorer, up to and including IE8, was the only major browser not to provide native SVG support. IE8 and older require a plug-in to render SVG content. For mobile web developers wondering about compatibility, iOS 3.2+, Opera Mini 5+, Opera Mobile 10+ and Android 3+ also support rendering SVG graphics.

Statistics show that 84.71% of Internet users have a web browser that supports or partial support SVG (Caniuse, 2013).

The statistics therefore form an upper bound on what percentage of users can run 3D graphics applications built upon S3GL.

In a web browser that supports SVG, no extra software has to be installed for S3GL applications to be able to run. For Internet Explorer, that does not support SVG natively, there are number of plug-ins available to assist, including: Adobe SVG Viewer, SVG Web, or Google Chrome Frame.

Table 8.17: Supported operating systems and web browsers

Operating System	Browser	SVG	Client side manipulation
Windows	Internet Explorer	Support via plug-in	Yes
	Chrome	Yes	Yes
	Firefox	Yes	Yes
	Opera	Yes	Yes
	Safari	Yes	Yes
Mac OS X	Chrome	Yes	Yes
	Firefox	Yes	Yes
	Opera	Yes	Yes
	Safari	Yes	Yes
Linux	Chrome	Yes	Yes
	Firefox	Yes	Yes
	Opera	Yes	Yes
Android	Build-in web browser	Yes	Yes
	Opera	Yes	Yes
iOS	Safari	Yes	Yes

8.6 System Testing

The main technical activities in software testing process include planning, generating and selecting test cases, preparing test environment, testing the program under test and observing its dynamic behaviour, analyzing the observed behaviour on each test case, report test results (Figure 8.3).

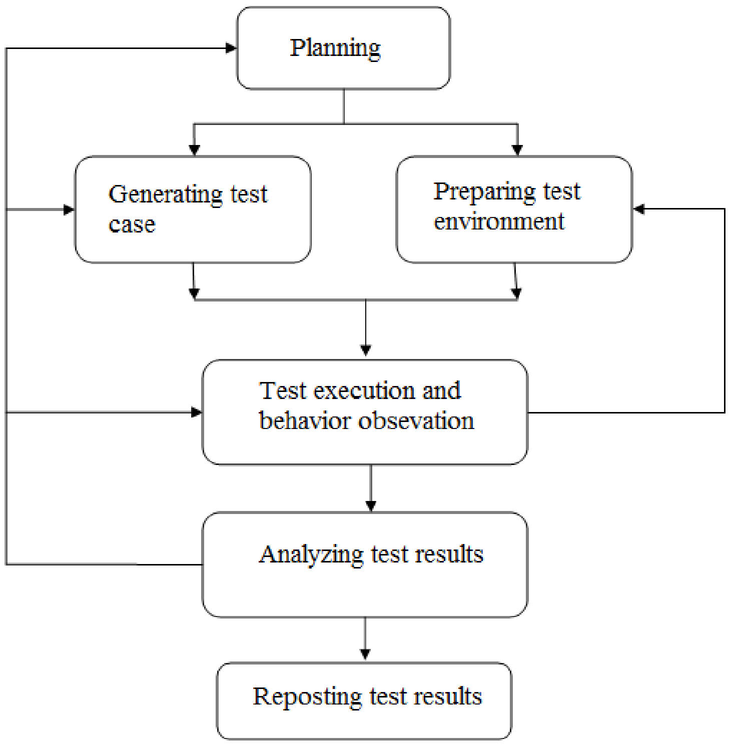


Figure 8.3 Activities in software testing process

The system testing is carried out using Use case based testing. Use cases have been derived from the requirements, and then system testing can be performed by testing that the system satisfies each of the Use cases.

In order to test the developed S3GL, a test Use case is design, and the system testing is carried out as following:

1. Design the test plan. Descript the 3D scenario that will be rendered on the web page.
2. Prepare the test environment. All the test is carry on Microsoft Visual Studio 2010.
3. Prepare the test Use case. Define 3D models by using the functions provided by S3GL.
4. Execute the test and observe the results;
5. Analyze the test results.

Use case 1: Adding a 3D object on web page

The first Use case is described in Table 8.18. A 3D object is defined by using the primitives in S3GL. After geometrical transformation and perspective projection, the 3D object will be mapped to SVG viewport, and rendered on the web browser directly.

Table 8.18 Use case 1-Adding a 3D object on web page.

ID	Use case 001
Title	Adding a 3D object on web page
Actor	User
Description	A cylinder is created by S3GL, and rendered on web browser directly.

The result of the Use case 1 is shown in Figure 8.4. Since SVG supports many UI events and pointing events, mouse events are used to change the parameters (position, view angle) of the 3D camera in S3GL. By using the mouse, user can navigate the 3D scene, and view the 3D model from different angle. The test result shows that the S3GL can be used to develop 3D object, and render the result directly on web page. And the 3D scene can by navigated by using mouse.

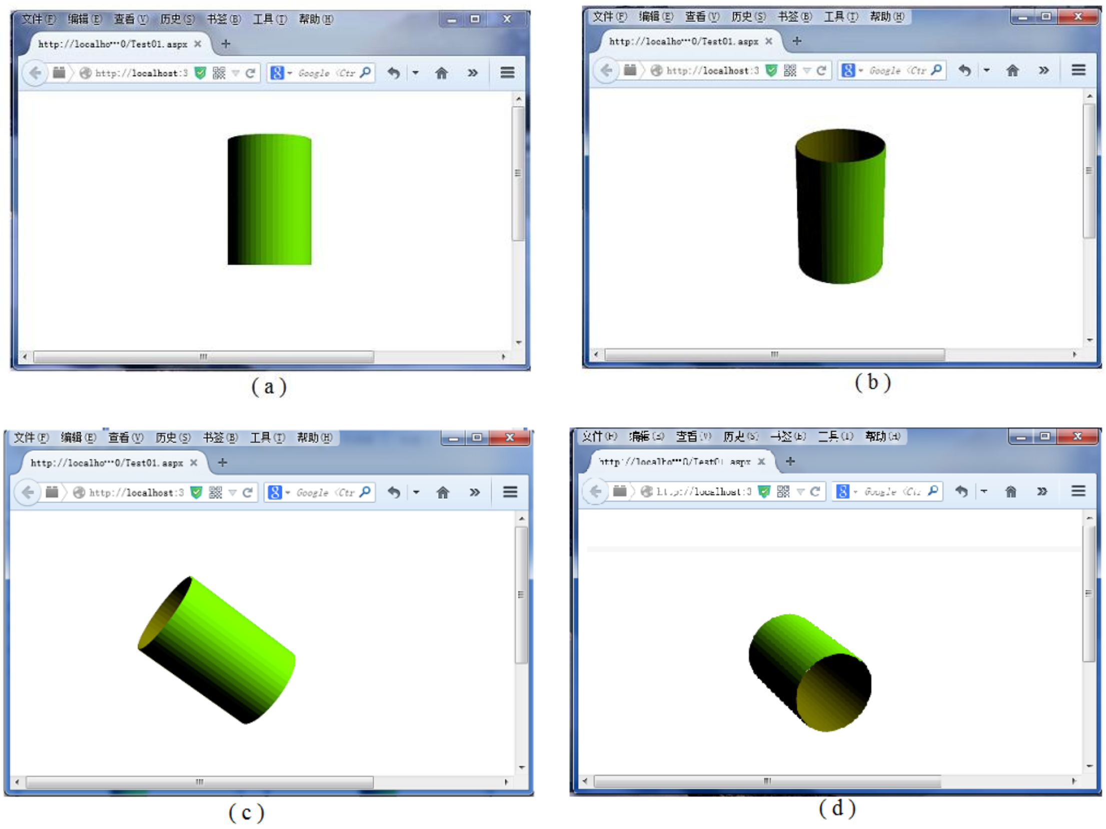


Figure 8.4 Screenshot of use case 1

Use case 2: Adding multiple 3D objects on web page

The second Use case is described in Table 8.19.

Table 8.19 Use case 2- Adding multiple 3D objects on web page.

ID	Use case 002
Title	Adding multiple 3D objects on web page
Actor	User
Description	A cone and a sphere are created by S3GL, and rendered on web browser directly.

The result of the Use case 2 is shown in Figure 8.5. By using the mouse, user can navigate the 3D scene, and view the 3D model from different angle. The test result shows that the S3GL can be used to develop 3D objects, and render the result directly on web page. The position relationships between objects are displayed correctly.

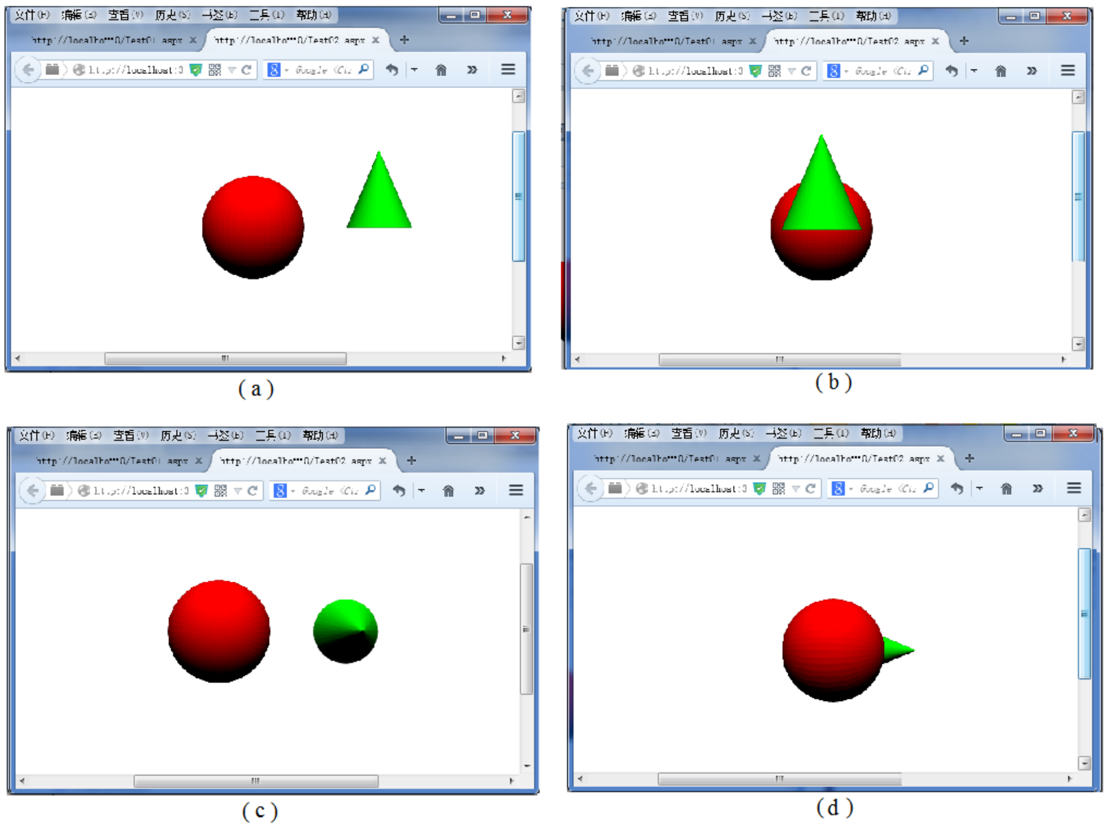


Figure 8.5 Screenshot of use case 2

The test result shows that the S3GL can be used to develop 3D scene, and render the result directly on web page. And the 3D scene can by navigated by using mouse.

After the validation of the testing environment, four 3D test applications will be implemented to test the algorithm proposed in Chapter 5, 6, 7.

8.7 The Validation of the Theory and Algorithms Using Software System

Four 3D test applications are implemented based on S3GL to validate the new framework proposed in Chapter 5, the new Gouraud shading and Phong shading algorithm proposed in Chapter 6, and the new texture mapping algorithms developed in Chapter 7.

1. 3D static objects.
2. A Gouraud shading box
3. A Phong shading box.
4. A 3D plant.

Those applications are tested on a Dell Laptop with Intel Core i5 CPU and with Windows 7 operating system. The version of the plug-in to view SVG was Adobe SVG Viewer 3.01 for Internet Explorer 9.0. Those case studies are also tested also on other major web browsers, includes: Firefox, Chrome, Opera, and Safari, that no plug-in needed.

8.7.1 3D Static Objects

The first test application is 3D static objects consist of a table, a lamp, a sphere, and a pen case (Figure 8.6). The purpose of this test application is to validate the algorithm provided in Chapter 5 and Chapter 7, and it will have the functionality outlined below.

1. Build 3D SVG model.
2. Rendering 3D object on web browser.
3. Using texture on specified 3D object.
4. Interactive manipulation.

The reasons of choose this 3D scene are:

1. It can be used to validate whether different 3D objects can be created and rendered correctly.
2. It can be used to validate whether the position relationship between objects can be rendered correctly.
3. It can be used to validate whether the texture mapping algorithm can generate correct texture for different object.

The 3D static objects can be viewed from different angle. During the test, the average number of second per frame will be recorded, and the file size of the SVG file generated by S3GL will also be recorded for further analysis.

