Implementing Surfaces in OpenGL

by

Hui Zhao

A research paper presented to the

University of Waterloo

in partial fulfilment of the requirement for the degree of

Master of Mathematics

in

Statistics

Waterloo, Ontario, Canada, 2006

Acknowledgments

I would like to thank Professor Wayne Oldford for all the help given in this essay.

Contents

1 Why OpenGL **6**

1.1 OpenGL Highlights 7

1.2 The objective of this project 9

2 OpenGL techniques overview **12**

2.1 An Introduction 12

2.2 OpenGL Graphic Primitives and Attributes 13

2.2.1 Colour 15

2.2.2 Points, Line segments 15

2.2.3 Polygons 16

2.3 Geometric transformations and Projections 18

2.3.1 Affine transformations 19

2.3.2 Translation 20

2.3.3 Rotation 21

2.3.4 Scaling 23

2.3.5 Projection transformations 25

2.4 Three-dimensional viewing 27

2.4.1 World coordinates to viewing coordinates 28

2.4.2 OpenGL routines 30

2.5 Surface Rendering 31

2.5.1 Lighting 31

2.5.2 Illumination models 32

2.5.3 Polygonal rendering methods 33

2.5.4 Illumination and surface rendering in OpenGL 33

3 OpenGL Implementation in MCL **35**

3.1 MCL to OpenGL Interface 35

3.1.1 Entry Points and Records 36

3.1.2 Implementation 37

3.1.3 Pseudo Code 38

3.2 AGL Programming Overview 38

3.2.1 General procedure 39

3.2.2 Code demonstration 40

4 Surface Implementation **43**

4.1 Design 43

4.1.1 Class diagram 43

4.1.2 Class details 45

4.2 Implementation 47

4.2.1 Wire-Frame 47

4.2.2 Surface rendering 48

4.2.3 Rotation 51

4.2.4 Scaling 51

5 Summary **52**

Bibliography **54**

Chapter 1

Why OpenGL

Introduced in 1992, OpenGL has become the industry's most widely used programming interface supporting two dimensional and three dimensional graphics application. Quail is an interactive, display-oriented, quantitative programing environment for data analysis and visualization. According to Oldford(1998, page 3) "An interactive and dynamic visual display model is the key to statistical analysis, the key to fostering the illusion that the user is working directly with their familiar concepts, and the key to an integrated support environment. And this integrated environment must be flexible so as to allow the analyst to pursue new lines of attack whenever appropriate." Quail has its own self-contained graphics system, but could take advantage of a more sophisticated graphical development environment.

OpenGL is a natural choice for the evolution of Quail.

1.1 OpenGL Highlights

OpenGL has the following advantages.

- Simplicity in Software Development

  The OpenGL design is intuitive and logical. The result is that powerful graphics applications can be developed relatively quickly and simply based on OpenGL. In addition, OpenGL drivers encapsulate information about the underlying hardware, freeing the applications from having to design for specific hardware features.

- Multi-platform Availability

  OpenGL runs on every major operating system including Mac OS X, OS/2, many variations of UNIX (e.g. SUN Solaris, IBM AIX, HP UX, Linux) with X windows, and Microsoft windows systems. It also works with every major windowing system, including Microsoft Windows, Mac OS X (i.e. Quartz, the heart of the Mac OS X windowing environment), Presentation Manager, and X-Window System.

- Stability and Reliability

OpenGL implementations have been available on all platforms for years. Additions to the specification are well controlled, and proposed updates are announced in time for developers to adopt changes. Backward compatibility requirements ensure that existing applications do not become obsolete.

- High Visual Quality and Performance

  When available, OpenGL directly uses graphics processing hardware to improve rendering speeds. Graphics card vendors, such as 3Dlabs, ATI Technologies and NVIDIA, have developed high-end graphics chip technology. These graphics chips implement most of the OpenGL functions, and only a few OpenGL routines are left for CPU processing. 3Dlabs' GLINT 300SX graphics processor was the industry's first full 3D-capable graphics chip. Where once this kind of graphics chip was expensive and only available on high-end workstations and supercomputers, today there are more and more low-cost OpenGL accelerator chips having a least some OpenGL functions implemented in hardware for general PC users. No doubt, the 2D and 3D gaming industry has been the driving force behind this wide availability of graphics hardware.

1.2 The objective of this project

Quail is an interactive, statistical, data visualization programming environment. We emphasize data visualization, because statistical analysis can benefit immensely from it. Visualization can provide understanding of statistical data, and can reveal intricate structure that might not be obvious from a set of numbers. Examining a surface, either from a real world data set or from explicit functions, is useful to statistical analysis. For example, "A statistical analyst may need to visualize surfaces generated by a parameterized regression model or a non-parametric smoothing algorithm" (Poirier 1992, page 4).

A surface plot is one of Quail's graphics facilities. We focus on surface plots as the first step to have Quail make use of OpenGL. The current surface plot makes no use of graphics hardware acceleration since complex displaying operations, for example rotation, are all done in Quail and use the CPU only.

OpenGL will solve these problems. Furthermore, OpenGL can achieve more powerful visual effects and functionality, such as applying lighting, surface material properties (e.g. transparent or translucent surface), different textures, etc, none of which are available in the current Quail graphics package. This paper uses OpenGL to implement surface plots to explore the power of OpenGL within Quail.

Currently Quail runs on both MAC OS X and Microsoft platform. Choosing

MAC OS X as the starting point, there are three basic tasks ahead of us.

1. Make OpenGL accessible from MCL

   OpenGL is callable from Ada, C++, Fortran, Python, Perl and Java through various OpenGL bindings which are found in OpenGL official web-site (OpenGL Web). Meanwhile, calling OpenGL from Common Lisp is broadly being practised (Zhao, 2006). The OpenGL binding for Allegro in UNIX platform by R. Mann (Mann, 1998) was one of these pioneer attempts. The OpenGL engine in AgentSheet's free "OpenGLforMCL" package (AgentSheet, 2005) transplanted R. Mann's work to Mac OS X platform. This package constructs the infrastructure of our OpenGL application in Quail.

2. Provide a windowing system for the OpenGL drawing context

   The OpenGL libraries do not provide interactive input and output routines. An Apple-specific programming interface is needed to communicate with the Mac OS X windowing system. Mac OS offers three application programming interfaces (or APIs). They are NSOpenGL classes for Cocoa applications, the AGL API for Carbon applications, and the CGL API accessible for both Cocoa and Carbon applications. These Apple specific APIS do not create OpenGL content. They are what Cocoa and Carbon applications use to communicate with the Mac OS X windowing system. This paper will use AGL

API in a MCL environment to communicate with the Mac OS X windowing system.

3. Identify the most effective OpenGL techniques to implement a surface.

This paper assumes readers have little OpenGL experience. The goal is to tackle problems and questions associated with implementing surfaces using OpenGL in a common lisp environment.

This essay first introduces relevant OpenGL techniques, and then gives a detailed description of an implementation.

Chapter 2

OpenGL techniques overview

2.1 An Introduction

OpenGL provides a basic or core library of functions for specifying graphics primitives, attributes, geometric transformations, viewing transformations and many other operations. To be hardware independent, the core library does not provide interactive input and output routines; however these routines are available in auxiliary libraries.

Generally speaking, there are three kinds of items available from the OpenGL basic library which are distinguished by their prefix: OpenGL functions, symbolic constants and built-in data types. The prefixes are part of OpenGL naming con-

ventions.

On top of GL stands GLU, standing for "Graphics Library Utilities". GLU functions extend the GL functions, and these routines are distinguished by having the prefix "glu". Furthermore, OpenGL Utility Toolkit (GLUT) contains a set of functions with prefix "glut" to interact with windowing systems. Table 2.1 gives a summary of OpenGL libraries.

Since OpenGL libraries contain only device independent graphics functions, and window-management operations depend on the specific computer system, we cannot create the display window directly. However, there are several window-system libraries that support OpenGL for a variety of machines. In most Unix system, the OpenGL Extension to the X window system(GLX) fulfills this role. In Mac OS X, in addition to GLX, the AGL interface can implement window-management operation.

2.2 OpenGL Graphic Primitives and Attributes

A computer scene contains various components which can be called graphics output primitives. These geometric primitives include points, straight line segments, circles, conic sections, and polygons. All geometric primitives are positioned in a reference frame called the world coordinate system. Furthermore, each geometric

Table 2.1: A summary OpenGL Libraries

Library	Functionality	Prefix
Core	Specify graphics primitive, attributes, geometric transformations, viewing transformations and many other operations	"gl" for function, "GL_" for symbolic constant, "GL" for built-in data type
Utility	Set up viewing and projection matrices, draw curves, spheres, or tessellations, display quadrics and B-splines, process the surface rendering operations or some other task	"glu"
Utility Toolkit	Contains a set of functions to interact with windowing system	"glut"

primitive has associated with it various attributes such as size and color. In general, there are two methods used for dealing with attributes. First an OpenGL function can be called with attribute values as arguments. Secondly, and more typically, OpenGL attributes are treated as state variables. All OpenGL primitives are displayed with the current state values until these attribute setting are changed.

2.2.1 Colour

Colour is a basic attribute for all primitives, and this can be represented by a red-green-blue triple (R, G, B). OpenGL provides two modes: RGB and RGBA . The difference is that RGBA has a fourth parameter, α, for color blending of which simulation of transparency, or translucent effects is an important application. The most often used color routines are

$$glColor(colorComponent) \tag{2.1}$$

and

$$glClearColor(colorComponent) \tag{2.2}$$

2.2.2 Points, Line segments

A point is given by either a two dimensional or a three dimensional position in the world coordinate system. The attributes associated with points include colour and

size. For a raster system, the point size is an integer multiple of the pixel size, so that a large point is displayed as a square block of pixels. The following OpenGL routines create two points located at (x, y, z) and (x^*, y^*, z^*).

```
glBegin(GL_POINT)
  glVertex(x, y, z)
  glVertex(x*, y*, z*)
glEnd()
```

OpenGL line segments are defined by two endpoint coordinate positions. We enclose a list of glVertex functions between a glBegin/glEnd pair, and use symbolic constant GL_LINES to tell OpenGL to connect these endpoints to form a set of straight line segments. A straight line segment can be displayed with three basic attributes: color, width, and line style.

2.2.3 Polygons

For this surface implementation, a filled area is an important graphics component. Although in principle any filled area shape is possible, we will only be concerned about convex polygons as provided by OpenGL. Just as a curved line can be approximated with a set of line segments, a curved surface can be described by a set of polygonal facets, sometimes referred to as a polygon mesh. This is how we will

implement surfaces. A polygon is a plane segment determined by a set of vertices, and the order in which they are to be connected by line segments. In OpenGL, a polygon must be convex.

We also distinguish the back face from the front face of a polygon by imagining an object enclosed by a set of polygonal surfaces. The side of a polygon facing into an object is called the back face, the outward side is the front face. Color and other attributes can be set for the back face and the front face separately.

Mathematically, a plane in $\mathbf{R}^3$ is represented as

$$Ax + By + Cz + D = 0 \tag{2.3}$$

A vector space is formed through multiplying the vector (A, B, C) by constants. The normal vector of the polygon contained by the plane represented by the equation 2.3 belongs to this vector space. This normal vector is perpendicular to the polygon, and points in a direction from the back face of the polygon to its front face. Figure 2.1 shows a normal vector of a polygon.