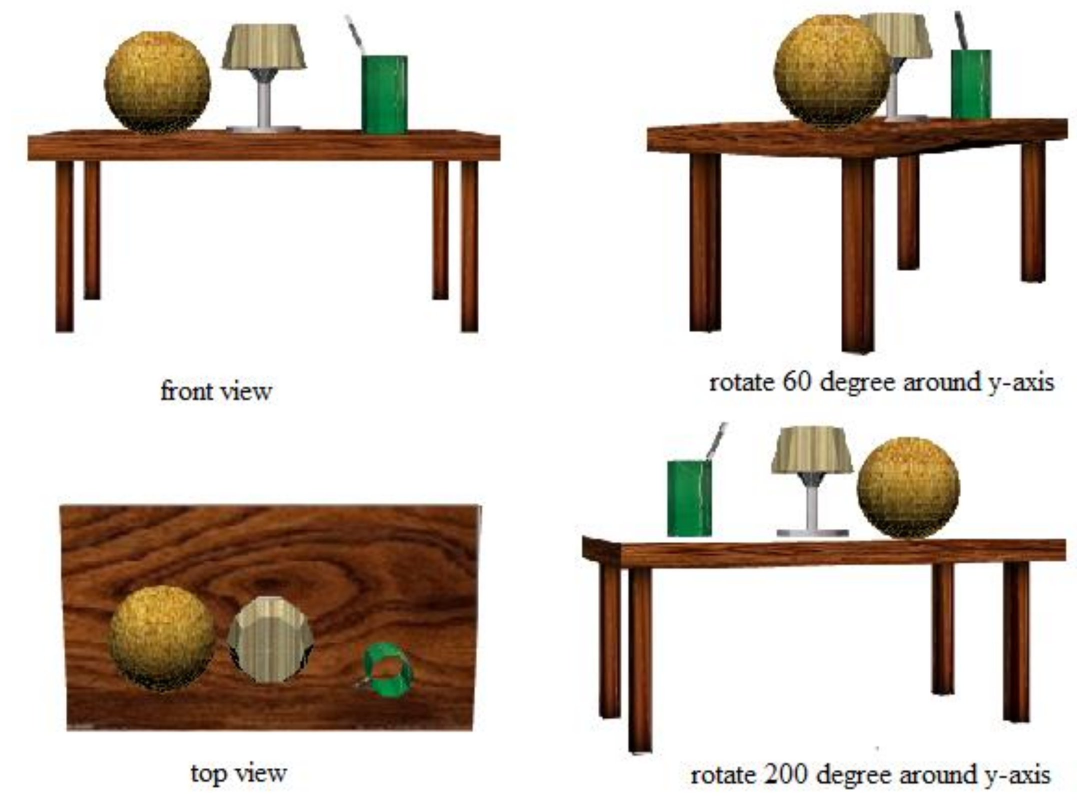

front view
rotate 60 degree around y-axis
top view
rotate 200 degree around y-axis

Figure 8.6 3D static objects

The test result shows that the right depth test, and the objecst are rendered correctly. This application also apply texture mapping on sphere, cube, and cylinder. It generates right texture for each object, and improves the realism of the scene. The user can navigate the 3D scene and rotate the camera by using the mouse.

The average number of second per frame that is achieved with this test application on different web browser is shown in Table 8.20. As can be seen in the table, the test application works well on different web browser, and the render times are also similar on different web browser.

Table 8.20 Static object render time

Web browser	IE	Firefox	Chrome	Safari	Opera
Render rate (seconds/frame)	0.36	0.42	0.43	0.45	0.40

Another performace measure is the amount memory the application uses. This is especially important if the developer wishes to target hardware with limited memory. The maximum amount of memory of this application is 101KB.

Figure 8.7 shows a 3D video case, the file size for such a 3D model created by X3D is 36,213 KB; and the file size created by VRML is 33,030 KB. Compare with the 3D static objects created by S3GL, the file size is only about 1/300 of the file size created by X3D or VRML.

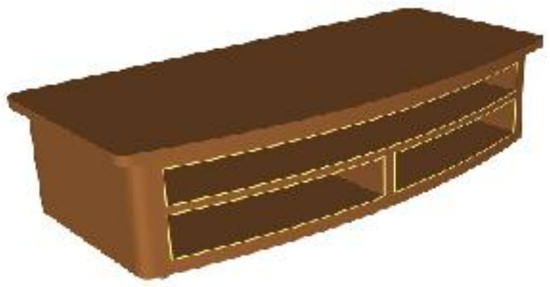


Figure 8.7 3D video case (free model download from http://www.3dcadbrowser.com)

Both those performance tests show the algorithm proposed in Chapter 5 can be used to generate 3D scene. And the texture mapping described in Chapter 7 can also work well to create realistic 3D object. By compare the final file size with X3D and VRML, it shows the S3GLcan significantly reduce the file size, which is important for web-based applcations.

8.7.2 A Gouraud Shading Box

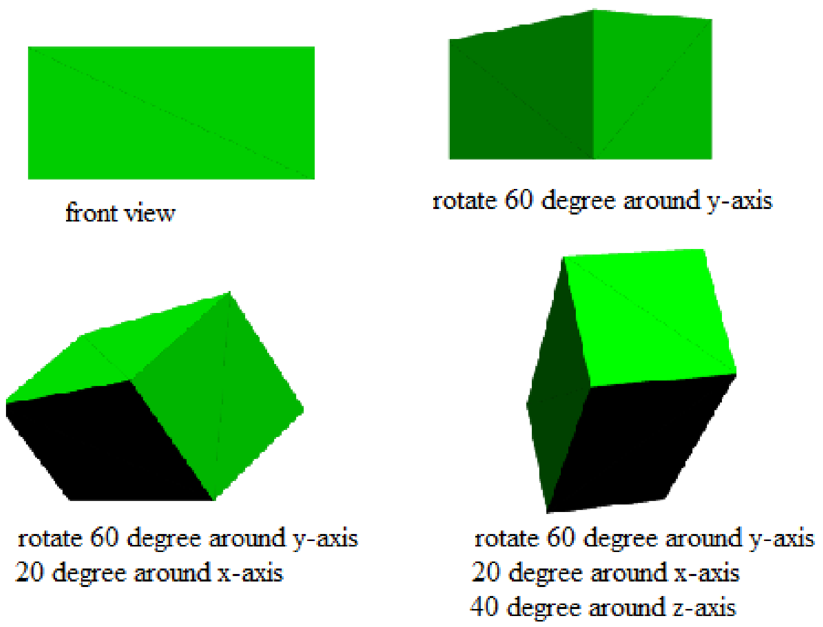
The second test application is a box rendered by Flat shading and a box rendered by the new Gouraud shading algorithm separately. The purpose of this test application is to validate the Gouraud shading algorithm proposed in Chapter 6, and it will have the functionality outlined below.

1. Build 3D SVG model.
2. Rendering 3D object on web browser.
3. Using Flat shading and Gouraud shading on 3D object.
4. Interactive manipulation.

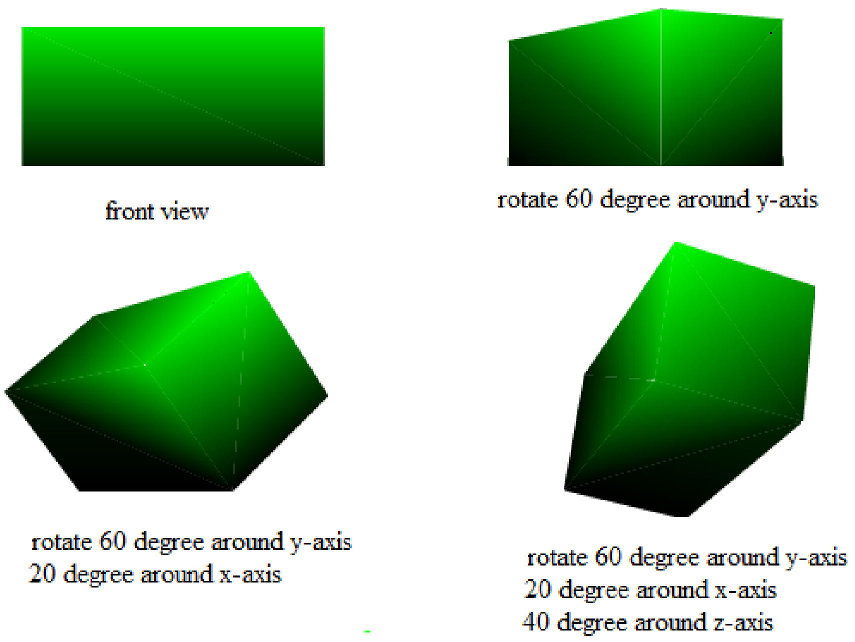
The reason of choose this scene is that it can be used to check whether the Gouraud shading algorithm proposed in Chapter 6 can generate smooth colour transitions on hard boundary.

During the test, the average number of second per frame will be recorded, and the file size of the SVG file generated by S3GL will also be recorded for further analysis.

Figure 8.8(a) shows a green box viewed from different angle, and the box is rendered by Flat shading; Figure 8.8(b) shows a box rendered by the Gouraud shading algorithm proposed in Chapter 6. The point light is defined on the top front side of the scene.


front view
rotate 60 degree around y-axis
rotate 60 degree around y-axis
20 degree around x-axis
rotate 60 degree around y-axis
20 degree around x-axis
40 degree around z-axis

(a)


front view
rotate 60 degree around y-axis
rotate 60 degree around y-axis
20 degree around x-axis
rotate 60 degree around y-axis
20 degree around x-axis
40 degree around z-axis

(b)

Figure 8.8 Flat shading vs. Gouraud shading

The test result shows that by using the new Gouraud shading algorithm, it can generate smooth colour transitions on hard boundary.

The average number of second per frame which was achieved with this test application on different web browser is shown in Table 8.21. As can be seen in the table, the test application works well on different web browser, and the render time required for Flat shading is far less than Gouraud shading algorithm since there are more arithmetic calculation involved in Gouraud shading algorithm.

Table 8.21 Flat shading and Gouraud shading render rate

Web browser	IE	Firefox	Chrome	Safari	Opera
Flat shading Render rate (seconds/frame)	0.007	0.0072	0.0067	0.007	0.0069
Gouraud shading Render rate (seconds/frame)	0.11	0.12	0.11	0.12	0.12

The maximum amount of memory of Flating shading is 2 KB, and maximum amount of memory of area interpolation Gouraud shading algorithm is 4KB. This shows the Gouraud shading algorithm will use more memory than Flat shading.

Both those performance tests show the Gouraud shading algorithm developed in Chapter 6 can work well to create realistic 3D object.

8.7.3 A Phong Shading Box

The third test application is a box rendered by Flat shading and a box rendered by the new Phong shading algorithm. The purpose of this test application is to validate the Phong shading algorithm provided in Chapter 6, and it will have the functionality outlined below.

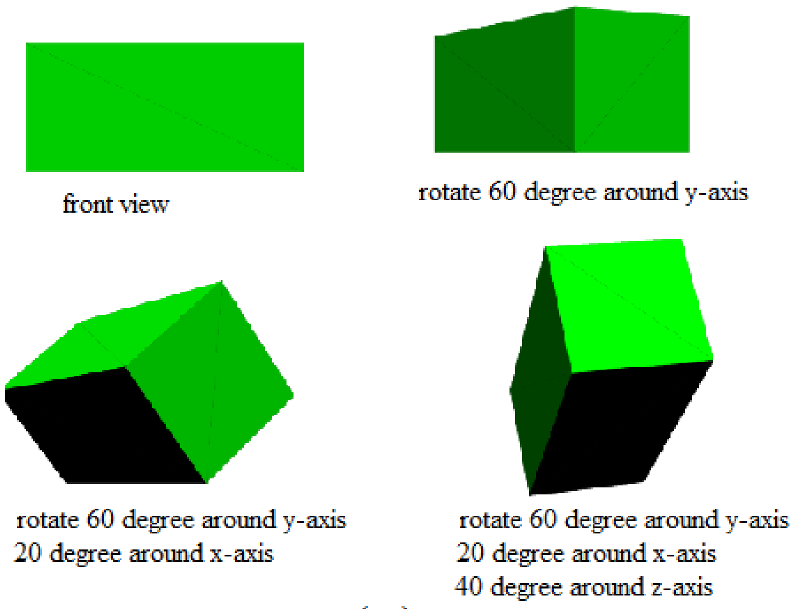
1. Build 3D SVG model.
2. Rendering 3D object on web browser.
3. Using Flat shading and Phong on 3D object.
4. Interactive manipulation.

The reasons of choose this scene are:

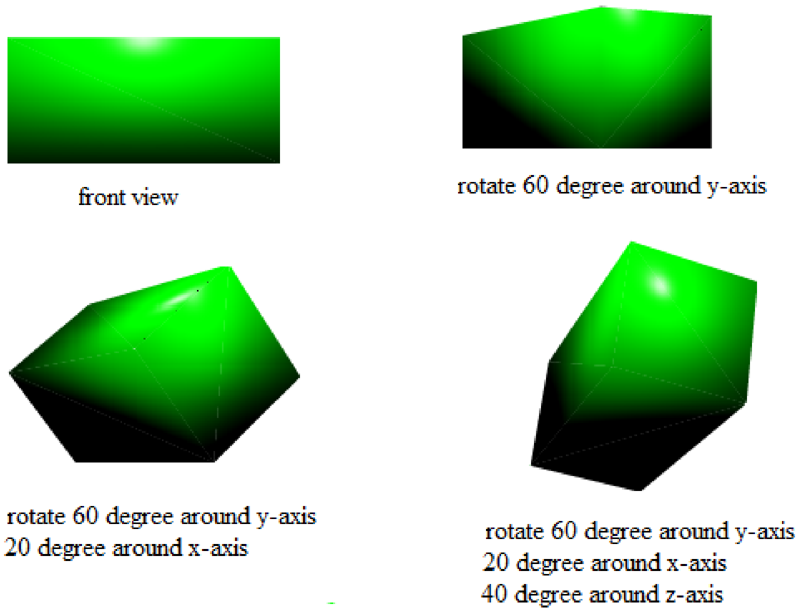
1. It can be used to check whether the Phong shading algorithm proposed in Chapter 6 can generate smooth colour transitions in hard boundary.
2. It can be used to check whether the highlight can be rendered correctly on a big flat surface.

During the test, the average number of second per frame will be recorded, and the file size of the SVG file generated by S3GL will also be recorded for further analysis.

Figure 8.9(a) shows a green box viewed from different angle, and the box is rendered by Flat shading; Figure 8.9(b) shows a box rendered by the Phong shading algorithm proposed in Chapter 6. The point light is defined on the left side of the scene.


front view
rotate 60 degree around y-axis
rotate 60 degree around y-axis
20 degree around x-axis
rotate 60 degree around y-axis
20 degree around x-axis
40 degree around z-axis

(a)


front view
rotate 60 degree around y-axis
rotate 60 degree around y-axis
20 degree around x-axis
rotate 60 degree around y-axis
20 degree around x-axis
40 degree around z-axis

(b)

Figure 8.9 Flat shading vs. Phong shading

The test result shows that there is no highlight by using Flat shading. By using the new Phong shading algorithm, it can generate smooth colour transitions on hard boundary, and the highlight of the object can be generated correctly.

The average number of second per frame which was achieved with this test application on different web browser is shown in Table 8.22. As can be seen in the table, the test application works well on different web browser, and the render time required for Flat shading is far less than area interpolation Phong shading algorithm since there are more arithmetic calculation involved in area interpolation Phong shading algorithm.

Table 8.22 Flat shading and Phong shading render rate

Web browser	IE	Firefox	Chrome	Safari	Opera
Flat shading Render rate (seconds/frame)	0.007	0.0072	0.0067	0.007	0.0069
Phong shading Render rate (seconds/frame)	0.15	0.16	0.22	0.17	0.19

The maximum amount of memory of Flating shading is 2 KB, and maximum amount of memory of area interpolation Phong shading algorithm is 4KB. This shows the area interpolation Phong shading algorithm will use more memory than Flat shading.

Both those performance tests show the Phong shading algorithm developed in Chapter 6 can work well to create realistic 3D object.

8.7.4 3D Plant

The forth test application is a plant in a flowerpot (Figure 8.10). The purpose of this test application is to validate the algorithm provided in Chapter 5 and Chapter 7, and it will have the functionality outlined below.

1. Build 3D SVG model.
2. Rendering 3D object on web browser.
3. Using texture on specified 3D object.
4. Interactive manipulation.

The reasons of choose this 3D scene are:

1. It can be used to validate whether different 3D objects can be created and rendered correctly.
2. It can be used to validate whether the position relationship between objects can be rendered correctly.
3. It can be used to validate whether the texture mapping algorithm can generate correct texture for different object.

The 3D plant can be viewed from different angle. During the test, the average number of second per frame will be recorded, and the file size of the SVG file generated by S3GL will also be recorded for further analysis.

The flowerpot is created by using revolution object in S3GL; and the leaves of the plant are generated by using Bezier surface. Texture mapping proposed in Chapter 7 are used in this application, include texture mapping algorithms for Bezier surface, texture mapping algorithms for revolution surface, and p texture mapping algorithms for cylinder.

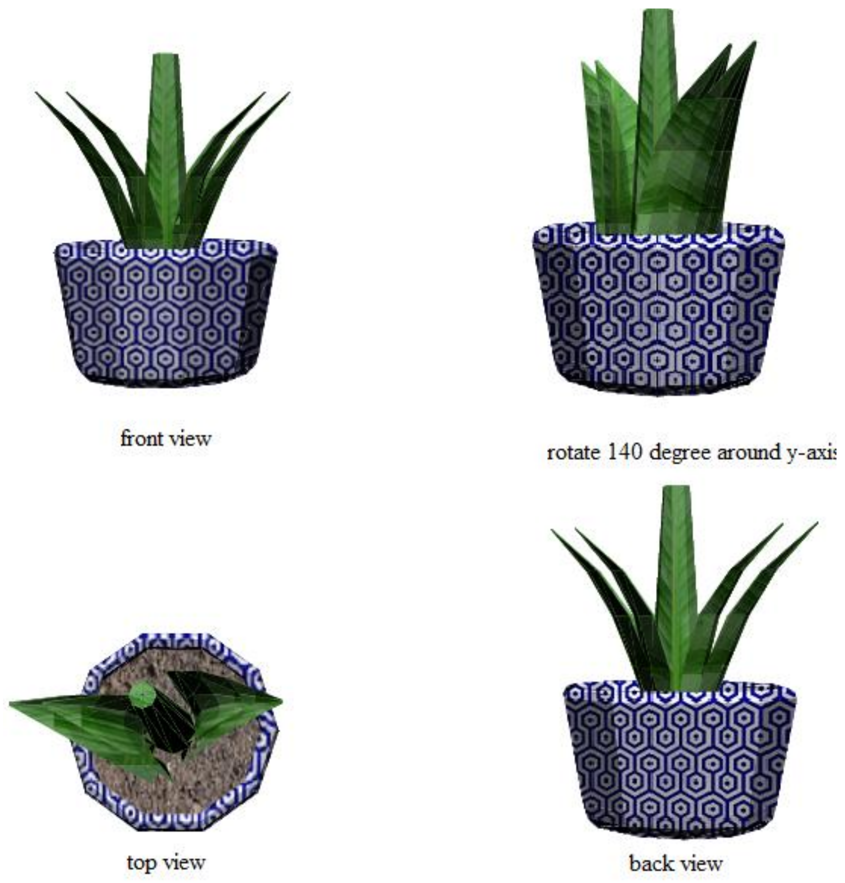


Figure 8.10 Texture mapping 3D plant

The average number of second per frame which was achieved with this test application on different web browser is shown in Table 8.23. As can be seen in the table, the test application works well on different web browser, and the render rate are also similar on different web browser.

Table 8.23 3D plant render rate

Web browser	IE	Firefox	Chrome	Safari	Opera
Render rate (seconds/frame)	0.62	0.58	0.63	0.61	0.62

The maximum amount of memory of this application is 66 KB.

Figure 8.11 shows a 3D paint pot model. Campare with the 3D plant, this is a relative simple 3D model. But the file size for such a 3D model created by X3D is 157 KB; and the file size created by VRML is 142 KB. Compare with the 3D plant model created by the SVG GL, the file size is just about 1/2 of the file size created by X3D or VRML.

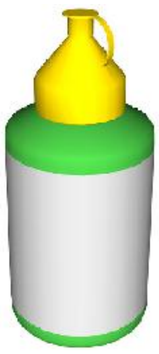


Figure 8.11 3D paint pot model (free model download from http://www.3dcadbrowser.com)

Both those performance tests show the algorithm proposed in Chapter 5 can be used to generate 3D scene. And the texture mapping described in Chapter 7 can also work well to create realistic 3D object. By compare the final file size with X3D and VRML, it shows S3GLcan significantly reduce the file size, which is important for web-based applcations.

8.8 Summary

A test software system-S3GL is developed by using Visual Studio 2010. The system is developed based on the theory proposed in this project. The purpose of S3GL is to validate the framework-the SVG GL proposed in Chapter 5; the new Gouraud shading and Phong shading algorithms developed in Chapter 6; and the new texture mapping algorithm developed in Chapter 7.

Two Use case are developed firstly to validate the test environment. Then 4 test applications are implementing to validate the algorithm proposed in this project:

1. 3D static objects, to validate the algorithms provided in Chapter 5 and Chapter 7.
2. 3D Gouraud shading box, to validate the Gouraud shading algorithm proposed Chapter 6.
3. 3D Phong shading box, to validate the Phong shading algorithm proposed Chapter 6.
4. 3D plant, validate the algorithms provided in Chapter 5 and Chapter 7.

The testing shows satisfactory results in representing different graphics content. It proves the algorithms proposed in Chapter 5 can be used to generate 3D scene. And the new Gouraud shading algorithm and Phong shading algorithm developed in Chapter 6, the texture mapping algorithms proposed in Chapter 7 can work well to create realistic 3D object. By compare the final file size with X3D and VRML, it shows S3GL can significantly reduce the file size, which is important for web-based applcations.

Chapter 9 The Discussions of the Proposed Methods for 3D Web-Based Presentations

9.1 Introduction

In this chapter, 4 demo applications based on S3GL will be presented:

1. A 3D bottle, to test the potential application of the new framework for product presentation.

2. Building site simulation, to test the potential application of the new framework for city planning and community management.

3. Shopping mall, to test the potential application of the new framework for e-business.

4. 3D landscape, to test the potential application of the new framework for 3D terrain simulation.

Those applications are used to evaluate the suggested methods as proof of concept, and also investigate the potential application fields of S3GL. Those applications were tested on the same environment as the 4 test applications in Chapter 8. During the demo the average number of second per frame will be recorded, and the file size of the SVG file generated by S3GL will also be recorded for further analysis.

9.2 3D Bottle

The first demo application is 3D bottle (Figure 9.1). The purpose of this application is to test the potential application of the new framework for product presentation, and it has the functionalities:

1. Build 3D SVG model.

2. Rendering 3D object on web browser.

3. Using texture on specified 3D object.

4. Interactive manipulation.

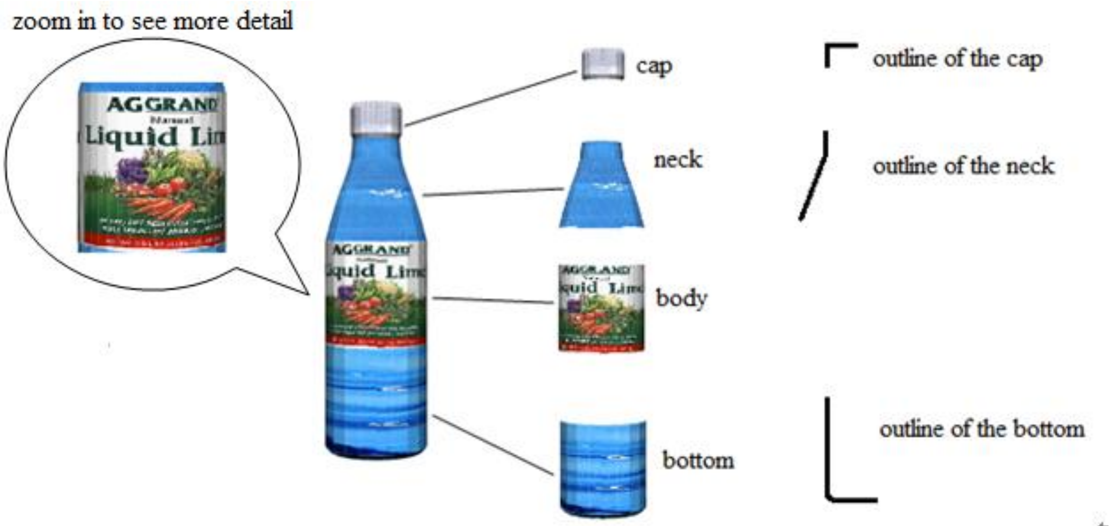


Figure 9.1 3D bottle

The geometry of the bottle is created by using the 3D objects provided in the S3GL. The cap of the bottle is created by the revolution object in S3GL, the outline of the cap is shown on the right side, and the rotate axis is *y*-axis; the neck of the bottle is created by revolution object as well, and the outline is shown on the right side, the rotate axis is *y*-axis; the body of the bottle is created by using cylinder in S3GL; finally, the bottom of the bottle is created revolution object. Two kids of texture mapping algorithms proposed in Chapter 7 are used; they are texture mapping algorithms for revolution object, and texture mapping algorithms for cylinder. The user can navigate the 3D scene and rotate the camera by using the mouse (Figure 9.2). By zooming in the camera, more details of the texture can be viewed.