Polygon vertices must be specified in a $COUNTERCLOCKWISE$ order as we view the polygon from "outside" or view its front face. The following example demonstrates how to produce the polygon shown in Figure 2.1 in OpenGL

```
glBegin(GL_POLYGON)
```

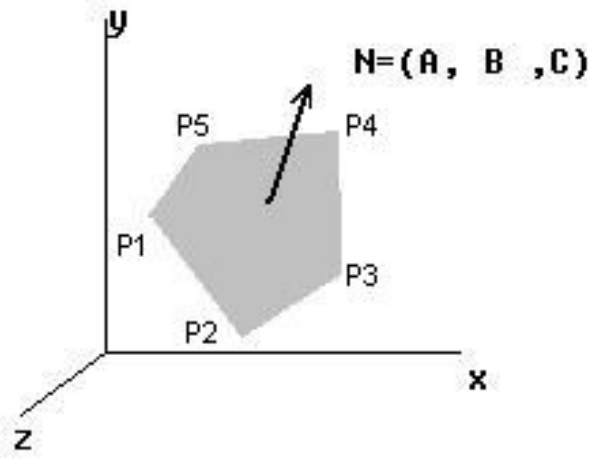

Figure 2.1: The normal vector N of a polygon

```
        glVertex(P1)
        glVertex(P2)
        glVertex(P3)
        glVertex(P4)
        glVertex(P5)
    glEnd()
```

2.3 Geometric transformations and Projections

For computer graphics, geometric transformations such as translation (i.e moving an object), rotation, and scaling are fundamental operations. We will briefly discuss

these first in this section, and then discuss projections, that is a three dimensional object has to be projected down to two dimensions before it can be printed or displayed on a two dimensional graphics output device.

2.3.1 Affine transformations

An affine transformation has the form

$$\mathbf{t*} = A\mathbf{t} + \mathbf{c} \tag{2.4}$$

or in more detail

$$\begin{pmatrix} x^* \\ y^* \\ z^* \end{pmatrix} = \begin{pmatrix} a_{xx} & a_{xy} & a_{xz} \\ a_{yx} & a_{yy} & a_{yz} \\ a_{zx} & a_{zy} & a_{xz} \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} c_x \\ c_y \\ c_z \end{pmatrix}$$

here

$$\mathbf{t} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \quad and \quad \mathbf{t*} = \begin{pmatrix} x^* \\ y^* \\ z^* \end{pmatrix}$$

Each of the transformed coordinates x^*, y^* and z^* is a linear function of the original coordinates x, y, and z.

More conveniently, homogeneous coordinates allow affine transformations to be easily represented by a single matrix. By using homogeneous coordinates, Equation

2.4 can be represented as

$$\mathbf{p*} = A\mathbf{p} \tag{2.5}$$

which is expanded as

$$\begin{pmatrix} \mathbf{t*} \\ 1 \end{pmatrix} = \begin{pmatrix} A & \mathbf{c} \\ \mathbf{0}^T & 1 \end{pmatrix} \cdot \begin{pmatrix} \mathbf{t} \\ 1 \end{pmatrix}$$

where $\mathbf{t}^*$, $\mathbf{t}$, $\mathbf{c}$ and $\mathbf{0}$ are 3 by 1 column vectors. A is a 3 by 3 matrix.

Geometric transformations are examples of affine transformations.

2.3.2 Translation

A translation moves an object from one position to another without a change of shape or scale. A point $P = (x, y, z)$ in three dimensional space is shifted to a new location $P^* = (x^*, y^*, z^*)$ by adding shifts d_x, d_y, d_z. Mathematically, this can be represented in the matrix format by

$$\begin{pmatrix} x^* \\ y^* \\ z^* \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Or

$$\mathbf{p}^* = T\mathbf{p} \tag{2.6}$$

where T is the translation matrix.

In OpenGL, the 4 by 4 translation matrix T is produced by the function call

$$glTranslate(d_x, d_y, d_y) \tag{2.7}$$

This new generated translation matrix will be multiplied by the current matrix, and the current matrix will be updated. The new position is determined by the current matrix. In OpenGL, with the following function, we can assign the identity matrix to the current matrix

$$glLoadIdentity() \tag{2.8}$$

2.3.3 Rotation

The easiest three dimensional rotation is to rotate an object around the Cartesian coordinate axes. The transformation matrices are given by the following equations:

around z axis

$$\begin{pmatrix} x^* \\ y^* \\ z^* \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

around x axis

$$\begin{pmatrix} x^* \\ y^* \\ z^* \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} . \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

around y axis

$$\begin{pmatrix} x^* \\ y^* \\ z^* \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} . \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

or is represented as

$$\mathbf{p}^* = R(\theta).\mathbf{p} \tag{2.9}$$

The rotation around any arbitrary rotation axis in space can be done in five steps

1. Translate the object so that the rotation axis passes through the coordinate origin

2. Rotate the object so that the axis of rotation coincides with one of the coordinate axes. This can be accomplished by two steps of rotation operations around the Cartesian coordinate axes. First, rotate the object so that the axis of rotation falls into the x-y plane, the x-z plane, or the z-y plane. Next,

rotate the object around the Cartesian coordinate axis which is perpendicular to the plane containing the axis of rotation.

3. Conduct the specified rotation around the coincident coordinate axis

4. Apply inverse rotation of step 2 to bring the rotation axis back to its original orientation

5. Apply the inverse translation of step 1 to bring the rotation axis back to its original spatial position

OpenGL provides the routine

$$glRotate(theta, rx, ry, rz) \tag{2.10}$$

to accomplish a rotation. Here theta is a rotation angle in degrees, and the r = (rx, ry, rz) defines the orientation of the rotation axis. This OpenGL command produces a 4 by 4 transformation matrix.

2.3.4 Scaling

To alter the size of an object, we apply a scaling transformation. A three dimensional scaling operation is performed by multiplying object positions (x, y, z) by scaling factor s_x, s_y and s_z to produce the transformed coordinates (x^*, y^*, z^*).

The transformation matrix is given by

$$\begin{pmatrix} x^* \\ y^* \\ z^* \\ 1 \end{pmatrix} = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_y & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

or is represented as

$$\mathbf{p}^* = S\mathbf{p} \tag{2.11}$$

Scaling an object will change the position of the object relative to the coordinate origin. We can construct a scaling transformation with respect to any fixed position (x, y, z) by using the following transformations.

1. Shift the fixed point to the coordinate origin
2. Apply scaling transformations.
3. Shift the fixed point back to its original position

In OpenGL, the 4 by 4 scaling matrix S is produced by the function call

$$glScale(s_x, s_y, s_y) \tag{2.12}$$

2.3.5 Projection transformations

Here, we only clarify some terminology. Orthogonal projection, or orthographic projection, shown in Figure 2.2, is a transformation of an object into a projection plane along the direction of the projection plane's normal vector. In oblique parallel projections, shown in Figure 2.3, the projection path is not perpendicular to the projection plane, and coordinate positions are transferred to the projection plane along parallel lines. However in perspective projection, shown in Figure 2.4, the projection is along paths which converge to a point called the projection reference point.

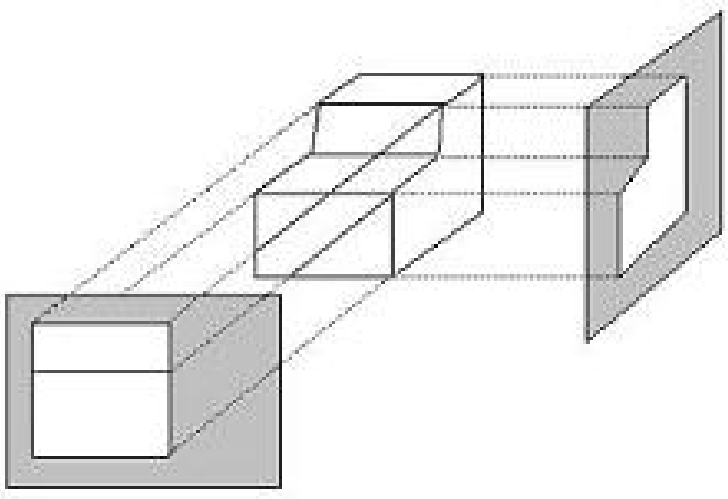

Figure 2.2: Orthogonal projections of an object

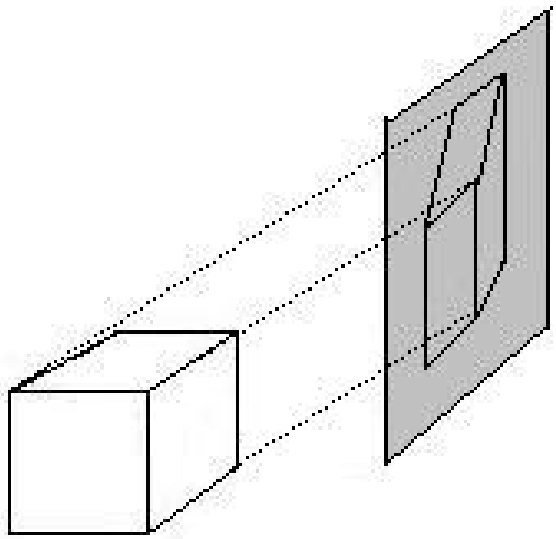

Figure 2.3: Oblique parallel projections of an object

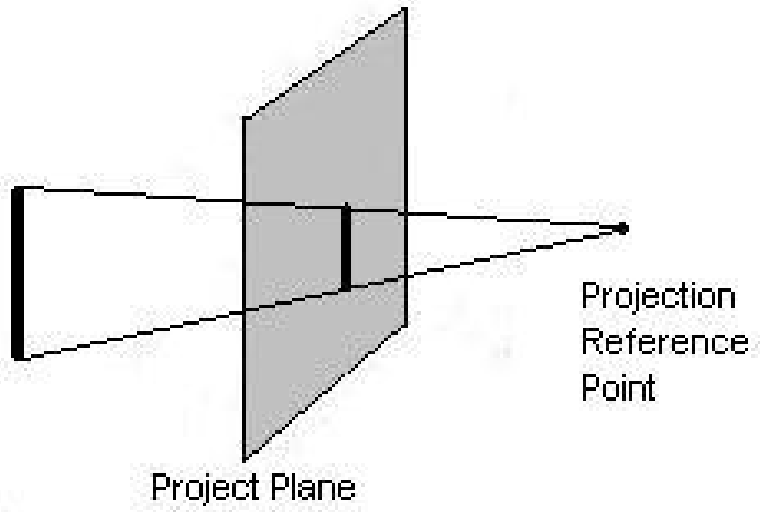

Figure 2.4: Perspective projections of an object

2.4 Three-dimensional viewing

Procedures for generating a three dimensional scene are analogous to that of taking a photo. First of all, choose a position to place the camera, and orient the camera in the direction to be photographed, and then snap the shutter. Figure 2.5 shows the general processing steps for creating and transforming a three dimensional scene to the output device.

Figure 2.5: Three dimensional viewing pipeline

Given a scene constructed in the world coordinate, a viewing coordinate system is defined by given the origin of the viewing coordinate system and the orientation of the projection plane, which we can imagine as a camera film plane. Usually this projection plane is perpendicular to the z axis. Mathematically, the objects in the world coordinate are transformed to the viewing coordinates to project the

objects to the projection plane. Following up the projection operation, objects are mapped to the normalized coordinates. Finally, the normalized coordinates are transferred to a viewport specified in the device coordinates. This process is called the three-dimensional viewing pipeline.