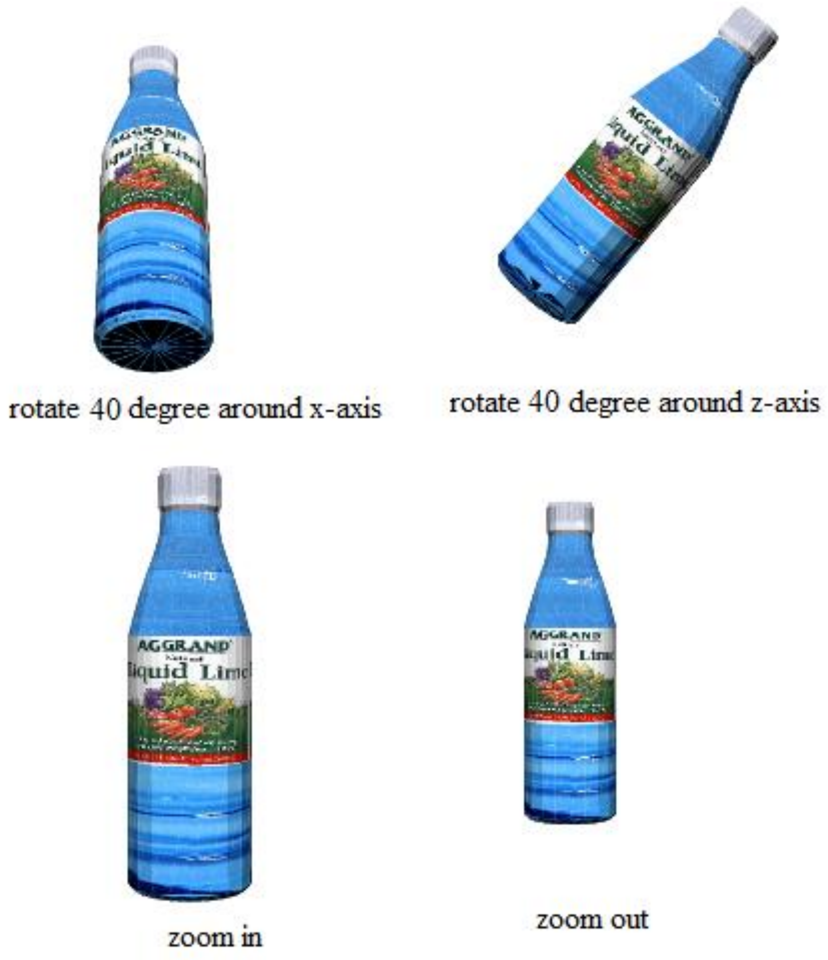

rotate 40 degree around x-axis
rotate 40 degree around z-axis
zoom in
zoom out

Figure 9.2 Rotate the camera to navigate the 3D bottle

The average number of second per frame which was achieved with this application on different web browser is shown in Table 9.1. As can be seen in the table, the test application works well on different web browser, and the render times are also similar on different web browser. The maximum amount of memory of this application is 36KB.

Table 9.1 3D bottle render rate

Web browser	IE	Firefox	Chrome	Safari	Opera
Render rate (seconds/ frame)	0.50	0.52	0.53	0.51	0.49

Figure 9.3 shows a 3D coke can, the file size for such a 3D model created by X3D is 162 KB; and the file size created by VRML is 148 KB. Compare with

the 3D bottle model created by S3GL, the file size is only about 1/5 of the file size created by X3D or VRML.

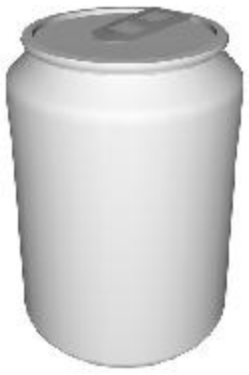


Figure 9.3 A 3D coke can (free model download from http://www.3dcadbrowser.com)

The results show S3GL can be used for simulating 3D houseware, such as bottle, can, flask, and similar product with realistic texture. It can be used for product presetation for web-based application, and can significantly reduce the file size of the 3D model.

9.3 Building Site Simulation

The second demo application is building simulation (Figure 9.4). The purpose of this application is to test the potential application of the new framework for city planning and community management, and it will have the functionality outlined below.

1. Build 3D SVG model.
2. Rendering 3D object on web browser.
3. Using texture on specified 3D object.
4. Interactive manipulation.

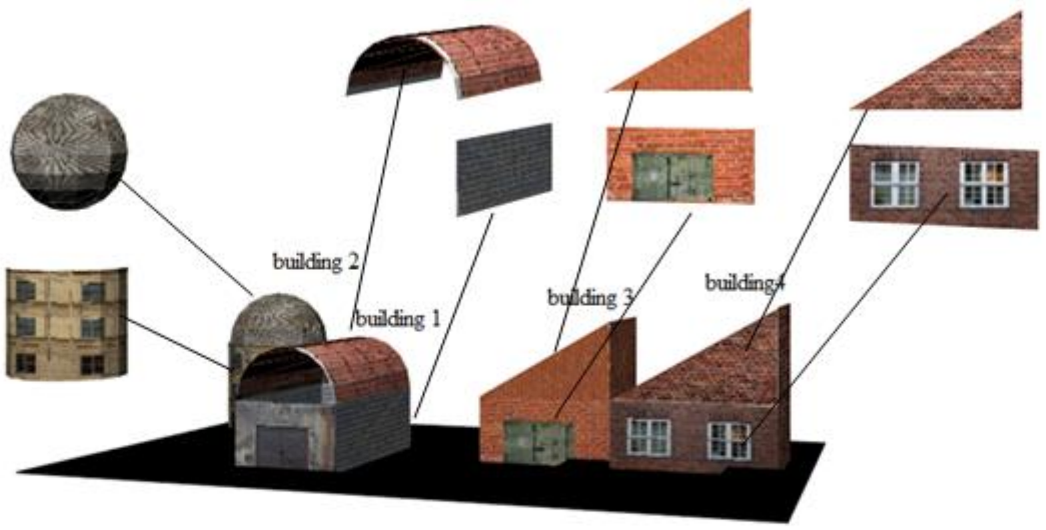
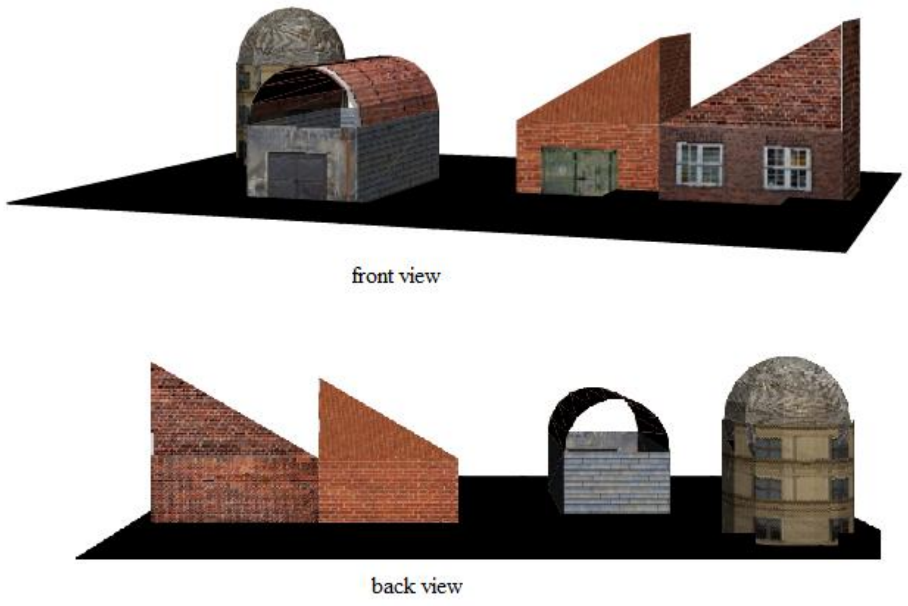


Figure 9.4 Building site simulation

The geometry of the building site is created by using the 3D objects provided in the S3GL. Building 1 consist of one extrusion object and four planes, the extrusion is used as the roof, and the planes are used as wall; building 2 consist of a sphere and a cylinder, the sphere is used as roof, and the cylinder is used as wall; building 3 consist of 2 triangle and 5 planes; and building consist of 2 triangle and 4 planes. Four kids of texture mapping algorithms proposed in Chapter 7 are used; they are texture mapping algorithms for triangle, texture mapping algorithms for plane, texture mapping algorithms for sphere, and texture mapping algorithms for cylinder. The user can navigate the 3D scene and rotate the camera by using the mouse (Figure 9.5).



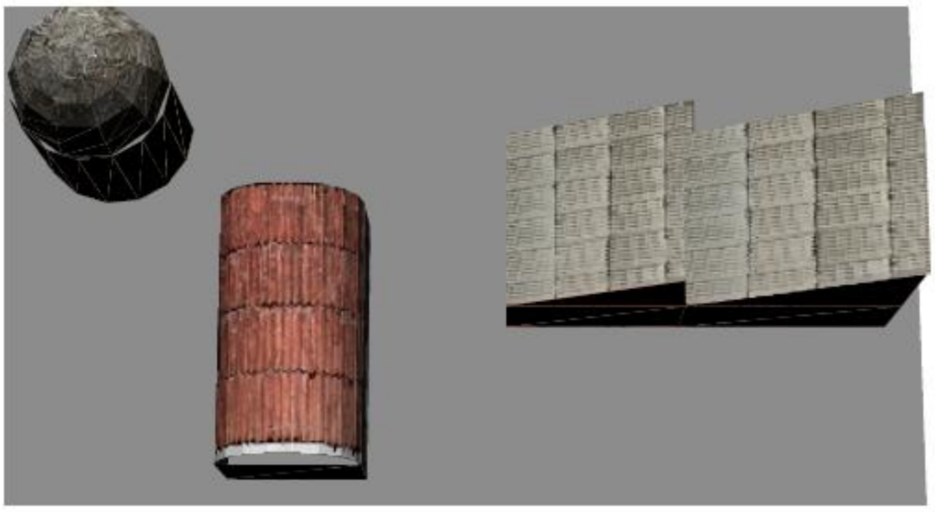


Figure 9.5 Rotate the camera to navigate the building site

The average number of second per frame which was achieved with this application on different web browser is shown in Table 9.2. As can be seen in the table, the test application works well on different web browser, and the render times are also similar on different web browser. The maximum amount of memory of this application is 48KB.

Table 9.2 Building simulation render rate

Web browser	IE	Firefox	Chrome	safari	Opera
Render rate (seconds/frame)	0.87	0.89	0.87	0.92	0.85

Figure 9.6 shows a 3D case wall model, the file size for such a 3D model created by X3D is 431 KB; and the file size created by VRML is 390 KB. Compare with the 3D building site model created by the SVG GL, the file size is only about 1/8 of the file size created by X3D or VRML.

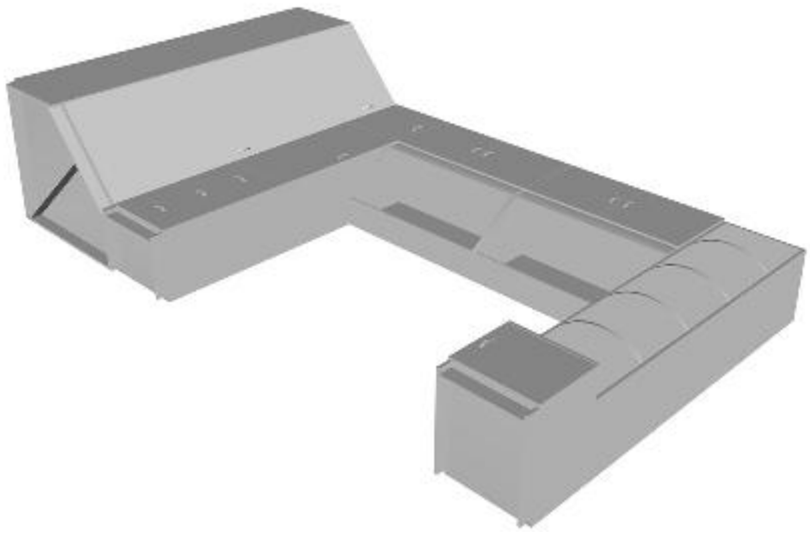


Figure 9.6 A 3D case wall model (free model download from http://www.3dcadbrowser.com)

The results show that S3GL can be used for simulating 3D architecture, such as house, barn, tower, and similar building with realistic texture. It can be used for city planning and community management, and can significantly reduce the file size of the 3D model.

9.4 Shopping Mall

The third demo application consists of 2 scenarios, the first is the buildings of a supermarket, and the second is the inside of the supermarket (Figure 9.7, Figure 9.8). The purpose of this application is to test the potential application of the new framework for e-business, and it will have the functionality outlined below.

1. Build 3D SVG model.
2. Rendering 3D object on web browser.
3. Using texture on specified 3D object.
4. Interactive manipulation..

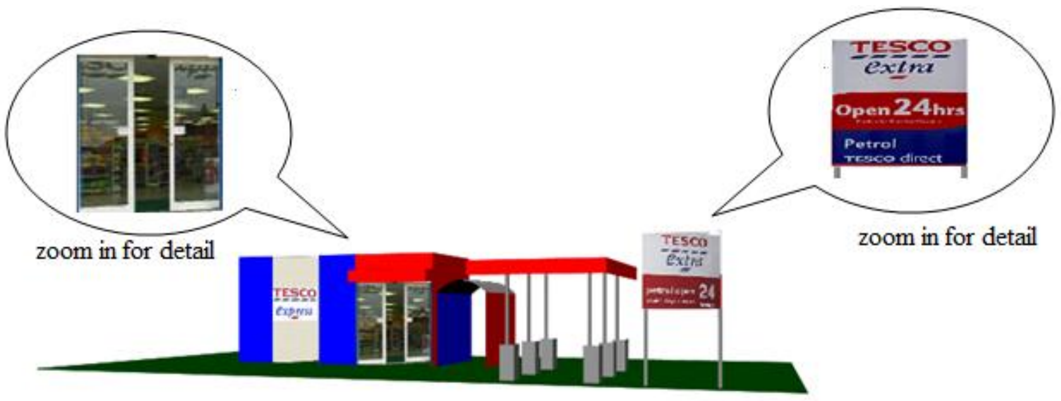


Figure 9.7 Supermarket

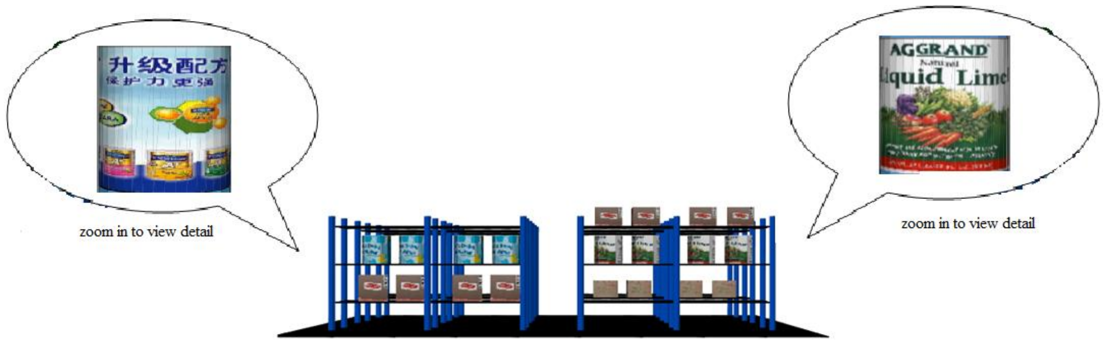


Figure 9.8 The inner scenario of the supermarket

The geometry of the supermarket is created by using the 3D objects provided in the S3GL, including plane, cylinder and Bezier surface. Three kids of texture mapping algorithms proposed in Chapter 7 are used; they are texture mapping algorithms for plane, texture mapping algorithms for cylinder, and texture mapping algorithms for Bezier surface. The user can navigate the 3D scene and rotate the camera by using the mouse (Figure 9.9).

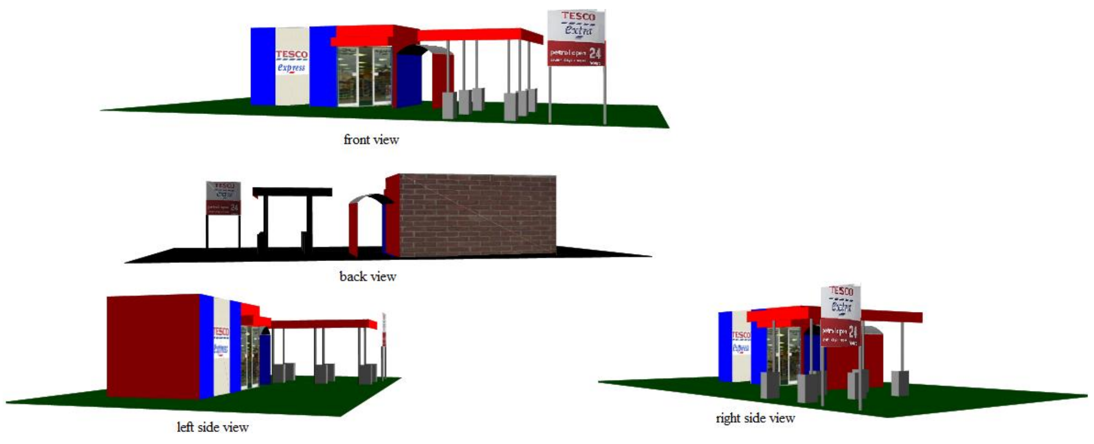


Figure 9.9 Rotate the camera to navigate the supermarket

The average number of second per frame which was achieved with this application on different web browser is shown in Table 9.3. As can be seen in the table, the test application works well on different web browser, and the render times are also similar on different web browser. The maximum amount of memory of this application for the supermarket is 44.5KB, and for the inner scenario is 477KB.

Table 9.3 The supermarket render rate

Web browser	IE	Firefox	Chrome	safari	Opera
Outside render rate (seconds /frame)	0.83	0.79	0.83	0.77	0.80
Inside render rate (seconds /frame)	0.72	0.73	0.73	0.85	0.72

Figure 9.10 shows a 3D building model, the file size for such a 3D model created by X3D is 1,864 KB; and the file size created by VRML is 1,682 KB. Compare with the 3D supermarket model created by the SVG GL, the file size is only about thirtieth of the file size created by X3D or VRML.

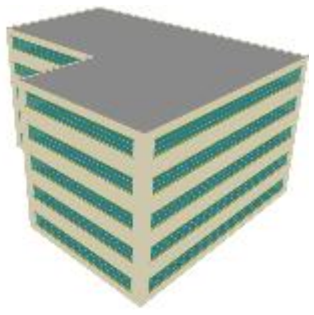


Figure 9.10 A 3D building model (free model download from http://www.3dcadbrowser.com)

The results show that S3GL can be used for simulating 3D supermarket, and the inner scenario of the market with realistic texture. It can be used for web-based e-business, and can significantly reduce the file size of the 3D model.

9.5 3D Landscape

The final demo application is a 3D landscape (Figure 9.11). The purpose of this application is to test the potential application of the new framework for 3D terrain simulation, and it will have the functionality outlined below.

1. Build 3D SVG model.
2. Rendering 3D object on web browser.
3. Using texture on specified 3D object.
4. Interactive manipulation.

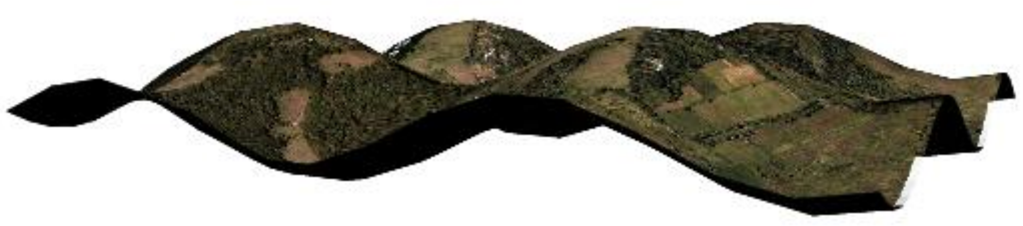


Figure 9.11 3D landscape

The geometry of the landscape is created by using the 3D objects provided in the S3GL, mainly extrusion object. The texture mapping algorithms for extrusion object proposed in Chapter 7 is used. The user can navigate the 3D scene and rotate the camera by using the mouse (Figure 9.12).

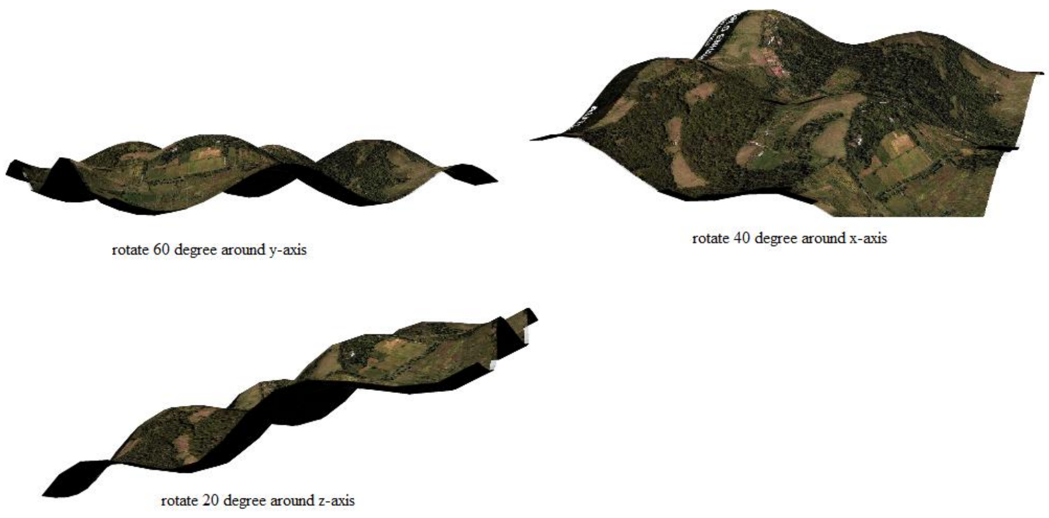

rotate 60 degree around y-axis
rotate 40 degree around x-axis
rotate 20 degree around z-axis

Figure 9.12 Rotate the camera to navigate the landscape

The average number of second per frame which was achieved with this application on different web browser is shown in Table 9.4. As can be seen in the table, the test application works well on different web browser, and the render times are also similar on different web browser. The maximum amount of memory of this application is 120KB.

Table 9.4 3D landscape render rate

Web browser	IE	Firefox	Chrome	safari	Opera
Render rate (frame/seconds)	0.95	0.96	0.99	0.95	0.94

Figure 9.13 shows a 3D landscape, the file size for such a 3D model created by X3D is 32,328KB; and the file size created by VRML is 29,247 KB. Compare with the 3D supermarket model created by the SVG GL, the file size is only about 1/200 of the file size created by X3D or VRML.

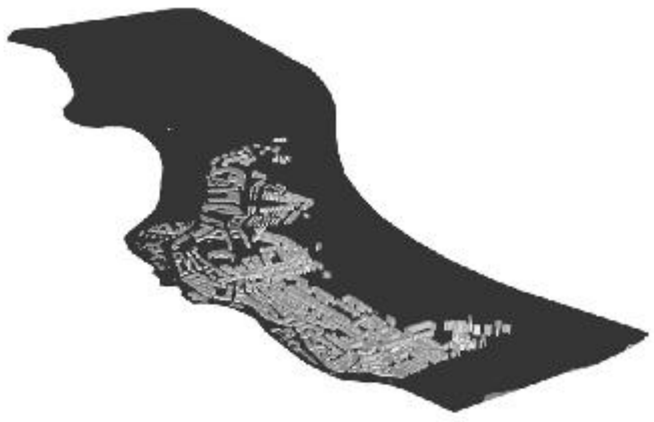


Figure 9.13 3D model of Ciudad Del Puerto (free model download from http://www.3dcadbrowser.com)

The results show that S3GL can be used for simulating 3D landsceape with realistic texture. It can be used for terrain simulation for web-based application, and can significantly reduce the file size of the 3D model.

9.6 Evaluation of the Applications of S3GL Presentations

Four demo applications are developed in this section.

1. A 3D bottle model.
2. A 3D building site.
3. A supermarket.
4. A 3D landscape.

Those applications are used to investigate the potential application fields of S3GL. By successfully running those demo application, it shows the S3GL can be used for product demonstration, urban environment simulation, city planning; for warehouse demonstration and 3D terrain simulation.

Once again, those demo application also validate the new framework proposed in Chapter 5, the texture mapping algorithm developed in Chapter 7.

Four similar 3D models are also developed by using X3D and VRML. By compare the file size created by X3D and VRML with the respective file size created by S3GL, it shows S3GL can significantly reduce the file size which will be benefit for web-based application.

All these make the S3GL an ideal tool for creating 3D model for web-based applications, and achieving real time(less than 1 second) interactive with the 3D model. Currently, all calculations involving model transformation, texture processing are carried out by software, this significantly reduce the performance of the S3GL. If some calculation expensive operation can be implemented by hardware, the performance of S3GL can be significantly increased (the average rendering time can be reduced to 0.1 second), and even can be used for web-based 3D animation.

Chapter 10 Conclusions and Further Work

This chapter concludes this thesis with a summary of what has been done, in the context of the research goals stated in the introduction, as well as with an outline of the original contribution to the body of knowledge, and a discussion of future work.

10.1 Summary

This thesis is a summary of an original research work, which is unique in several regards.

1. It is a unique and original synthesis of the subjects related to web based 3D model, which is based on an analysis of a broad range of theoretical and practical resources from traditional and modern 3D models, as well as from the related disciplines such as SVG.
2. The discipline of the SVG GL is defined through both a technological description, and by the proposed unique set of defining factors –differentiators, which are: the use of 3D modelling; the use of SVG rules for the presentation efficiency maximization; high interactivity – which are also thoroughly discussed.
3. Numerous theoretical and functional, as well as practical and technological aspects of web based 3D model are described in depth.
4. The theoretical assumptions and rules are tested in practice, and validated through the practical process of development of a S3GL.
5. The developed S3GL is used in real-life applications, to demonstrate that efficient and effective 3D models may be developed today, without the need to wait for any further advances in computer technologies, or in data availability.
6. Four demo applications of 3D models are presented, with a discussion of how and where they can be of benefit.

10.2 Main Achievements

This work provides a broad summary of knowledge relating to the subject of web based 3D model, including their different theoretical, functional, technological and practical aspects. In the process of its completion all of the research objectives stated in the introductory chapter were successfully fulfilled:

1. Evaluation and analysis of web based 3D models: the state-of-the-art of the web based 3D models have been evaluated and analyzed. The importance of web-based 3D model has been discussed. Further research requirements have been identified and clearly stated.

2. Proposition, design and development of the new framework-SVG GL for web-based applications: the SVG GL has been defined through a technological description, and through a unique proposed set of defining factors, which include: the use of 3D visualization, the employment of SVG rules, high interactivity.

3. Core development and practical validation: a test software system-S3GL has been developed, enabling the validation of the discussed the SVG GL principles, and of the proposed defining factors. This work includes the design and development of various new SVG 3D models of primitives, free form surfaces, new algorithms for SVG 3D model manipulations including transformation and projection, new algorithms for SVG 3D model enhancement including shading and texture mapping.