2.4.1 World coordinates to viewing coordinates

A viewing coordinate system is defined by selecting a viewing origin $P_0 = (x_0, y_0, z_0)$ in world coordinate, and a coordinate reference frame. The reference frame here is determined by 3 vectors u, v, n, and is called a uvn viewing coordinate reference frame. Vector u, (u_x, u_y, u_z), defines the orientation of the x_{view} axis. Vector v, (v_x, v_y, v_z), and vector n, (n_x, n_y, n_z), define the y_{view} axis and the z_{view} axis respectively. The vectors u, v, and n are pairwise orthonormal. Typically, the projection plane in the viewing coordinates is parallel to the x-y plane. Figure 2.6 shows a uvn viewing coordinate reference frame.

The transformation from the world to the viewing coordinates can be accomplished by the following matrix

$$M = R \times T \tag{2.13}$$

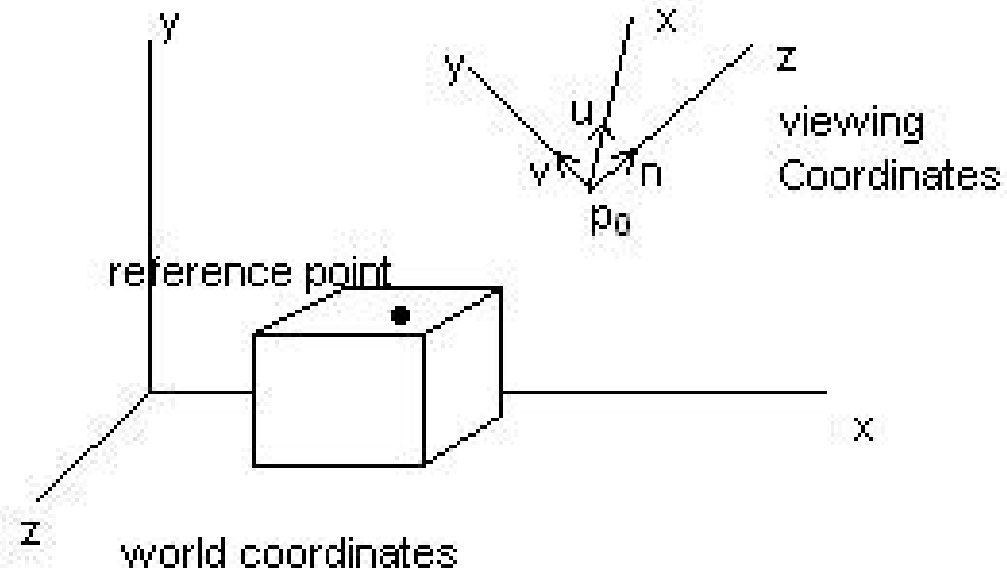

Figure 2.6: A uvn viewing coordinate reference frame

where R is

$$R = \begin{pmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

and T is

$$T = \begin{pmatrix} 1 & 0 & 0 & -x_0 \\ 0 & 1 & 0 & -y_0 \\ 0 & 0 & 0 & -z_0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

2.4.2 OpenGL routines

The OpenGL viewing transformation function is given by

$$glLookAt(x_0, y_0, z_0, x_{ref}, y_{ref}, z_{ref}, V_x, V_y, V_z) \tag{2.14}$$

where $P_0 = (x_0, y_0, z_0)$ is the origin of the viewing coordinate system, $P_{ref} = (x_{ref}, y_{ref}, z_{ref})$ is the center of the scene we want to aim the camera, the vector $V = (V_x, V_y, V_z)$ is the orientation of y_{view} axis.

The OpenGL orthogonal projection routine is given by

$$glOrtho(x_{wmin}, x_{wmax}, y_{wmin}, y_{wmax}, d_{near}, d_{far}) \tag{2.15}$$

where the argument (x_{wmin}, y_{wmin}) and (x_{wmax}, y_{wmax}) define the clipping window in the viewing plane, and (d_{near}, d_{far}) describe the view volume.

OpenGL does not provide oblique parallel projection. Perspective projection is given by

$$glFrustum(x_{wmin}, x_{wmax}, y_{wmin}, y_{wmax}, d_{near}, d_{far}) \tag{2.16}$$

2.5 Surface Rendering

A surface in real life is visible only in the presence of a light source. This light source can have a variety of shapes and characteristics. The source is typically separate from the surface and shining on it, but it might also be the case that the surface is the source itself. The term, surface rendering, means a procedure for applying a illumination model to calculate the color of an illuminated position on the surface. Here, mainly we consider two elements: surface properties, and lighting effects. First of all, we introduce lighting.

2.5.1 Lighting

A light source is an object to illuminate the scene. We can specify its position, the color, the emission direction, and its shape. The simplest light source is point

light source with a single color. Radial intensity attenuation is a factor we have to take into account, otherwise undesirable display effects can result. Radial intensity attenuation describes the attenuation of light amplitude as distance increases, when a light source travels outward. a_0, a_1 and a_2 are three parameters we could use to adjust this intensity attenuation based on some empirical model .

2.5.2 Illumination models

In a basic illumination model, we usually consider background lighting to give the scene a general brightness. This background lighting is called ambient with a parameter I_d, determining its level of the intensity. We introduce two terminologies: diffuse reflection and specular reflection. Imagine a surface that is rough, and so tends to scatter the light in all directions. This scattered light is called diffuse reflection. However, for some shiny material the reflected light is concentrated into a highlight spot, called specular reflection. K_d, the diffuse-reflection coefficient, is a parameter which determines the fraction of the incident light that is to be scattered. When combined with ambient lighting, K_a, the ambient-reflection coefficient, is assigned to a surface to adjust the lighting effects. It also could be expected, the parameter n_s, specular-reflect exponent, is determined by the type of a surface, where a very shiny surface is modelled with a large value, and smaller values are

used for duller surfaces. All the parameters stated above will be used for setting OpenGL surface properties.

2.5.3 Polygonal rendering methods

The simplest method for rendering a polygonal surface is to assign the same color to all surface points inside that polygon. This approach is called flat surface rendering. Another method is called Gouraud surface rendering which interpolates the intensities of colors across the polygon area, which can eliminate the intensity discontinuities that occur in flat surface rendering.

2.5.4 Illumination and surface rendering in OpenGL

OpenGL routine *glLight* is used to set up a point light source and designate its position, type, color, and other properties. Note that, we could have maximum eight light sources at same time in an OpenGL scene.

Reflection coefficients and other optical properties associated with a surface are set by using the function

$$glMaterial(surface, surfaceProperties, PropertiesValues) \tag{2.17}$$

OpenGL routine only provides flat-intensity surface rendering and Gouraud

surface rendering which is indicated by

$$glShadeModel(surfaceRenderingMethod) \tag{2.18}$$

Chapter 3

OpenGL Implementation in MCL

Chapter 1 lists three tasks to complete a surface by using OpenGL in Mac OS X environment. This chapter detailedly discusses how to make OpenGL accessible from MCL and how to set up a windowing system in which OpenGL drawing is displayed onscreen.

3.1 MCL to OpenGL Interface

OpenGL is shipped as a collection of standard packages with Mac OS X. Meanwhile OpenGL accelerator hardware is also built into every iMac, iBook, PowerBook, PPC. Generally speaking, there are two types of binaries in Mac OS X: Code Fragment Manager (CFM), and Mach-O. CFM works under MacOS 8, 9, and X,

while Mach-O only works under OS X.

MCL is a CFM application. This means calling a CFM library from MCL is easier than calling Mach-O libraries. Note that, MCL does not support calling Mach-O libraries. Even so, it is still possible to use Mach-O libraries. Fortunately, OpenGL CFM libraries are still available in MAC OS X. These libraries are: OpenGLEngine, OpenGLLibrary, OpenGLMemory, OpenGLRenderer, OpenGLRendererATI, and OpenGLUtility.

An obvious task is to access the low-level Mac OS interfaces, prior to any OpenGL trials. MCL provides a mechanism to communicate with MAC OS X by accessing OS Entry Points and Records.

3.1.1 Entry Points and Records

OS Entry Points can be regarded as an interface in MCL which calls procedures in the Mac OS X. For example, the procedure to draw and fill a circle in a window requires calling an entry point, which calls a Mac OS X procedure that knows how to draw and fill a circle. Be aware, the term "trap" and "entry point" are interchangeable in this document, and we do not distinguish between them. Every Mac OS X entry point is described in an interface file located in the "Interfaces" folder within the "Library" folder within the directory where MCL is installed.

When the value of *autoload-traps* is true, information is automatically read from the relevant interface file.

The "#_*symbol*" reader macro tries to load the trap definition of *symbol* from the appropriate interface file and interns _*symbol* in the traps package. For example, #_*DrawString* loads _*DrawString* and interns the symbol _*DrawString* in the traps package. The "require-trap" macro is an alternative way to auto load a trap whether called at read time or at macro expand time. After calling "#_*symbol*" or "require-trap", MCL searches the entry point in known system libraries. Although MCL gives definitions of over 2000 Macintosh entry points, it is still possible to extend the set of libraries to meet the problems. The function "add-to-shared-library-search-path" with parameter "libname", where "libname" is a case-sensitive string which names the shared library in question, informs MCL of any additional libraries to search for entry points. The macro "deftrap" provides a way to define your own trap and its behavior.

3.1.2 Implementation

After examining the file of traps.idx located in the "index" folder within the "interfaces" folder within the installed MCL 5.0 directory, no trap names related to OpenGL were found. Therefore using "add-to-shared-library-search-path" to in-

form MCL the additional OpenGL libraries is necessary.

One strategy to access the underlying OpenGL libraries is to define a wrapper function, having the same name as the OpenGL function, in which we issue a "require-trap" macro to invoke the underlying OpenGL routine.

3.1.3 Pseudo Code

Define a trap in line having the name same as the OpenGL function, and provide arguments, return values, and leave the implementation form blank.

Define a wrapper function

Use the macro "with-cstrs" to allocate memory for return values.

Make underlying OpenGL calls.

Export the wrapper function as an external accessible symbol.

3.2 AGL Programming Overview

Apple Graphics Library (AGL) is the Apple interface to OpenGL for Carbon applications. It can be used by both Mach-O and CFM binaries. AGL provides a mechanism for Carbon applications to communicate with the Mac OS X windowing

system.

3.2.1 General procedure

The following steps describes how a Carbon application provides data for OpenGL processing

1. Set up a list of buffer and renderer attributes that support the OpenGL drawing which will be performed

2. Request, from the operating system, a pixel format object that encapsulates pixel storage information and the renderer and buffer attributes required by the application. The returned pixel format object contains all possible combinations of renderers and displays available on the system that the program runs on and that meets the requirements specified by the attributes.

3. Create a rendering context to hold state information that controls such things as drawing color, view and projection matrices, characteristics of light, and conventions used to pack pixels. The rendering context must be associated with a pixel format which is created in the previous step.

4. Bind a drawable object to the rendering context. A drawable object is what captures the OpenGL drawing sent to the rendering context. A drawable

object can be a Carbon window, offscreen memory, a full-screen graphics device, etc.

5. Make the rendering context the current context. OpenGL automatically targets the current context. Although the application might have several rendering contexts set up, only the current one is the active one for drawing purposes.