4. S3GL functional description: practical knowledge has been gathered, and combined with the practical know-how of S3GL development, in order to discuss functional aspects and recommendations that maximize usability of the SVG GL products

5. Application development: 4 specialized 3D web-based applications, based on the developed S3GL, have been developed and validated.

6. Practical demonstration: the developed S3GL has been used to demonstrate the possibility of successful development of web-based 3D models, without the need to wait for further technological or scientific progress.

7. Identification of applications: the usability of S3GL in different application areas has been further demonstrated by the identification and discussion of a wide range of potential applications.

10.3 The Contributions to New Knowledge Generations

The predicted contributions to the new knowledge body presented in Section 1.4 have been achieved. The research work outlined in this thesis contributes to the body of scientific knowledge in a number of aspects, on two levels: theoretical and practical. A complete framework for web based SVG 3D modelling is presented, particularly focused on developing accurate and flexible algorithms and 3D modelling. Experimental results presented in this thesis show the efficiency and accuracy of the 3D models. A summary of the respective contributions is presented below:

The primary contribution of this project is the proposition, design and development of a new generic framework for modelling and constructing SVG-based 3D models for efficient web-based applications. This framework can be applied widely in interactive manipulation web-based environments.

The main contributions of this PhD project are:

1. Proposition, design and development of a new framework for SVG 3D modelling based on classical 3D graphic theory and SVG. While the model is initialized using classical 3D graphics, the scene model is extended using SVG. A new algorithm to present 3D graphics with SVG has been proposed. The framework including (1) the definition of a 3D scene in the framework, (2) the integration of 3D objects, camera, transformation, projection, light model and texture in a 3D scene, (3) the rendering 3D objects on the web page and (4) enabling the end-user to interactively manipulate objects on the web page.

2. Design and development of a new 3D graphics library for 3D geometric transformation and projection in SVG 3D.

3. Design and development of a set of primitives in SVG 3D, include triangle, sphere, cylinder, cone, etc.

4. Proposition, design and development of the new area interpolation Goraud shading algorithm and area interpolation Phong Shading algorithm to implement Gouraud shading and Phong shading in SVG 3D. The algorithms can be used to generate smooth shading and create highlight for 3D objects.

5. Proposition, design and development of the new texture mapping algorithms-pattern based image transformed texture mapping algorithm for SVG 3D oriented toward web-based 3D modelling applications. Texture mapping algorithms for different 3D objects such as plane, sphere, cylinder, cone, etc. will also be proposed, designed and developed.

This constitutes a unique and significant contribution to the disciplines of web based 3D modelling, as well as to the process of 3D model popularization.

10.4 Further Work

Due to the time and resource limitation, a number of areas in this PhD project have been identified for further improvement. The proposed future work encompasses four distinguished themes: 1) improvements of the SVG GL; 2) new techniques SVG 3D solid modelling; 3) the development of new applications; and 4) the continued popularization of the SVG GL.

The first theme concerns functional and technical improvements of the SVG GL. These include: the introduction of new data formats, the broadening of the range of the representational forms available, the strengthening of the 3D model handling mechanisms, the further development of the drawing optimization mechanisms, as well as other improvements that will be identified as desirable in the future.

The second theme is related to the study the new technology for SVG 3D solid modelling. In this project, the technologies have been developed mainly for SVG 3D surface modelling. However, the SVG 3D models should also be very useful for product production purposes

The third theme is concerned with development and – where possible practical deployment, of a broader range of applications of the SVG GL. This includes the practical development of the already-identified applications, as well as the identification, analysis and the potential subsequent implementation of the completely new application ideas.

The last theme are related to the intended continuation of efforts in the popularization of the SVG GL, which may be done both through practical demonstrations of the newly developed applications, as well as through publication of research results in the future.

Reference

Alkalay, A. (2007) *Some SVG Games*. [cited 10th October 2013] Available from http://avi.alkalay.net/2007/08/svg-games.html.

Angel, E. (2003) *Interactive Computer Graphics: A Top-down Approach Using OpenGL*. Boston, USA: Addison-Wesley.

Badros, G., Tirtowidjojo, J. and Marriott, M. (2001) A Constraint Extension to Scalable Vector Graphics, *Proceedings of the 10th International Conference on World Wide Web*, pp. 489-498. Hong Kong, May 2001.

Baravalle, A., Gribaudo, M. and Lanfranc, V. (2003) Using SVG and XSLT for Graphic Representation, *Scalable Vector Graphics (SVG) Open 2003 Conference*. Vancouver, Canada, July 2003.

Baru, C., Behere, A. and Cowart, C. (2001) Representation and Display of Geospatial Information: a Comparison of ArcXML and SVG, *Proceedings of the Second International Conference on Web Information Systems Engineering 2001*, pp. 48-53. Kyoto, Japan, December 2001.

Belongie, S., Malik, J. and Puzicha, J. (2002) Shape Matching and Object Recognition Using Shape Contexts, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(4), pp. 509 – 522.

Bertoline, G. (1998) Visual Science: An Emerging Discipline1, *Journal for Geometry and Graphics*, 2(2), pp.181-187.

Bishop, G., Weimer, M. (1986) Fast Phong Shading, *Newsletter ACM SIGGRAPH Computer Graphics*, 20(4), pp. 103-106.

Blinn, J. F. (1977) Models of Light Reflection for Computer Synthesized Pictures, *Proceedings of the 4th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 192-198. California, USA, July 1977.

Bliss, F., Dill, J. and Machover, C. (2002) Graphics in Advanced Computer-Aided Design, *IEEE Computer Graphics and Applications*, 22(3), pp. 22-23.

Blundell, B. (2008) *An Introduction to Computer Graphics and Creative 3-D Environments*. Berlin, Germany: Springer.

Boivin, S., Gagalowicz, A. (2001) Image-Based Rendering of Diffuse, Specular and Glossy Surfaces From a Single Image, *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 107-116. Los Angeles, USA , August 2001.

Brodlie, K. W., Wood, J. and Duce, D. A. (2002) XML for Visualization, *Proceedings of the 2002 International Conference on EuroWeb*, pp. 10-10. Oxford, UK, December 2002.

Brutzman, D., Daly, L. (2007) *X3D: Extensible 3D Graphics for Web Authors*. Massachusetts, USA: Morgan Kaufmann.

Buss, R. (2003) *3D Computer Graphics: A Mathematical Introduction with OpenGL*. Cambridge, UK: Cambridge University Press.

Caniuse (2013) *Compatibility Table for Support of SVG in Desktop and Mobile Browsers*. [cited 20th January 2014] Available from http://caniuse.com/svg.

Carey, R., Greenberg, D. (1985) Textures for Realistic Image Synthesis, *Computer and Graphics*, 9(3), pp. 125-138.

Carlbom, J., Paciorek, J. (1978) Planar Geometric Projections and Viewing Transformations, *Journal for ACM Computing Surveys*, 10(4), pp. 465-502.

Chang, H., Raffensperger, J. and Churcher, N. (2004) Displaying Linear Programs and Their Solutions with XML and SVG, *Proceedings of the 2004 Australasian Symposium on Information Visualisation*, pp. 141-150. Christchurch, New Zealand, January 2004.

Chang, Y. H., Chuang, T. (2002) Online Aggregation and Visualization of Census Data: Population Mapping with SVG, XML, and Free Software, *Proceedings of SVG Open 2002*. Zurich, Switzerland, July 2002.

Chen, B., Dachille, F. and Kaufman, A. (1999) Forward Image Mapping, *Proceedings of the Conference on Visualization '99*, pp 89-96. San Francisco, USA, October 1999.

Chen, R., Liu, L. and Dong, G. (2010) Local Resampling for Patch-Based Texture Synthesis in Vector Fields, *International Journal of Computer Applications in Technology*, 38(1/2/3), pp.124-133.

Comninos, P. (2005) *Mathematical and Computer Programming Techniques for Computer Graphics*. Berlin, Germany: Springer.

Cook, C., Torrance, K. (1982) A Reflectance Model for Computer Graphics, *Journal of ACM Transactions on Graphics*, 1(1), pp. 7-24.

Coxeter, H. S. M. (1974) *Projective Geometry (second edition)*. Berlin, Germany: Springer.

Criag A., James N. and Stuart M. (2008) Web Software Visualization Using Extensible 3D (X3D) Graphics, *Proceedings of the ACM 2008 Symposium on Software Visualization*, pp. 213-214. Ammersee, Germany, September 2008.

Dahlström, E., Dengler, P. and Grasso, A. (2011) *Scalable Vector Graphics (SVG) 1.1 (Second Edition)*. [cited 20th October 2012] Available from http://www.w3.org/TR/SVG11/Overview.html.

Dailey, D., Frost, J. and Strazzullo, D. (2012) *Building Web Applications with SVG*. USA: Microsoft Press.

Danchilla, B. (2012) *Beginning WebGL for HTML5*. New York, USA: Apress.

Delahunty, B. (2003) *Introduction to Mono - ASP.NET with XSP and Apache*. [cited 21st January 2014] Available from http://www.codeproject.com/Articles/9738/Introduction-to-Mono-ASP-NET-with-XSP-and-Apache.

Delmarcelle, T., Hesselink, L. (1993) Visualizing Second-Order Tensor Fields with Hyper Streamlines, *Computer Graphics and Applications, IEEE*, 13(4), pp. 25-33.

Deng, K. et al. (2002) Texture Mapping with a Jacobian-Based Spatially-Variant Filter, *Proceedings of the 10th Pacific Conference on Computer Graphics and Applications*, pp. 460-461. Washington, USA, October 2002.

Dischler, J. M., Ghazanfarpour, D. (2001) A Survey of 3D Texturing, *Computers & Graphics*, 25(1), pp. 135–151.

Eggert, D.W.. Lorusso, A. and Fisher, R.B. (1997) Estimating 3-D Rigid Body Transformations: a Comparison of Four Major Algorithms, *Machine Vision and Applications*, 9(5-6), pp 272-290.

Eisenberg, J. (2002) S*VG Essentials*. Sebastopol, USA: O'Reilly Media.

Elinas, P., Stürzlinger, W. (2000) Real-Time Rendering of 3D Clouds, *Journal of Graphics Tools*, 5(4), pp. 33-45.

El-Khalili, E. (2005) 3D Web-Based Anatomy Computer-Aided Learning Tools, *The International Arab Journal of Information Technology*, 2(3), pp. 248-252.

Farin, G. (1996) *Curves and Surfaces for Computer-Aided Geometric Design: A Practical Code*. Orlando, USA: Academic Press.

Foley, J. et al. (2013) *Computer Graphics Principles and Practice(3rd edition)*. Boston , USA: Addison-Wesley.

Foskey, M., Otaduy, A. and Lin, M. C. (2002) ArtNova: Touch-Enabled 3D Model Design. *IEEE Virtual Reality Conference 2002*, pp. 119-119. Orlando, Florida, USA, March 2002.

Funt, B., Drew, M. and Brockington, M. (1992) Recovering Shading From Colour Images, *Proceedings of the 2nd European Conference on Computer Vision*, pp. 124-132. Santa Margherita Ligure, Italy, May 1992.

Gaedke, M., Turowski, K. (2000) Integrating Web-based E-Commerce Applications with Business Application Systems, *Netnomics*, 2(2), pp 117-138.

Gálve , A., Iglesias, A. and Cobo, A.(2007) Bézier Curve and Surface Fitting of 3D Point Clouds Through Genetic Algorithms, Functional Networks and Least-Squares Approximation, *Proceedings of the 2007 International Conference on Computational Science and Its Applications*, pp.680-693. Kuala Lumpur, Malaysia, August 2007.

Gobithasan, R., Jamaludin, M.A. (2005) Using Mathematic & MATLAB for CAGD/CAD Research and Education. *The 2nd International Conference on Research and Education in Mathematics (ICREM 2)-2005*, pp. 518-525. University Putra, Malaysia, May 2005.

Golovinskiy, A., Kim, V. and Funkhouser, T. (2009) Shape-Based Recognition of 3D Point Clouds in Urban Environments, I*nternational Conference on Computer Vision 2009*. Kyoto, Japan, September 2009.

Gouraud, H. (1971) Continuous Shading of Curved Surfaces, *IEEE Transactions on Computers*, 20(6), pp. 623 – 629.

Grissom, S. et al. (1995) Using Visual Demonstrations to Teach Computer Science, *Proceedings of the 26th SIGCSE Technical Symposium on Computer Science Education*, pp. 370-371. Nashville, USA, March 1995.

Groover, M., Zimmers, E. (1983) *CAD/CAM: Computer-Aided Design and Manufacturing*. New Jersey, USA: Prentice Hall.

Gross, H., Thoennessen, U. and Hansen, W. v. (2005) 3D Modelling Of Urban Structures, *Object Extraction for 3D City Models, Road Databases and Traffic Monitoring - Concepts, Algorithms, and Evaluation (CMRT) 2005*, pp. 137-142. Vienna, Austria, August 2005.

Haeberli, P., Segal, M. (1993) Texture Mapping as Fundamental Drawing Primitive, *Proceedings of 4th Eurographics Workshop Rendering*. Paris, France, June 1993.

Harris, M. et al. (2003) Simulation of Cloud Dynamics on Graphics Hardware, *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pp. 92-101. San Diego, USA, July 2003.

Heckbert, P. (1986) Survey of Texture Mapping, *Journal of IEEE Computer Graphics and Applications*, 6(11), pp. 56-67.

Heeger, D., Bergen, J. (1995) Pyramid-Based Texture Analysis/Synthesis, *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, pp. 229-238. Los Angeles, USA, August 1995.

Hees, H. (2006) *3D Computer Graphics*. Florida, USA: InstaBook Corporation.

Hibbard, B. (1998) VisAD: Connecting People to Computations and People to People, *IEEE Computer Graphics*, 3(32), pp. 10-12.

Huang, H. et al. (2011) An SVG-Based Method to Support Spatial Analysis in XML/GML/SVG-Based WebGIS, *International Journal of Geographical Information Science*, 25(10), pp.1561-1574.

Jimenez, J., Cruz, J. (2013) High Performance 3D Visualization on the Web: a Biomedical Case Study, *IWBBIO13*, 15(3), pp. 465-471.

Jones, W. (2004) *Beginning Directx 9 (Game Development Series)*. Newcastle, UK: Premier Press.

Kevin, L. (2003) *KevLinDev*. [cited 2nd May 2015] Available from http://www.kevlindev.com/games/index.htm.

Kennedy, M., Kopp, S. (2001) *Understanding Map Projections*. California, USA: Esri Press.

Kent, J., Carlson, W. and Parent, R. (1992) Shape Transformation for Polyhedral Objects, *Computer Graphics*, 26(2), pp.47-54.

Kobayashi, A., Takagi, S. and Inoue, N. (2003) Extensions of SVG for Human Navigation by Cellular Phone, *Proceeding SIGGRAPH '03 ACM SIGGRAPH 2003 Web Graphics*, pp. 1-1. San Diego, CA, USA, July 2003.

Kolbe, T. H., Groger, G. and Plumer, L. (2005) CityGML – Interoperable Access to 3D City Models, *Proceeding of the 1st International Symposium on Geo-information for Disaster Management*, pp. 883-899. Delft, Netherlands, March 2005.

Krysl, P., Ortiz, M. (2001) Extraction of Boundary Representation From Surface Triangulations, *International Journal for Numerical Methods in Engineering*, 50(7), pp. 1737-1758.

Kumar, P. et al. (2008) Grasping Molecular Structures Through Publication-Integrated 3D Models, *Trends in Biochemical Sciences*, 33(9), pp. 408-412.

Lee, S. et al. (2002) Ubiquitous Access for Collaborative Information System Using SVG, *Proceedings of SVG Open 2002*, pp. 34-41, Zurich, Switzerland, July 2002.

Lefebvre, S., Neyret, F. (2003) Pattern Based Procedural Textures, *Proceedings of the 2003 Symposium on Interactive 3D Graphics*, pp. 203-212. Monterey, USA, April 2003.

Lewis, C. et al. (2002) Development of an SVG GUI for the Visualization of Genome Data, *Proceedings of SVG Open 2002*. Zurich, Switzerland, July 2002.

Lindsey, K. (2003) *3D Geometry*. [cited 10th September 2011] Available from http://www.kevlindev.com/geometry/index.htm.

MacEachren, A. M. (1998) Cartography, GIS, and the World Wide Web, *Progress in Human Geography 1998*, 22(4), pp. 575-589.

Machover, C., Whitted, T. (1998) Computer Graphics In Entertainment, *Computer Graphics and Applications, IEEE*, 18(1), pp. 22-23.

Marescaux, L., Clement, J. M. and Tassetti, V. (1998) Virtual Reality Applied to Hepatic Surgery Simulation: The Next Revolution, *Annals of Surgery*, 228(5), pp. 627-634.

Martin, R. (1989) Sweeping of Three-Dimensional Objects, *Computer-Aided Design*, 22(4), pp. 223–234.

Maynard, P. (2005) *Drawing Distinctions: The Varieties of Graphic Expression*. New York, USA: Cornell University Press.

Michael, L., Eric B. and Gregory P. (1997) *Direct3D Professional Reference*. San Francisco, USA: New Riders Pub.

Mihaela, J (2011) *Inkscape 0.48 Illustrator's Cookbook-GUTL*. Birmingham, UK: Packt Publishing.

Mitra, T., Chiueh, T. (1999) Dynamic 3D Graphics Workload Characterization and the Architectural Implications, *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 62-71. Haifa, Israel, Nov 1999.

Mong, J., Brailsford, D. (2003) Using SVG as the Rendering Model for Structured and Graphically Complex Web Material, *Proceedings of the 2003 ACM Symposium on Document Engineering*, pp. 88-91. Grenoble, France, November 2003.

Nicolae, G., Moldoveanu, F. and Telea, A. (2004) Shading in a Distributed Environment, *Proceedings of 8th International Conference on Information Visualisation*, pp. 1003-1006. London, UK, July 2004.

Oliveira, M., Bishop, G. and Mcallister, D. (2000) Relief Texture Mapping, *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 359-368. New Orleans, USA, July 2000.

Ortiz, S. (2010) Is 3D Finally Ready for the Web?, *IEEE Computer Society*, 43(1), pp. 14-16.

Pamela, R. (2013) *The Konqueror Handbook*. [cited 12th December 2013] Available from https://docs.kde.org/development/en/applications/konqueror/index.html.

Parent, R. (2012) *Computer Animation, Third Edition: Algorithms and Techniques*. San Francisco, USA: Morgan Kaufmann.

Patrick C. (2012) *OpenGL Insights*. Boca Raton, USA: CRC Press.

Peterson, M. (2012) *Online Maps with APIs and WebServices*. Berlin, Germany: Springer.

Peter. (2011) *Rotating 3D SVG Cube*. [cited 8th October 2013] Available from http://www.petercollingridge.co.uk/blog/rotating-3d-svg-cube.

Pharr, M., Humphreys, G. (2004) *Physically Based Rendering: From Theory to Implementation*. Massachusetts, USA: Morgan Kaufmann.

Phong, B. (1975) Illumination for Computer Generated Pictures, *Communications of the ACM*, 18(6), pp. 311-317.

Pietroni, N., et al. (2007) A Survey on Solid Texture Synthesis, *IEEE Computer Graphics and Applications*, 30(4), pp. 74-89.

Probets, S., et al. (2001) Vector Graphics: From PostScript and Flash to SVG, *ACM Symposium on Document Engineering*, pp. 135-143. Atlanta. Georgia, USA, November 2001.

Rahman, A., Younas, A. (2007) *Web-Based Dynamic Visualization of 3D Spatial Data*. [cited 10th November 2012] Available from http://eprints.utm.my/403/2/Alias_Abdul_Rahman,_Adnan_Younas_fksg.pdf.

Riley, K. F., Hobson, M. P. and Bence, S. J. (2006) *Mathematical Methods for Physics and Engineering (3rd edition): A Comprehensive Guide*. Cambridge, UK: Cambridge University Press.

Ritschel, T. et al. (2012) The State of the Art in Interactive Global Illumination, *Journal of Computer Graphics Forum*, 31(1), pp. 160-188.

Rosenbaum, R., Schumann, H. and Tominski, C. (2004) Presenting Large Graphical Contents on Mobile Devices - Performance Issues, *Proceedings of IRMA2004*, pp. 371-374. New Orleans, USA, May 2004.

Salisbury, C. F., Farr, S. and Moore, J. A. (1999) Web-Based Simulation Visualization Using Java3D, *Proceedings of the 31st Conference on Winter Simulation*, pp. 1425-1429. Phoenix, USA, December 1999.

Sayed Y., Satya K. and Dave M. (2010) *Pro Android 2*. New York, USA: Apress.

Schlick, C. (1994) A Survey of Shading and Reflectance Models, *Journal of Computer Graphics Forum*, 13(2), pp. 121-131.

Seulin, R., Merienne, F. and Gorria, P. (2002) Simulation of Specular Surface Imaging Based on Computer Graphics: Application on a Vision Inspection System, *Journal on Applied Signal Processing*, 2002(1), pp. 649-658.

Sellers, G., Wright, R. S. and Haemel, N. (2010) *OpenGL SuperBible: Comprehensive Tutorial and Reference (5th Edition)*. New Jersey, USA: Addison Wesley Professional.

Selman, D. (2002) *Java 3D Programming*. Connecticut, USA: Manning Publications.

Sharon, S. (2013) *The Adobe Illustrator CS6 WOW! Book*. Berkeley, USA: Peachpit Press.

Sheng, Y. et al. (2005) Visualization GML with SVG, *Geoscience and Remote Sensing Symposium*, pp. 3648-3651. Seoul, Korea, July 2005.

Shreiner, D. (2009) *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1*. Boston, USA: Addison-Wesley Professional.

Shuma, S. S. P., Laub, W. S. and Yuen M. M. F. (2001) Solid Reconstruction From Orthographic Views Using 2-Stage Extrusion, *Computer-Aided Design*, 33(1), pp. 91–102.

Skarler, V. (2009) eManaging Ambient Organizations in 3D, *Journal of Theoretical and Applied Electronic Commerce Research*, 4(3) pp. 30-42.

Sons, K. et al. (2010) XML3D: Interactive 3D Graphics for the Web, *Proceedings of the 15th International Conference on Web 3D Technology*, pp. 175-184. Los Angeles, USA, July 2010.

Spanaki, M., Antoniou, B. and Tsoulos, L. (2004) Web Mapping and XML Technologies: A Close Relationship, *In 7th AGILE Conference on Geographic Information Science*, pp. 831-836. Heraklion, Greece, April 2004.

Strauss, P. S. (1990) A Realistic Lighting Model for Computer Animators, *Journal of IEEE Computer Graphics and Applications*, 10(6), pp. 56-64.

Su, X. Y. et al. (2006) Scalable Vector Graphics (SVG) Based Multi-Level Graphics Representation for Engineering Rich-Content Exchange in Mobile Collaboration Computing Environments, *Journal of Computing and Information Science in Engineering*, 6(2), pp. 96-102.

Sumner, R., Thuerey, N. and Gross, M. (2008) The ETH Game Programming Laboratory: a Capstone for Computer Science and Visual Computing, *Proceedings of the 3rd International Conference on Game Development in Computer Science Education*, pp. 46-50. Miami, USA, February 2008.

Tabellion, E., Lamorlette, A. (2004) An Approximate Global Illumination System for Computer Generated Films, *Journal of ACM Transactions on Graphics*, 23(3), pp. 469-476.

Tarini, M., Cignoni, P. and Rocchini, C. (2000) Real Time, Accurate, Multi-Featured Rendering of Bump Mapped Surfaces, *Computer Graphics Forum*, 19(3), pp. 119-130.

Taubin, G. et al. (1998) Geometry Coding and VRML, *Proceedings of IEEE*, 86(6), pp. 1128-1243.

Tautenhahn, L. (2002) *SVG-VML-3D 1.3*. [cited 12th September 2011] Available from http://www.lutanho.net/svgvml3d/.

Tucker, A. (2004) *Computer Science Handbook, Second Edition*. London, UK: Chapman and Hall/CRC.

Turonova, B. (2009) *3D Web Technologies and Their Usability for The Project 3D Mobile Internet, Technical Report, Faculty of Electrical Engineering*. Czech Technical University in Prague.

Vanhatupa, J. (2013) On the Development of Browser Games – Current Technologies and the Future, *International Journal of Computer Information Systems and Industrial Management Applications*, 5(1), pp. 60-68.

Vucinic, D. et al. (2008) Towards Interoperable X3D Models and Web-based Environments for Engineering Optimization Problems, *International Conference on Engineering Optimization*. Rio de Janeiro, Brazil, June 2008.

Walsh, A., Sevenier, M. (2000) *Core Web 3D*. New Jersey, USA: Prentice hall PTR.

Watt, A. H. (1999) *3D Computer Graphics (3rd Edition)*. New Jersey, USA: Addison Wesley Professional.

Wei, L. et al. (2008) Inverse Texture Synthesis, *Proceedings of ACM SIGGRAPH 2008*. Los Angeles, USA, August 2008.

Wolff, L. (1996) Generalizing Lambert's Law for Smooth Surfaces, *Proceedings of the 4th European Conference on Computer Vision*, pp. 40-53. Cambridge, UK, April 1996.

Wolff, L., Nayar, S. and Oren, M. (1998) Improved Diffuse Reflection Models for Computer Vision, *Journal of International Journal of Computer Vision*, 30(1), pp. 57-71.

Wolff, D. (2005) Using OpenGL in Java with JOGL, *Journal of Computing Sciences in Colleges*, 21(1), pp. 223-224.

Wong, U., Wong, H. and Tang, Z. (2005) An Interactive System for Visualizing 3D Human Organ Models, *Proceedings of the 9th International Conference on Computer Aided Design and Computer Graphics*, pp. 403-408. Hong Kong, China, December 2005.

Wong, G., Wang, J (2013) *Real-Time Rendering: Computer Graphics with Control Engineering (Automation and Control Engineering)*. Florida, USA: CRC Press.

Wright, R., Lipchak, R. (2004) *OpenGL Super Bible*. Indianapolis,USA: Sams.

Yoon, SY., Laffey, J. (2008) Understanding Usability and User Experience of Web-Based 3D Graphics Technology, *International Journal of Human-Computer Interaction*, 24(3), pp. 288-306

Zhang, X., Gao, Y. (2009) Generalised Ambient Reflection Models for Lambertian and Phong Surfaces, *Proceedings of the 16th IEEE International Conference on Image Processing*, pp. 3993-3996. Piscataway, USA, November 2009.