6. Issue OpenGL drawing commands.

3.2.2 Code demonstration

The following code fragments (AgentSheet, 2005) give a flavour of how to implement the above six steps in terms of MCL.

```
(let
  (
  ;; Set up an array of attributes that describes the buffer
  ;; characteristics and renderer capabilities.
  ;; The following codes set up attributes of colour, double
  ;; buffering, and a pixel depth of 32 bits
  (Attributes {AGL_RGBA AGL_DOUBLEBUFFER AGL_DEPTH_SIZE 32 AGL_NONE})
```

```
;; create a context
   (rlet
     (
       (GDhandleArray :pointer)
     )
     (%put-ptr GDhandleArray (#_getMainDevice))
     (let
        (
        ;; Obtain a pixel format object which contains a
        ;; list of all appropriate renderer-display combinations
        (Aglpixelformat (aglChoosePixelFormat GDhandleArray 1 attributes))
        )
       (#_disposePtr attributes)
       ;; Bind the pixel format object to a rendering context
       (setf (aglcontext Self)
             (aglCreateContext AGLPixelFormat (global-gl-context)))
       ;;Release the pixel format object
       (aglDestroyPixelFormat AGLPixelFormat)
     )
   )
```

```
;; Get a port associated with the Carbon window by calling the
;; Window Manager function. After a rendering context is attached
;; to the Carbon window, its viewport is set to the full size of
;; the window.
;; Bind the window to the rendering context
(aglSetDrawable (aglcontext Self) (#_GetWindowPort (wptr Self)))
;;Make the rendering context the current context
(aglSetCurrentContext (aglcontext Self))
;; Issue OpenGL command
;; glClear(gl_color_buffer_bit) Indicates the buffers currently
;; enabled for color writing
(glclear gl_color_buffer_bit)
(glfinish)
```

Chapter 4

Surface Implementation

This chapter describes the design and development of a surface by using OpenGL in MCL.

4.1 Design

4.1.1 Class diagram

Figure 4.1 gives a high level class diagram that describes the types of objects, and the static relationship among these classes.

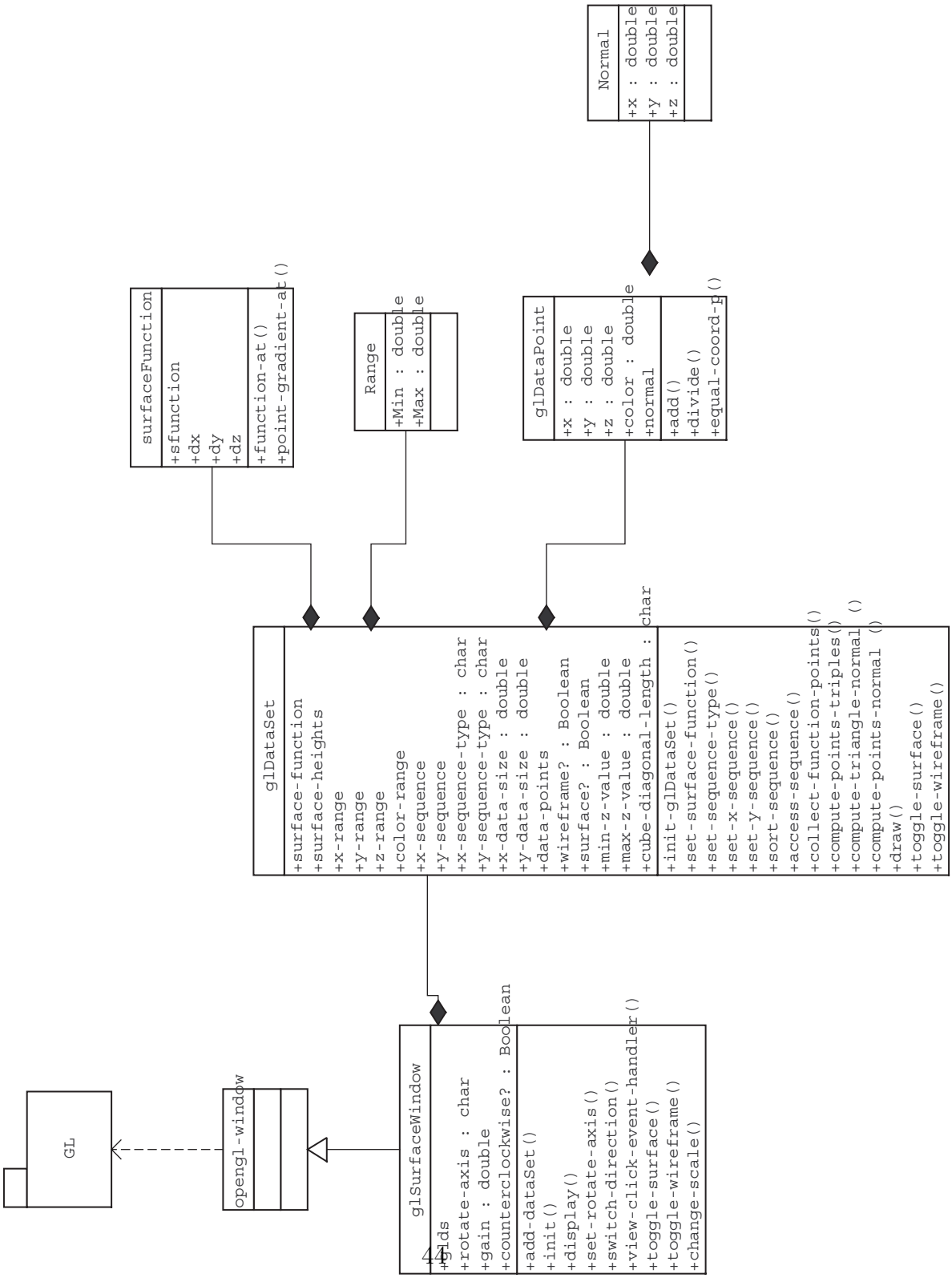

Figure 4.1: Class Diagram

4.1.2 Class details

Class glDataPoint

Class $glDataPoint$ defines a three dimensional position in the world coordinate system located at (x, y, z), having attributes of color and a normal vector at that point.

Class surfaceFunction

Class $surfaceFunction$ delivers two major functionalities:

1. Compute the height by giving a (x, y) pair through evaluating the function at point (x, y).

2. Compute partial derivatives with respect to x, y and z respectively, if analytical results exist.

Class glDataSet

Class $glDataSet$ playes a role as a three dimensional data point container. Class $surfaceFunction$ and class $glDataPoint$ hold an aggregation or composition relationship with class $glDataSet$. That is an instance of class $glDataSet$ has an object

of $surfaceFunction$ and many objects of $glDataSet$ as its parts. Class $glDataSet$'s major responsibilities include:

1. Compute heights by giving either an x range and a y range or an x sequence and a y sequence
2. Compute the normal vector for each date point
3. Assemble data points into a data structure supporting object storage and efficient lookup facilities, for example: a hash table
4. Draw the data set by using the OpenGL commands

Class glSurfaceWindow

Class $glSurfaceWindow$ is inherited, or called to realize a generalization in UML language, from class *Openglwindow* (AgentSheet, 2005). The OpenGL configurations and the event handling are handled here.

4.2 Implementation

4.2.1 Wire-Frame

I used the following methods to implement a wire-frame surface. The objects of three dimensional data points are stored in a hash table. The pair (i, j) is the key for each key-value pair in the hash table. The number i is the ordinal number of the sequence x. The number j is the ordinal number of the sequence y. Four adjacent triples are fetched from the hash table to form a polygon. That is for each point (i, j), the points $(i + 1, j), (i + 1, j + 1)$, and $(i, j + 1)$ are adjacent to point (i, j), and together these four points form a rectangle having point (i, j) as its bottom-left corner. These four vertexes are given to the OpenGL polygon drawing routine in counterclockwise order. Therefore the curved surface is approximated by polygonal facets. Figure 4.2 shows a wire-frame surface for a bivariate normal distribution.

Pseudo code for the wire-frame implementation:

Iterate i M times, where M is the size of the x sequence.

Iterate j N times, where N is the size of the y sequence.

Get the point (i, j) and its three adjacent points from the has table.

Issue OpenGL command to draw a quadrilateral.

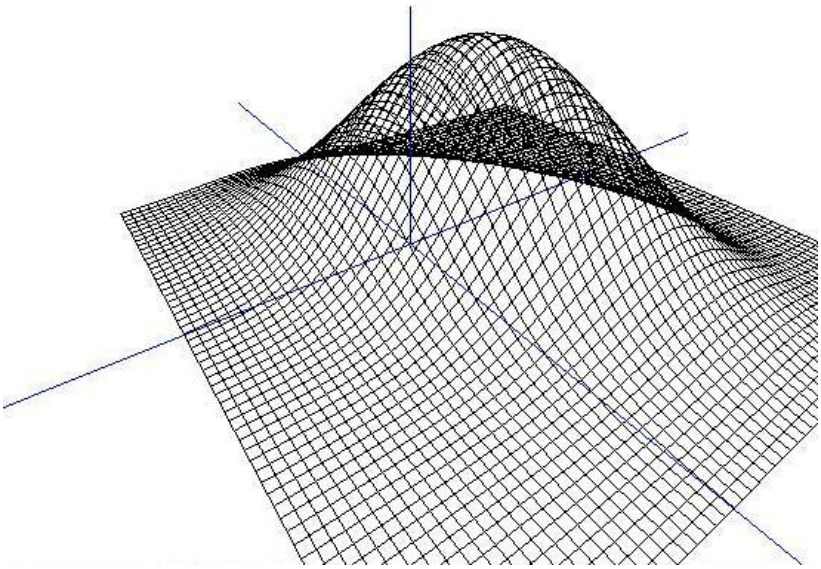

Figure 4.2: A wire-frame surface for a bivariate normal distribution

4.2.2 Surface rendering

I used the Gouraud method to implement the surface rendering. The normal vector for each three dimensional point needs to be calculated. I gave two solutions for computing the normal vector based on the different circumstance.

1. If a surface is given implicitly, as the set of points (x, y, z) satisfying $F(x, y, z) = 0$, then, a normal vector at a point (x, y, z) on the surface is given by its gradient

$$\nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right) \tag{4.1}$$

Quail provides a facility to compute the partial derivatives.

2. If a surface is given explicitly, that is $z = f(x, y)$, then, a normal vector at a point (x, y, z) on the surface is given by

$$\left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, -1\right) \tag{4.2}$$

3. If there are no analytic methods available, a normal vector for each point can be obtained by averaging the normal vectors of all polygons in the surface mesh that share that point. This is illustrated in the figure 4.3

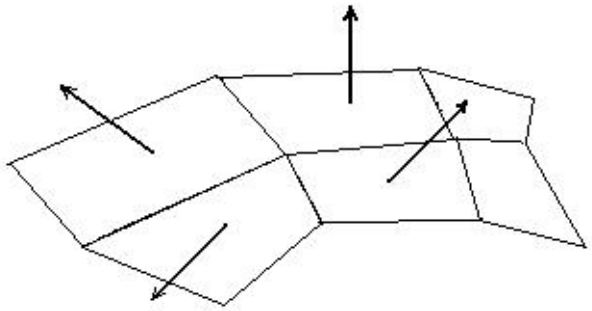

Figure 4.3: A normal is computed by averaging the surrounding normal vectors

Pseudo code for the surface rendering

Set up an ambient lights to give general brightness.

Set up a local lights, and designate its position, diffuse and sepcular properties.

Turn the ambient lights and the local lights on.

Designate the surface material properties.

Designate polygon display mode as GL_FILL.

Iterate i M times, where M is the size of the x sequence.

Iterate j N times, where N is the size of the y sequence.

Get the point (i, j) and its three adjacent points from the has table.

Issue OpenGL routines to draw a polygon having each vertex associated with a normal vector.

Figure 4.4 is an example by using the surface rendering technique for the above wire frame surface.

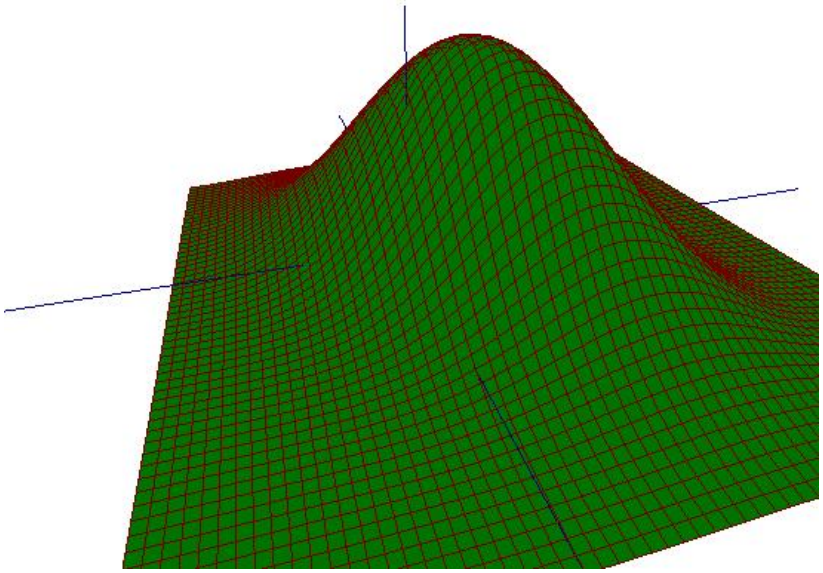

Figure 4.4: Surface rendering for a bivariate normal distribution

4.2.3 Rotation

OpenGL provides the routine $glRotate(theta, rx, ry, rz)$ to complete rotation operations. Where theta is a rotation angle in degree, and vector $r = (rx, ry, rz)$ defines the orientation of the rotation axis. For example, if we want a rotation around z axis with 5 degree, we simply set vector $r = (0, 0, 1)$, and issue the OpenGL command $glRotate(5, 0, 0, 1)$. The change of the speed of the rotation can be achieved by increasing the rotation angle.

Pseudo code for the rotation:

Set up the rotation axis and a rotation angle.

Continue the rotation operation until a stop is requested.

Issue the OpenGL command $glRotate(theta, rx, ry, rz)$.

Redraw the view.

4.2.4 Scaling

OpenGL provides the routine $glScale(s_x, s_y, s_y)$ to complete scaling transformations. Where s_x, s_y and s_z are the scaling factors. For example, if we want to enlarge the object as 2 times as its original size, we simply issue the OpenGL command $glScale(2, 2, 2)$.

Chapter 5

Summary

This paper solved the problem of implementing a surface, either from a real world data set or from explicit or implicit functions, by using the OpenGL technique in a MCL environment. In summary, the three tasks listed in chapter 1 are accomplished through the following methods

1. Make OpenGL accessible from MCL

   AgentSheet's free "OpenGLforMCL" package is used to access OpenGL from MCL (AgentSheet, 2005).

2. Provide a windowing system for the OpenGL drawing context

   AGL APIs is used to construct a drawing context.

3. Identify the most effective OpenGL techniques to implement a surface.

   A surface is approximated with a set of polygon meshes to form a wire-frame surface. And the surface rendering technique is applied to render the surface. The rotation and scaling operations on the surface are also developed.

This is a start. To make Quail fully benefit from OpenGL needs more work. The next step would be to merge the current OpenGL surfaces into a Quail surface plot.

Glossary

- **AGL**

  Apple Graphics Library API. It is part of the Apple implementation of OpenGL in Mac OS X.

- **Allegro**

  Allegro CL. It is an implementation of common lisp language.

- **API**

  Application Programming Interface.

- **Carbon**

  A set of APIs for developing Mac OS X applications.

- **CFM**

  Code Fragment Manager. It is one of Mac OS X application binary formats.

- **CGL**

  The Core OpenGL API. It is the basis for the NSOpenGL classes and AGL

- **CL**

  Common Lisp.

- **Cocoa**

  An object-oriented application environment is designed specifically for developing Mac OS X-only native applications.

- **GL**

  The OpenGL core library.

- **GLU**

  The OpenGL Graphics Library Utilities.

- **GLUT**

  The OpenGL Utility Toolkit.

- **Mach-O**

  Mach object-file-format. It is one of Mac OS X application binary formats.

- **MCL**

  Macintosh Common Lisp. It is a high-level object-oriented programming language.

- **NSOpenGL**

  One of three Apple-specific OpenGL APIs to communicate with the underlying MAC OS X windowing system.

Bibliography

[OpenGL ARB Guide] OpenGL Architecture Review Board, 2005, "OpenGL(R) Programming Guide: The Official Guide toLearning OpenGL(R), Version 2", the Fifth Edition

[OpenGL ARB Reference] OpenGL Architecture Review Board, 2004, "OpenGL(R) Reference Manual: The Official Reference Document to OpenGL, Version 1.4", the Fourth Edition

[Oldford 1998] R.W. Oldford, 1998, "The Quail Project: A Current Overview" University of Waterloo, Canada. An invited paper presented at the 30th Symposium on the Interface, Minneapolis MN

[Hurley Oldford 1999] C.B. Hurley and R.W. Oldford, 1999, "Statistical Graphics in Quail: An Overview". An invited paper presented at the 1999 Biennial Meeting of the International Statistical Institute in Helsinki Finland

[Apple Guide 2004] Apple Computer, Inc., 2004, "OpenGL Programming Guide for Mac OS X"

[Apple Reference 2004] Apple Computer, Inc., 2004, "AGL Framework Reference"

[Guy Steele 1990] Guy L. Steele JR., 1990, "COMMON LISP : the language"Second Edition, Digital Press, 1990

[Digitool 1996] Digitool, Inc., 1996, "Macintosh Common Lisp Reference"

[Hearn 1994] Donald Hearn, 1994, "Computer Graphics" Second Edition, Prentice-Hall, 1994

[Salomon 1999] D. Salomon, 1999, "Computer graphics and geometric modeling" New York : Springer, 1999

[Poirier 1992] Paul Poirier, 1992, "Visualizing Surfaces". A master research paper, University of Waterloo

[Zhao 2006] Hui Zhao, 2006, "A Quick Preview on OpenGL and Common Lisp"

[OpenGL Web] http://www.opengl.org/

[Mann 1998] R Mann, 1998, http://www.cs.uwaterloo.ca/ mannr/software/OpenGL/allegro-xlib-and-gl.tar.gz

[AgentSheet 2005] AgentSheet, 2005, http://www.agentsheets.com/lisp/OpenGLforMCL5.zip

Appendix

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;                              glSurfaceWindow.lisp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Draw surfaces of arbitrary functions by using OpenGL
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; define structures
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defstruct Range
  (min 0 )
  (max 0 )
  (:documentation "structure Range will be used for coordinates, color, size, or
any pairs")
)

(defstruct normal
  (x 0 )
  (y 0 )
  (z 0 )
  (:documentation "structure normal will be used for plan and vertex")
)

(defstruct normals
  (normal1 )
  (normal2 )
  (normal3 )
  (normal4 )
  (:documentation "normals will be used for computing vertex normal when using
mesh patch approximation")
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define class surface function
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defclass surfaceFunction()
   (
    ( sfunction :accessor sfunction-of :initform nil :initarg :sfunction
:documentation "a function")
    ( dx :accessor dx-of :initform nil :initarg :dx :documentation "partial derivative
w.r.t x for point (x, y, z)")
    ( dy :accessor dy-of :initform nil :initarg :dy :documentation "partial derivative
w.r.t y for point (x, y, z)")
    ( dz :accessor dz-of :initform nil :initarg :dz :documentation "partial derivative
w.r.t z for point (x, y, z)")
   )
   (:documentation "a general 2 dimensional function")
)

(defmethod set-sfunction( (self surfaceFunction) function)
  (setf (sfunction-of self) function)
)

;;; compute heights given (x, y) pair
;;; the function is either a list or a common lisp function object
(defmethod function-at ((self surfaceFunction) x y)

  (cond

   ;; the function is a common lisp function object
   ((functionp (sfunction-of self)) (funcall (sfunction-of self) x y)
    )

   ;; the function is a list
   ((listp (sfunction-of self)) (funcall (eval `(lambda(x y) ,(sfunction-of self))) x y)
    )
  )
)
```

```
;;; substitute the prefix q:: by cl::
(defun q-to-cl-functions (fns &rest body)

  "Restores the original meaning of the functions listed, ~
   but only at the top level, not in a called function. ~
   Then evaluates body, returning the value of the last form."

  (let* (
        (a-list
         (mapcar #'(lambda (f) (cons f (find-symbol (string f) (string-upcase "cl"))))
                 fns)
         )
        (new-body (loop for form in body collect (sublis a-list form)))
        )
    (push 'progn new-body)
    new-body
  )
)

;;; substitute the prefix cl:: by q::
(defun cl-to-q-functions (fns &rest body)

  "Restores the original meaning of the functions listed, ~
   but only at the top level, not in a called function. ~
   Then evaluates body, returning the value of the last form."

  (let* (
         (a-list
          (mapcar #'(lambda (f) (cons f (find-symbol (string f) (string-upcase
"q"))))
                  fns)
         )
         (new-body (loop for form in body collect (sublis a-list form)))
        )
    (push 'progn new-body)
    new-body
    )
)

;;; compute the partial derivatives with respect to x, y, z respectively, if analytic
results exist
;;; use q:deriv to compute partial derivatives
;;; first convert cl:* cl:+ cl:expt cl:/ cl:- to q:* q:+ q:expt q:/ q:- for q:deriv
;;; then convert the q:* q:+ q:expt q:/ q:- in the returned results back to cl:* cl:+
cl:expt cl:/ cl:-
;;; then evaluate the partial derivatives at point (x, y)
;;; dz will always  be -1

(defmethod point-gradient-at ((self surfaceFunction) x y)
  (if (listp (sfunction-of self))
    (let*
       (
        (qList (second (cl-to-q-functions '(cl:* cl:+ cl:expt cl:/ cl:-) (sfunction-of
self))))
        ;; dx
        (dxQList (q:deriv qList :wrt 'x))
        (dxCLList (second (q-to-cl-functions '(q:* q:+ q:expt q:/ q:-) dxQList)))
        ;; dy
        (dyQList (q:deriv qList :wrt 'y))
        (dyCLList (second (q-to-cl-functions '(q:* q:+ q:expt q:/  q:-) dyQList)))
        )
       (setf (dx-of self) (funcall (eval `(lambda (x y) ,dxCLList)) x y))
       (setf (dy-of self) (funcall (eval `(lambda (x y) ,dyCLList)) x y))
       (setf (dz-of self) -1)
       )
    (error "The sfunction of ~s is not a list" self)
    )
  )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define class glDataPoint
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(defclass glDataPoint()
   (
    ( x :accessor x-of :initform nil :initarg :x :type double :documentation "x
coordinates")
    ( y :accessor y-of :initform nil :initarg :y :type double :documentation "y
coordinates")
    ( z :accessor z-of :initform nil :initarg :z :type double :documentation "z
coordinates")
    ( color :accessor color-of :initform nil :initarg :color :type double
:documentation "color")
    ( normal :accessor normal-of :initform nil :initarg :normal :type normal
:documentation "")
   )
   (:documentation "super class")
)

(defmethod initialize-instance ((self glDataPoint) &rest initargs)
  (declare (ignore initargs))
)

(defmethod assign-value ((self glDataPoint) xValue yValue zValue colorValue
normal)
  (setf (x-of self) xValue)
  (setf (y-of self) yValue)
  (setf (z-of self) zValue)
  (setf (color-of self) colorValue)
  (setf (normal-of self) normal)
)