Zink, J., Pettineo, M. and Hoxley, J. (2011) *Practical Rendering and Computation with Direct3D 11*. Florida, USA: CRC Press.

Appendix A

1. SVG file for a triangle:

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd"[]>
<svg viewBox="-150 -150 300 300" style="background-colour: #7d7d7d"
shape-rendering="geometricPrecision"
xmlns:xlink="http://www.w3.org/1999/xlink"
xmlns="http://www.w3.org/2000/svg">
  <script type="text/ecmascript" xlink:href="../Jscript/Mouse.js" />
  <defs />
  <g id="t01" enable-background="new">
    <path d="M0 90L180 90 180 -90z" style="fill: #54ac00 ;stroke:
#54ac00 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
  </g>
</svg>
```

2. SVG file for a plane:

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd"[]>
<svg viewBox="-150 -150 300 300" style="background-colour: #7d7d7d"
shape-rendering="geometricPrecision"
xmlns:xlink="http://www.w3.org/1999/xlink"
xmlns="http://www.w3.org/2000/svg">
  <script type="text/ecmascript" xlink:href="../Jscript/Mouse.js" />
  <defs />
  <g id="p1" enable-background="new">
    <path d="M0 0L90 -90 0 -90z" style="fill: #54ac00 ;stroke:
#54ac00 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
    <path d="M0 0L90 0 90 -90z" style="fill: #54ac00 ;stroke:
#54ac00 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
  </g>
</svg>
```

3. SVG file for a Cylinder:

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd"[]>
<svg viewBox="-150 -150 300 300" style="background-colour: #7d7d7d"
shape-rendering="geometricPrecision"
xmlns:xlink="http://www.w3.org/1999/xlink"
xmlns="http://www.w3.org/2000/svg">
   <script type="text/ecmascript" xlink:href="../Jscript/Mouse.js" />
   <defs />
   <g id="body" enable-background="new">
      <path d="M0 -128.6L0 0 -13.3 -128.9z" style="fill: #5f5f00 ;stroke:
#5f5f00 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
      <path d="M0 0L-13.3 0 -13.3 -128.9z" style="fill: #5f5f00 ;stroke:
#5f5f00 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
      <path d="M13.3 0L0 0 0 -128.6z" style="fill: #484800 ;stroke:
#484800 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
      <path d="M13.3 -128.9L13.3 0 0 -128.6z" style="fill: #484800 ;stroke:
#484800 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
      <path d="M25.4 -129.8L25.4 0 13.3 -128.9z" style="fill: #2c2c00 ;stroke:
#2c2c00 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
      <path d="M-13.3 0L-25.4 0 -25.4 -129.8z" style="fill: #6d6d00 ;stroke:
#6d6d00 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
      <path d="M-13.3 -128.9L-13.3 0 -25.4 -129.8z" style="fill: #6d6d00 ;stroke:
#6d6d00 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
      <path d="M25.4 0L13.3 0 13.3 -128.9z" style="fill: #2c2c00 ;stroke:
#2c2c00 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
      <path d="M-25.4 0L-35.4 0 -35.4 -131.1z" style="fill: #727200 ;stroke:
#727200 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
      <path d="M-25.4 -129.8L-25.4 0 -35.4 -131.1z" style="fill: #727200 ;stroke:
#727200 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
      <path d="M35.4 -131.1L35.4 0 25.4 -129.8z" style="fill: #0c0c00 ;stroke:
#0c0c00 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
      <path d="M35.4 0L25.4 0 25.4 -129.8z" style="fill: #0c0c00 ;stroke:
#0c0c00 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
      <path d="M-35.4 0L-42.1 0 -42.1 -132.9z" style="fill: #6d6d00 ;stroke:
#6d6d00 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
```

```
        <path d="M-35.4 -131.1L-35.4 0 -42.1 -132.9z" style="fill: #6d6d00 ;stroke:
#6d6d00 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
        <path d="M42.1 -132.9L42.1 0 35.4 -131.1z" style="fill: #000000 ;stroke:
#000000 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
        <path d="M42.1 0L35.4 0 35.4 -131.1z" style="fill: #000000 ;stroke:
#000000 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
        <path d="M-42.1 0L-45 0 -45 -135z" style="fill: #5f5f00 ;stroke:
#5f5f00 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
        <path d="M-42.1 -132.9L-42.1 0 -45 -135z" style="fill: #5f5f00 ;stroke:
#5f5f00 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
        <path d="M45 -135L45 0 42.1 -132.9z" style="fill: #000000 ;stroke:
#000000 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
        <path d="M45 0L42.1 0 42.1 -132.9z" style="fill: #000000 ;stroke:
#000000 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
        <path d="M43.5 -137.1L43.5 0 45 -135z" style="fill: #5fc100 ;stroke:
#5fc100 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
        <path d="M43.5 0L45 0 45 -135z" style="fill: #5fc100 ;stroke:
#5fc100 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
        <path d="M-45 -135L-45 0 -43.5 -137.1z" style="fill: #000000 ;stroke:
#000000 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
        <path d="M-45 0L-43.5 0 -43.5 -137.1z" style="fill: #000000 ;stroke:
#000000 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
        <path d="M37.5 0L43.5 0 43.5 -137.1z" style="fill: #6ddf00 ;stroke:
#6ddf00 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
        <path d="M37.5 -139.1L37.5 0 43.5 -137.1z" style="fill: #6ddf00 ;stroke:
#6ddf00 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
        <path d="M-43.5 -137.1L-43.5 0 -37.5 -139.1z" style="fill: #000000 ;stroke:
#000000 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
        <path d="M-43.5 0L-37.5 0 -37.5 -139.1z" style="fill: #000000 ;stroke:
#000000 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
        <path d="M27.6 0L37.5 0 37.5 -139.1z" style="fill: #72e900 ;stroke:
#72e900 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
        <path d="M27.6 -140.7L27.6 0 37.5 -139.1z" style="fill: #72e900 ;stroke:
#72e900 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
        <path d="M-37.5 -139.1L-37.5 0 -27.6 -140.7z" style="fill: #0c1900 ;stroke:
#0c1900 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
```

```
        <path d="M-37.5 0L-27.6 0 -27.6 -140.7z" style="fill: #0c1900 ;stroke: #0c1900 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
        <path d="M14.6 -141.7L14.6 0 27.6 -140.7z" style="fill: #6ddf00 ;stroke: #6ddf00 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
        <path d="M14.6 0L27.6 0 27.6 -140.7z" style="fill: #6ddf00 ;stroke: #6ddf00 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
        <path d="M-27.6 -140.7L-27.6 0 -14.6 -141.7z" style="fill: #2c5900 ;stroke: #2c5900 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
        <path d="M-27.6 0L-14.6 0 -14.6 -141.7z" style="fill: #2c5900 ;stroke: #2c5900 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
        <path d="M-14.6 -141.7L-14.6 0 0 -142.1z" style="fill: #489300 ;stroke: #489300 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
        <path d="M-14.6 0L0 0 0 -142.1z" style="fill: #489300 ;stroke: #489300 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
        <path d="M0 0L14.6 0 14.6 -141.7z" style="fill: #5fc100 ;stroke: #5fc100 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
        <path d="M0 -142.1L0 0 14.6 -141.7z" style="fill: #5fc100 ;stroke: #5fc100 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
    </g>
</svg>
```

4. **SVG file for a Cone:**

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN" "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd"[]>
<svg viewBox="-150 -150 300 300" style="background-colour: #7d7d7d" shape-rendering="geometricPrecision"
xmlns:xlink="http://www.w3.org/1999/xlink"
xmlns="http://www.w3.org/2000/svg">
    <script type="text/ecmascript" xlink:href="../Jscript/Mouse.js" />
    <defs />
    <g id="body" enable-background="new">
        <path d="M0 -135L0 0 -13.3 0z" style="fill: #444400 ;stroke: #444400 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
        <path d="M0 -135L13.3 0 0 0z" style="fill: #2e2e00 ;stroke: #2e2e00 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
        <path d="M0 -135L25.4 0 13.3 0z" style="fill: #131300 ;stroke: #131300 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
    <path d="M0 -135L-13.3 0 -25.4 0z" style="fill: #525200 ;stroke: #525200 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
    <path d="M0 -135L-25.4 0 -35.4 0z" style="fill: #565600 ;stroke: #565600 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
    <path d="M0 -135L35.4 0 25.4 0z" style="fill: #000000 ;stroke: #000000 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
    <path d="M0 -135L42.1 0 35.4 0z" style="fill: #000000 ;stroke: #000000 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
    <path d="M0 -135L-35.4 0 -42.1 0z" style="fill: #525200 ;stroke: #525200 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
    <path d="M0 -135L-42.1 0 -45 0z" style="fill: #444400 ;stroke: #444400 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
    <path d="M0 -135L45 0 42.1 0z" style="fill: #000000 ;stroke: #000000 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
    <path d="M0 -135L43.5 0 45 0z" style="fill: #71e700 ;stroke: #71e700 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
    <path d="M0 -135L-45 0 -43.5 0z" style="fill: #000000 ;stroke: #000000 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
    <path d="M0 -135L37.5 0 43.5 0z" style="fill: #7fff00 ;stroke: #7fff00 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
    <path d="M0 -135L-43.5 0 -37.5 0z" style="fill: #050a00 ;stroke: #050a00 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
    <path d="M0 -135L27.6 0 37.5 0z" style="fill: #84ff00 ;stroke: #84ff00 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
    <path d="M0 -135L-37.5 0 -27.6 0z" style="fill: #234700 ;stroke: #234700 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
    <path d="M0 -135L-27.6 0 -14.6 0z" style="fill: #418400 ;stroke: #418400 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
    <path d="M0 -135L14.6 0 27.6 0z" style="fill: #7fff00 ;stroke: #7fff00 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
    <path d="M0 -135L-14.6 0 0 0z" style="fill: #5cbb00 ;stroke: #5cbb00 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
    <path d="M0 -135L0 0 14.6 0z" style="fill: #71e700 ;stroke: #71e700 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
  </g>
</svg>
```

5. **SVG file for Extrusion:**

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd"[]>
<svg viewBox="-150 -150 300 300" style="background-colour: #7d7d7d"
shape-rendering="geometricPrecision"
xmlns:xlink="http://www.w3.org/1999/xlink"
xmlns="http://www.w3.org/2000/svg">
  <script type="text/ecmascript" xlink:href="../Jscript/Mouse.js" />
  <defs />
  <g id="e1" enable-background="new">
    <path d="M-51.4 -14.4L-54 0 -61.1 97.7z" style="fill: #000000 ;stroke:
#000000 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
    <path d="M54 0L51.4 -14.4 58.2 82.8z" style="fill: #65ce00 ;stroke:
#65ce00 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
    <path d="M54 0L58.2 82.8 61.1 97.7z" style="fill: #65ce00 ;stroke:
#65ce00 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
    <path d="M-51.4 -14.4L-61.1 97.7 -58.2 82.8z" style="fill: #000000 ;stroke:
#000000 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
    <path d="M-42.4 -27.4L-51.4 -14.4 -58.2 82.8z" style="fill:
#000100 ;stroke: #000100 ;stroke-width: 1; stroke-linejoin: round" opacity="1"
/>
    <path d="M51.4 -14.4L48.1 69.4 58.2 82.8z" style="fill: #7dff00 ;stroke:
#7dff00 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
    <path d="M51.4 -14.4L42.4 -27.4 48.1 69.4z" style="fill: #7dff00 ;stroke:
#7dff00 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
    <path d="M-27.9 -37.2L-42.4 -27.4 -48.1 69.4z" style="fill:
#2b5900 ;stroke: #2b5900 ;stroke-width: 1; stroke-linejoin: round" opacity="1"
/>
    <path d="M-42.4 -27.4L-58.2 82.8 -48.1 69.4z" style="fill: #000100 ;stroke:
#000100 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
    <path d="M42.4 -27.4L31.7 59.2 48.1 69.4z" style="fill: #88ff00 ;stroke:
#88ff00 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
    <path d="M42.4 -27.4L27.9 -37.2 31.7 59.2z" style="fill: #88ff00 ;stroke:
#88ff00 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
    <path d="M-27.9 -37.2L-48.1 69.4 -31.7 59.2z" style="fill: #2b5900 ;stroke:
#2b5900 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
```

```
        <path d="M-9.7 -42.6L-27.9 -37.2 -31.7 59.2z" style="fill: #53a900 ;stroke: #53a900 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
        <path d="M27.9 -37.2L9.7 -42.6 11.1 53.7z" style="fill: #84ff00 ;stroke: #84ff00 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
        <path d="M27.9 -37.2L11.1 53.7 31.7 59.2z" style="fill: #84ff00 ;stroke: #84ff00 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
        <path d="M9.7 -42.6L-9.7 -42.6 -11.1 53.7z" style="fill: #72e800 ;stroke: #72e800 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
        <path d="M9.7 -42.6L-11.1 53.7 11.1 53.7z" style="fill: #72e800 ;stroke: #72e800 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
        <path d="M-9.7 -42.6L-31.7 59.2 -11.1 53.7z" style="fill: #53a900 ;stroke: #53a900 ;stroke-width: 1; stroke-linejoin: round" opacity="1" />
    </g>
</svg>
```