;;; add two points' coordinates together
(defmethod add ((Self glDataPoint) (dp glDataPoint))
  (setf (x-of Self) (+ (x-of Self) (x-of dp)))
  (setf (y-of Self) (+ (y-of Self) (y-of dp)))
  (setf (z-of Self) (+ (z-of Self) (z-of dp)))
  (setf (color-of Self) (+ (color-of Self) (color-of dp)))
  (setf (normal-of Self) (+ (normal-of Self) (normal-of dp)))
)

;;; divide a point's coordinate by a non zero constant
(defmethod divide ((self glDataPoint) (value number))
  (if (/= value 0)
    (let
      ()
      (setf (x-of Self) (/ (x-of Self) value))
      (setf (y-of Self) (/ (y-of Self) value))
      (setf (z-of Self) (/ (z-of Self) value))
      (setf (color-of Self) (/ (color-of Self) value))
    )
    (error "The divide of ~s by zero" self)
  )
)

;;; compare two points' coordinate equality
(defmethod equal-coord-p ((self glDataPoint) (dp glDataPoint))
   (and (= (x-of Self) (x-of dp) )
        (= (y-of Self) (y-of dp) )
        (= (z-of Self) (z-of dp) )
        )
   )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define class glDataSet
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defclass glDataSet()
   (
    ( x-range :accessor x-range-of :initform (make-Range :min 0.0 :max 0.0)
:initarg :x-range :type Range :documentation "x axis range")
    ( y-range :accessor y-range-of :initform (make-Range :min 0.0 :max 0.0)
:initarg :y-range :type Range :documentation "y axis range")
    ( z-range :accessor z-range-of :initform nil :initarg :z-range :type Range
:documentation "z axis range")
    ( color-range :accessor color-range-of :initform nil :initarg :color-range :type
Range :documentation "color range")
```

```
   ( x-sequence :accessor x-sequence-of :initform nil :initarg :x-sequence
:documentation "x Sequence, could be list or vector or array")
   ( y-sequence :accessor y-sequence-of :initform nil :initarg :y-sequence
:documentation "y Sequence could be list or vector or array")
   ( x-sequence-type :accessor x-sequence-type-of :initform nil :initarg :x-
sequence-type   :documentation "x Sequence type, could be list or vector or
array")
   ( y-sequence-type :accessor y-sequence-type-of :initform nil :initarg :y-
sequence-type :documentation "y Sequence type, could be list or vector or array")
   ( surface-function :accessor surface-function-of :initform (make-instance
'surfaceFunction ) :initarg :surface-function :documentation "surface Function
either cl function or list")
   ( surface-heights :accessor surface-heights-of :initarg :surface-heights :initform
NIL :documentation  "The heights of the surface at each (x,y) location in the grid.
The locations must be arranged in a vector varying the y coordinate fastest.")
   ( x-data-size :accessor x-data-size-of :initform 0 :initarg :x-data-size
:documentation "the size of x sequence ")
   ( y-data-size :accessor y-data-size-of :initform 0 :initarg :y-data-size
:documentation "the size of y sequence ")
   ( data-points :accessor data-points-of :initform (make-hash-table) :initarg :data-
points :documentation "all 3 dimension data points stored in a hash table")
   ( min-z-value :accessor min-z-value-of :initform 0 :initarg :min-z-value
:documentation "min z value to determine the camaer position")
   ( max-z-value :accessor max-z-value-of :initform 0 :initarg :max-z-value
:documentation "max z value to determine the camaer position")
   ( cube-diagonal-length :accessor cube-diagonal-length-of :initform 0 :initarg
:cube-diagonal-length :documentation "")
   ( wireframe? :accessor wireframep :initform T :initarg :wireframe?
:documentation "switch on off wireframe, default is on")
   ( surface? :accessor surfacep :initform T :initarg :surface? :documentation
"switch on off surface, default is no surface")
   ( hiddenLines? :accessor hidden-Lines-p :initform T :initarg :hiddenLines?
:documentation "switch on off iHiddenLines, default is on")
  )
  (:documentation "class")
)
```

```
(defmethod toggle-surface ((self glDataSet))
  (if  (surfacep self)
    (setf (surfacep self) nil)
    (setf (surfacep self) T)
  )
)

(defmethod toggle-wireframe ((self glDataSet) )
  (if  (wireframep self)
    (setf (wireframep self) nil)
    (setf (wireframep self) T)
  )
)

(defmethod toggle-hidden-lines ((self glDataSet) )
  (if  (hidden-Lines-p self)
    (setf (hidden-Lines-p self) nil)
    (setf (hidden-Lines-p self) T)
  )
)

(defmethod initialize-instance ((self glDataSet) &rest initargs)
  (declare (ignore initargs))
   ;; create hash-table
   (setf (data-points-of Self) (make-hash-table))
   ;; create surface function class
   (setf (surface-function-of  Self) (make-instance 'surfaceFunction ))
   (setf (max-z-value-of self) 0)
   (setf (min-z-value-of self) 0)
   (setf (x-range-of self) (make-Range :min 0.0 :max 0.0))
   (setf (y-range-of self) (make-Range :min 0.0 :max 0.0))
   (setf (x-data-size-of self) 0)
   (setf (y-data-size-of self) 0)
   (setf (wireframep self) T)
   (setf (surfacep self) T)
   (setf (hidden-Lines-p self) T)
```

```
)


(defmethod set-surface-function ((self glDataSet) functionObject)
  (if (or (functionp functionObject) (listp functionObject))
    (set-sfunction (surface-function-of self) functionObject)
    (error "The  ~s is not a function" functionObject)
  )
)

;;; sequence will  be a list or a vector or a array
(defmethod set-sequence-type ((self glDataSet) axis sequence)
  (if (eq axis :x)
    (cond
     ((listp sequence) (setf (x-sequence-type-of self) :list))
     ((vectorp sequence) (setf (x-sequence-type-of self) :vector))
     ((arrayp sequence) (setf (x-sequence-type-of self) :array))
    )
    (cond
     ((listp sequence) (setf (y-sequence-type-of self) :list))
     ((vectorp sequence) (setf (y-sequence-type-of self) :vector))
     ((arrayp sequence) (setf (y-sequence-type-of self) :array))
   )
 )
)


;;; assinge values to the slot x-sequence, and identify the sequence type, and
compute the sequence size
(defmethod set-x-sequence ((self glDataSet) xsequence)
  ;; assign value
  (setf (x-sequence-of self) xsequence)

  ;; identify the sequence type list = l vector = v array = a
  (set-sequence-type self :x xsequence)

  ;; compute the size x-sequence
  (cond
   ((listp (x-sequence-of self)) (setf (x-data-size-of self) (list-length (x-sequence-of
self))))
   ((vectorp (x-sequence-of self) ) (setf (x-data-size-of self) (length (x-sequence-of
self))))
   ((arrayp (x-sequence-of self)) (setf (x-data-size-of self) (array-total-size (x-
sequence-of self))))
  )

)

;;; assinge values to the slot y-sequence, and identify the sequence type, and
compute the sequence size
(defmethod set-y-sequence ((self glDataSet) ysequence)
  ;; assign value
  (setf (y-sequence-of self) ysequence)

  ;; identify the sequence type list = list vector = vector array = array
  (set-sequence-type self :y ysequence)

  ;; compute the size y-sequence
  (cond
   ((listp (y-sequence-of  self)) (setf (y-data-size-of  self) (list-length (y-sequence-
of self))))
   ((vectorp (y-sequence-of  self) ) (setf (y-data-size-of self) (length (y-sequence-
of self))))
   ((arrayp (y-sequence-of  self)) (setf (y-data-size-of self) (array-total-size (y-
sequence-of self))))
  )
)

(defmethod sort-sequence ((self glDataSet))
;; sort x and y sequence
  (setf (x-sequence-of self) (sort (x-sequence-of self) #'<=))
  (setf (y-sequence-of self) (sort (y-sequence-of self) #'<=))

  ;; compute range of x and y
```

```
  (setf (x-range-of self) (make-Range :min (access-sequence self :x 0) :max
(access-sequence self :x (- (x-data-size-of self) 1) )) )
  (setf (y-range-of self) (make-Range :min (access-sequence self :y 0) :max
(access-sequence self :y (- (y-data-size-of self) 1) )) )
)


;;; access the ith element in the sequence based on the type of sequence
(defmethod access-sequence ((self glDataSet) axis iteration)
  (if (eq axis :x)
    (if (< iteration (x-data-size-of self))
      (cond
       ((eq (x-sequence-type-of  self) :list) (nth iteration (x-sequence-of self)))
       ((eq (x-sequence-type-of self) :vector) (svref (x-sequence-of self) iteration ))
       ((eq (x-sequence-type-of self) :array) (aref (x-sequence-of self) iteration ))
       )

       (error "out of boundary ~s " iteration)
      )
    (if (< iteration (y-data-size-of  self))
      (cond
       ((eq (y-sequence-type-of self) :list) (nth iteration (y-sequence-of self)))
       ((eq (y-sequence-type-of self) :vector) (svref (y-sequence-of self) iteration ))
       ((eq (y-sequence-type-of self) :array) (aref (y-sequence-of self) iteration ))
       )
       (error "out of boundary ~s " iteration)
      )
  )
)


;;; compute heights given a function, x sequence and y sequence
(defmethod compute-points-triples ((self glDataSet) &optional functionObject
xSequence ySequence)

  (if (not (eq xSequence Nil))
    (set-x-sequence self xSequence)
  )

  (if (not (eq ySequence Nil))
    (set-y-sequence self ySequence)
  )

  (if (not (eq functionObject Nil))
    (if (arrayp functionObject)
      (if (eq 1 (nth 0 (array-dimensions functionObject)))
        (let*
          (
           (array-dimension (list (x-data-size-of self) (y-data-size-of self)))
           ( data-array (make-array array-dimension))
           )

          (dotimes (ith (x-data-size-of self))
            (dotimes (jth (y-data-size-of self))

              (setf (aref data-array ith jth) (aref functionObject 0 (+ (* (x-data-size-of
self) ith) jth)))

              )
            )
          (setf (surface-heights-of self) data-array)
         )
        (setf (surface-heights-of self) functionObject)
        )
      (set-surface-function self functionObject)
      )
    )

  (sort-sequence self)

  (if (and (not (eq (surface-function-of self) Nil)) (not (eq (x-sequence-of self)
Nil)) (not (eq (y-sequence-of self) Nil)))
    (progn
      (dotimes (i (x-data-size-of self))
```

```
      (dotimes (j (y-data-size-of self))
        (let* (
               (x (access-sequence self :x i))
               (y (access-sequence self :y j))
               (dp (make-instance 'glDataPoint))
               zValue
               )
          ;; dpNormal = nil, color = 0
          (if (arrayp functionObject)
            (setf zValue (aref (surface-heights-of self) i j))
            (setf zValue (function-at (surface-function-of self) x y))
            )

          (assign-value dp x y zValue 0 Nil)

          (if (> zValue (max-z-value-of  self))
            (setf (max-z-value-of self) zValue)
            )
          (if (< zValue (min-z-value-of self))
            (setf (min-z-value-of self) zValue)
            )
          ;; put into hash table
          (setf (gethash (complex i j) (data-points-of self) ) dp)
          )
        )
      )
    (setf (z-range-of self) (make-Range :min (min-z-value-of self) :max (max-z-
value-of self)) )
    (setf (cube-diagonal-length-of self) (sqrt
                                          (+
                                           (expt (- (Range-max (z-range-of self)) (Range-min
(z-range-of self))) 2)
                                           (expt (- (Range-max (x-range-of self)) (Range-min
(x-range-of self))) 2)
                                           (expt (- (Range-max (y-range-of self)) (Range-min
(y-range-of self))) 2)
                                           )
                                          )
    )
  )
  (error "missing surface funcations or x sequence or y sequence")
 )
)


;;; compute a normal vector for triangle
(defmethod compute-triangle-normal ((self glDataSet) dp1 dp2 dp3)
 (let
  (
   (x1 (x-of dp1))
   (y1 (y-of dp1))
   (z1 (z-of dp1))
   (x2 (x-of dp2))
   (y2 (y-of dp2))
   (z2 (z-of dp2))
   (x3 (x-of dp3))
   (y3 (y-of dp3))
   (z3 (z-of dp3))
   (n (make-Normal :x 0 :y 0 :z 0))
  )
  (setf (Normal-x n) (- (* (- y2 y1) (- z3 z1)) (* (- z2 z1) (- y3 y1))))
  (setf (Normal-y n) (- (* (- z2 z1) (- x3 x1)) (* (- x2 x1) (- z3 z1))))
  (setf (Normal-z n) (- (* (- x2 x1) (- y3 y1)) (* (- y2 y1) (- x3 x1))))
  n
 )
)


;;; compute the normal vector for a point by averaging the adjacent triangles
sharing that point
;;; the four triangles will be
;;; ((i j), (i+1 j), (i j+1)), ((i j), (i-1 j), (i j+1)), ((i j), (i-1 j), (i j-1)), ((i j), (i j-1),
(i+1 j))
(defmethod compute-points-normal ((self glDataSet))
```

```
(if (/= (hash-table-size (data-points-of self)) 0)
        (let

        (
         (dp (make-instance 'glDataPoint))
        )
        (dotimes (i (x-data-size-of self))
          (dotimes (j (y-data-size-of self))
            (let* (
                  ;; assgin value
                  (normal1 (make-Normal :x 0 :y 0 :z 0))
                  (normal2 (make-Normal :x 0 :y 0 :z 0))
                  (normal3 (make-Normal :x 0 :y 0 :z 0))
                  (normal4 (make-Normal :x 0 :y 0 :z 0))
                  (normalPoint (make-Normal :x 0 :y 0 :z 0))
                  )
              (if (and (< i (- (x-data-size-of self) 1)) (< j (- (y-data-size-of self) 1)))
                (let
                  (
                   (dp1 (gethash (complex i j) (data-points-of self)))
                   (dp2 (gethash (complex (1+ i) j) (data-points-of self)))
                   (dp3 (gethash (complex i (1+ j) ) (data-points-of self)))
                  )
                   (setf normal1 (compute-triangle-normal self dp1 dp2 dp3))
                )
               )
              (if (and (< i 0) (< j (- (y-data-size-of self) 1)))
                (let
                  (
                   (dp1 (gethash (complex i j) (data-points-of self)))
                   (dp2 (gethash (complex (1- i) j) (data-points-of self)))
                   (dp3 (gethash (complex i (1+ j)) (data-points-of self)))
                   )
                  (setf normal2 (compute-triangle-normal self dp1 dp2 dp3))

                        )
              )
              (if (and (< i 0) (< j 0))
                (let
                  (
                   (dp1 (gethash (complex i j) (data-points-of self)))
                   (dp2 (gethash (complex (1- i) j) (data-points-of self)))
                   (dp3 (gethash (complex i (1- j)) (data-points-of self)))
                  )
                  (setf normal3 (compute-triangle-normal self dp1 dp2 dp3))

                      )
              )
              (if(and (< j 0) (< i (- (x-data-size-of self) 1)))
                (let
                  (
                   (dp1 (gethash (complex i j) (data-points-of self)))
                   (dp2 (gethash (complex i (1- j)) (data-points-of self)))
                   (dp3 (gethash (complex (1+ i) j) (data-points-of self)))
                  )
                 (setf normal4 (compute-triangle-normal self dp1 dp2 dp3))

                      )
               )
              (setf (Normal-x normalPoint)(/ (+ (Normal-x normal1) (Normal-x
normal2) (Normal-x normal3)(Normal-x normal4)) 4) )
              (setf (Normal-y normalPoint)(/ (+ (Normal-y normal1) (Normal-y
normal2) (Normal-y normal3)(Normal-y normal4)) 4) )
              (setf (Normal-z normalPoint)(/ (+ (Normal-z normal1) (Normal-z
normal2) (Normal-z normal3)(Normal-z normal4)) 4) )
              (setf dp (gethash (complex i j) (data-points-of self)))
              (setf (normal-of dp) normalPoint)
              )
            )
          )
      )
          (error "The size of ~s is zero" (data-points-of self))
  )
)
```

```
(defmethod draw ((self glDataSet))

  (if  (hidden-Lines-p self)
    ;; enable the surface visibility : hidden lines
      (glenable gl_depth_test)
  )

  (glClear (logior GL_COLOR_BUFFER_BIT gl_depth_buffer_bit))
  (glenable gl_normalize)

;;; wireframe
  (if   (wireframep self)
    (progn
      ;(glcolor3f 0.0 0.0 1.0)
      ;; axes
      (glenable gl_normalize)
      (with-rgba-vector v (0.0 0.0 1.0 1.0)
        (glmaterialfv gl_front_and_back gl_ambient_and_diffuse V)
      )
      (gllinewidth 1.2)
      (glBegin gl_lines)
      (glVertex3f -2.1s0 0.0s0 0.0s0)
      (glVertex3f 2.1s0 0.0s0 0.0s0)
      (glVertex3f 0.0s0 -2.1s0 0.0s0)
      (glVertex3f 0.0s0 2.1s0 0.0s0)
      (glVertex3f 0.0s0 0.0s0 0.0s0)
      (glVertex3f 0.0s0 0.0s0 1.0s0)
      (glEnd)

      ;;; set to wireframe
      (gllinewidth 1.0)
      (glpolygonmode gl_front_and_back gl_line)


      (with-rgba-vector V (0.9 0.0 0.0 0.5)
        (glmaterialfv gl_front_and_back gl_ambient_and_diffuse V)
      )
      (let
        (
         (dp (make-instance 'glDataPoint))
        )
        (dotimes (i (1- (x-data-size-of self)) )
          (dotimes (j (1- (y-data-size-of self)) )
              (progn
                (glBegin gl_polygon)
                (setf dp (gethash (complex i j) (data-points-of self)))
                (glVertex3f (x-of dp) (y-of dp) (z-of dp))
                (setf dp (gethash (complex (1+ i) j) (data-points-of self)))
                (glVertex3f (x-of dp) (y-of dp) (z-of dp))
                (setf dp (gethash (complex (1+ i) (1+ j) ) (data-points-of self)))
                (glVertex3f (x-of dp) (y-of dp) (z-of dp))
                (setf dp (gethash (complex  i (1+ j)) (data-points-of self)))
                (glVertex3f (x-of dp) (y-of dp) (z-of dp))
                (glEnd)
              )
            )
        )
      )
    )
  )

;;; surface rendering
  (if  (surfacep self)
    (progn

      (glpolygonmode gl_front_and_back gl_fill)
      (if  (hidden-Lines-p self)
        (let
          ()
          (glenable gl_polygon_offset_fill)
          (glpolygonoffset 1.0 1.0)
        )
```

```
    )

    ;;; here the surface color will be set to light green
    (with-rgba-vector V (0.0 0.9 0.0 0.0)
      (glmaterialfv gl_front_and_back gl_ambient_and_diffuse V)
      )
    (glcolor3f 0.0 0.9 0.0)
    (let
      (
       (dp (make-instance 'glDataPoint))
      )
      (dotimes (i (1- (x-data-size-of self)) )
        (dotimes (j (1- (y-data-size-of self)) )
          (progn
            (glBegin gl_polygon)

            (setf dp (gethash (complex i j) (data-points-of self)))
            (glnormal3f (Normal-x (normal-of dp)) (Normal-y (normal-of dp))
(Normal-z (normal-of dp)))
            (glVertex3f (x-of dp) (y-of dp) (z-of dp))

            (setf dp (gethash (complex (1+ i) j) (data-points-of self)))
            (glnormal3f (Normal-x (normal-of dp)) (Normal-y (normal-of dp))
(Normal-z (normal-of dp)))
            (glVertex3f (x-of dp) (y-of dp) (z-of dp))

            (setf dp (gethash (complex (1+ i) (1+ j) ) (data-points-of self)))
            (glnormal3f (Normal-x (normal-of dp)) (Normal-y (normal-of dp))
(Normal-z (normal-of dp)))
            (glVertex3f (x-of dp) (y-of dp) (z-of dp))

            (setf dp (gethash (complex  i (1+ j)) (data-points-of self)))
            (glnormal3f (Normal-x (normal-of dp)) (Normal-y (normal-of dp))
(Normal-z (normal-of dp)))
            (glVertex3f (x-of dp) (y-of dp) (z-of dp))
            (glEnd)
          )
        )
      )

    )
    (gldisable gl_polygon_offset_fill)
    (glPopAttrib )
   )
  )
)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Define class glSurfaceWindow
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defclass glSurfaceWindow (opengl-window)
  (
   (glds :accessor glds-of :initform nil :initarg :glds :documentation "all data
points")
   (camera-to-object-distance :accessor camera-to-object-distance-of :initform 0
:initarg :camera-to-object-distance :documentation "")
   (rotate-axis :accessor rotate-axis-of :initform :x :initarg :rotateAxis
:documentation "which axis the rotation around")
   (gain :accessor gain :initform 3 :initarg :gain :documentation "each time the
degree changed")
   (counterclockwise? :accessor counterclockwisep :initform T :initarg
:counterclockwise? :documentation "rotation direction, default is counter-
Clockwise")
  )
  (:documentation "")
)

(defmethod add-dataSet ((self glSurfaceWindow) ds)
 (setf (glds-of self) ds)
)
```

```
(defmethod init ((self glSurfaceWindow))

  ;;; set camera position

  (aim-camera (camera Self) :eye-x 0.0d0 :eye-y 0.0d0 :eye-z (+ 1 (cube-
diagonal-length-of (glds-of Self))))

  ;;; set background color white and alpha = 0.5
  (glclearcolor 1.0 1.0 1.0 0.5)

  ;;; display lines with color gradations
  (glShadeModel gl_smooth)

  ;;; define surface property (R G B Alpha) and use specular reflection
  (with-rgba-vector
    specularity (0.5 0.5 0.5 1.0)
    (glmaterialfv gl_front_and_back gl_specular specularity)
    )
  (glmaterialf gl_front_and_back gl_shininess 10.0) ; 0 - 128

  ;;; lights, here has two lights, one is local lights located at
  ;;; at world coordinate position ( ).
  ;;; the other is the ambient lights to give a general brightness
  ;;; and also set the diffuse and sepcular properties
  ;;; (0.5 0.5 0.5 1.0) is light white lights

  ;;; set up the global lights
  (with-rgba-vector
    globalAmbient (0.5 0.5 0.5 1.0)
    (gllightmodelfv gl_light_model_ambient globalAmbient)
    )

  ;;; set up the local lights
  (with-rgba-vector position (-5.0 5.0 5.0 1.0)
    (gllightfv gl_light1 gl_position position)
    )

  (with-rgba-vector diffuse  (-5.0 5.0 5.0 1.0)
    (gllightfv gl_light1 gl_diffuse diffuse)
    )

  ;;; turn on the lights
  (glenable gl_lighting)
  (glenable gl_light1)

  )

(defmethod display ((self glSurfaceWindow))
  (draw (glds-of self))
)

(defmethod set-rotate-axis ((self glSurfaceWindow) axis)
  (setf (rotate-axis-of self) axis)
)

(defmethod switch-direction ((self glSurfaceWindow) )

  (if  (counterclockwisep self)
    (setf (counterclockwisep self) nil)
    (setf (counterclockwisep self) T)
  )
)

;;; override the view-click-event-handler
(defmethod view-click-event-handler ((Self glSurfaceWindow) Where)
  (declare (ignore Where))
  (let (
        (axis (rotate-axis-of self))
        (xAxis 0)
        (yAxis 0)
        (zAxis 0)
        (gain (gain self))
        (counterClockWise (counterclockwisep self))
       )
```

```
    (cond
     ((eq  axis :x) (setf xAxis 1))
     ((eq  axis :y) (setf yAxis 1))
     ((eq  axis :z) (setf zAxis 1))
    )

    (if counterClockWise
      (setf gain (+ gain))
      (setf gain (- gain))
    )

    (glMatrixMode gl_modelview)
    (loop
      (unless (mouse-down-p) (return))
      (glrotatef gain xAxis yAxis zAxis)
      (view-draw-contents Self)
   )
  )
)

(defmethod toggle-surface ((self glSurfaceWindow))
  (toggle-surface (glds-of self))
  (display self)
  (glflush)
)

(defmethod toggle-wireframe ((self glSurfaceWindow) )
  (toggle-wireframe (glds-of self))
  (display self)
  (glflush)
)

(defmethod change-scale ((self glSurfaceWindow) scalar)

  (glMatrixMode gl_modelview)
  (glScalef
   scalar scalar scalar
   )
  (view-draw-contents Self)
)

(defmethod toggle-hidden-lines ((self glSurfaceWindow) )
  (toggle-hidden-lines (glds-of self))
  (display self)
  (glflush)
)

(defun gl-surface-plot (xSequence ySequence surfaceFunction)
   (let
     (
      (x xSequence)
      (y ySequence)
      (sf surfaceFunction)
      (ds2 (make-instance 'glDataSet))
      )
     (compute-points-triples ds2 sf x y)
     (compute-points-normal ds2)
     (make-instance 'glSurfaceWindow :glds ds2)
     )
   )
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;                        glSurfaceWindow-examples.lisp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; examples for glSurfaceWindow
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(in-package :ccl)

(defun loadfile (Path)
  (load
   (concatenate
    'string
    (directory-namestring ccl:*Loading-File-Source-File*)
    Path)
   )
)

(loadfile "Load OpenGL for MCL")
(use-package :gl )
(loadfile "glSurfaceWindow")

#| example:  arbitrary function

;;; A fancier function

(defun my-surface (x y)
    (- (* 3 (expt (- 1 x) 2)
          (exp (- (+ (expt x 2)
                     (expt (+ y 1) 2)))))
       (* 10 (- (/ x 5) (expt x 3) (expt y 5))
          (exp (- (+ (expt x 2) (expt y 2))))
          )
       (* 1/3 (exp (- (+ (expt (+ x 1) 2)
                         (expt y 2)))))))

(setf x (list -3.0 -2.8 -2.6 -2.4 -2.2 -2.0 -1.8 -1.6 -1.4 -1.2 -1.0 -0.8 -0.6 -0.4 -0.2
0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 2.0 2.2 2.4 2.6 2.8 3.0))
(setf y (vector -3.0 -2.8 -2.6 -2.4 -2.2 -2.0 -1.8 -1.6 -1.4 -1.2 -1.0 -0.8 -0.6 -0.4 -
0.2 0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 2.0 2.2 2.4 2.6 2.8 3.0))
(setf fw1 (gl-surface-plot x y #'my-surface))

;;; visual control
(toggle-surface fw1)
(toggle-wireframe fw1)

;;; rotation
(switch-direction fw1)
(setf (rotate-axis-of fw1) :x)
(setf (rotate-axis-of fw1) :y)
(setf (rotate-axis-of fw1) :z)

|#


#| known heights

(defun my-surface (x y)
  (- (* 3 (expt (- 1 x) 2)
        (exp (- (+ (expt x 2)
                   (expt (+ y 1) 2)))))
     (* 10 (- (/ x 5) (expt x 3) (expt y 5))
        (exp (- (+ (expt x 2) (expt y 2))))
        )
     (* 1/3 (exp (- (+ (expt (+ x 1) 2)
                       (expt y 2)))))))

(setf x (list -3.0 -2.8 -2.6 -2.4 -2.2 -2.0 -1.8 -1.6 -1.4 -1.2 -1.0 -0.8 -0.6 -0.4 -0.2
0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 2.0 2.2 2.4 2.6 2.8 3.0))
(setf y (list -3.0 -2.8 -2.6 -2.4 -2.2 -2.0 -1.8 -1.6 -1.4 -1.2 -1.0 -0.8 -0.6 -0.4 -0.2
0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 2.0 2.2 2.4 2.6 2.8 3.0))

(defun gl-surface-plot-heights (xSequence ySequence)
```

```
 (let*
   (
    ( x xSequence)
    ( y ySequence)
    (array-dimension (list 1 (* (length xSequence) (length ySequence))))
    ( z (make-array array-dimension))
    )
;; collect heights
#|
   (dotimes (i (1- (list-length x)) )
     (dotimes (j (1- (list-length y)) )
       (setf (aref z i j) (+ (q::random-uniform :from -1 :to 1) (funcall #'my-surface
(nth i x) (nth j y)))
             )
       )
     )
|#
   (dotimes (i (1- (list-length x)) )
     (dotimes (j (1- (list-length y)) )
       (setf (aref z 0 (+ (* (list-length x) i ) j)) (+ (q::random-uniform :from -1 :to
1) (funcall #'my-surface (nth i x) (nth j y)))
             )
       )
     )
   ;;; give x sequence, y sequence and heights to plot

   (gl-surface-plot x y z)
   )
)

(setf fw2 (gl-surface-plot-heights x y))

;;; scale
(change-scale fw2 0.8)

;;; visual control
(toggle-surface fw2)
(toggle-wireframe fw2)

;;; rotation
(switch-direction fw2)
(setf (rotate-axis-of fw2) :x)
(setf (rotate-axis-of fw2) :y)
(setf (rotate-axis-of fw2) :z)
|#
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; general bivariate pdf
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defclass bivariate-pdf(surfaceFunction)
  (
   )
   (:documentation "super class for bivariate pdf")
)

(defgeneric compute-point-gradient(bivariate-pdf xValue yValue )
  (:documentation "")
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; bivariate normal function  given mu1, sigma1, mu2, sigma2, corr
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defclass bivariate-normal-pdf(bivariate-pdf)
   (
    ( mu1 :accessor mu1-of :initform nil :initarg :mu1 :type double :documentation
"")
    ( sigma1 :accessor sigma1-of :initform nil :initarg :sigma1 :type double
:documentation "")
    ( mu2 :accessor mu2-of :initform nil :initarg :mu2 :type double :documentation
"")
    ( sigma2 :accessor sigma2-of :initform nil :initarg :sigma2 :type double
:documentation "")
    ( corr :accessor corr-of :initform nil :initarg :corr :type double :documentation
"")
   )
   (:documentation "")
)

(defmethod init-bivariate-normal-pdf ((self bivariate-normal-pdf) mu1 sigma1
mu2 sigma2 corr)
  (if (and (> sigma1 0) (> sigma2 0))
    (progn
      (setf (mu1-of self) mu1)
      (setf (sigma1-of self) sigma1)
      (setf (mu2-of self) mu2)
      (setf (sigma2-of self) sigma2)
      (setf (corr-of self) corr)
    )
    (error "~s should be more than zero" sigma1)
  )
)

(defmethod initialize-instance ((self bivariate-normal-pdf) &rest initargs)
  (declare (ignore initargs))
)

(defmethod pdf-at ((self bivariate-normal-pdf) x y)
  (let* (
        (mu1 (mu1-of self))
        (sigma1 (sigma1-of self))
        (mu2 (mu2-of self))
        (sigma2 (sigma2-of self))
        (corr (corr-of self))
        (temp
              (+

               (-
                ;(/ (expt (- x mu1) 2) (expt sigma1 2))
                (expt (/ (- x mu1) sigma1) 2)
                (/ (* 2 corr (- x mu1) (- y mu2 ))
                   (* sigma1 sigma2  ) )
                )

               ;(/ (expt (- y mu2) 2) (expt sigma2 2))
                (expt (/ (- y mu2) sigma2) 2)
               )

         )
```

```
      )

    (*
     (/ 1 (* 2 pi sigma1 sigma2 (sqrt (- 1 (expt corr 2))))
     )

     (exp (- 0 (/ temp
                  (* 2 (- 1 (expt corr  2)))
                  )))
     )
   )
)

(defmethod compute-point-gradient ((self bivariate-normal-pdf) x y)
  (let* (
         (mu1 (mu1-of self))
         (sigma1 (sigma1-of self))
         (mu2 (mu2-of self))
         (sigma2 (sigma2-of self))
         (corr (corr-of self))
         (dx
              (*

               (pdf-at self x y)

               (- 0
                  (*
                   (/ 1 (* 2 (- 1 (expt corr  2))))
                   (- (* 2 (- x mu1) (/ 1 (expt sigma1 2)))
                      (* 2 corr (- y mu2) (/ 1 (* sigma1 sigma2)))
                      )
                   )
                  )
               )
         )

         (dy
                  (*

                   (pdf-at self x y)

                   (- 0
                      (*
                       (/ 1 (* 2 (- 1 (expt corr 2))))
                       (- (* 2 (- y mu2) (/ 1 (expt sigma2 2)))
                          (* 2 corr (- x mu1) (/ 1 (* sigma1 sigma2)))
                       )
                      )
                   )
                  )
          )

         (dz -1)
        )
    (setf (dx-of self) dx)
    (setf (dy-of self) dy)
    (setf (dz-of self) dz)

  )

  (vector (dx-of self) (dy-of self) (dz-of self))
)


;;; assemble data points into a hash table by given a function and x range and y
range
(defmethod collect-function-points ((self glDataSet) functionObject size xRange
yRange)

  (if ( and (typep size 'Range) (typep xRange 'Range) (typep yRange 'Range))
    (let*
      (

       (stepX (/ (- (range-max (x-range-of self)) (range-min (x-range-of self))) (-
```

```
(x-data-size-of self) 1) ))
      (stepY (/ (- (range-max (y-range-of self)) (range-min (y-range-of self))) (-
(y-data-size-of self) 1)))
      (startX (range-min (x-range-of self)) )
      (startY (range-min (x-range-of self)) )
      )

     (setf (x-data-size-of self) (range-min size))
     (setf (y-data-size-of self) (range-max size))
     (setf (x-range-of self) xRange)
     (setf (y-range-of self) yRange)
     (dotimes (i (x-data-size-of  self))
      (dotimes (j (x-data-size-of  self))
       (let* (
            ;;; assgin value
            (x (+ startX (* i stepX)))
            (y (+ startY (* j stepY)))
            (dp (make-instance 'glDataPoint))
            (v (compute-point-gradient functionObject x y ))
            (dpNormal
             (make-Normal :x (svref v 0) :y (svref v 1) :z (svref v 2)))
            zValue
            )
        (setf zValue (pdf-at functionObject x y))
        (assign-value dp x y zValue 0 dpNormal)
        (if (> zValue (max-z-value-of self))
          (setf (max-z-value-of self) zValue)
          )
        (if (< zValue (min-z-value-of  self))
          (setf (min-z-value-of  self) zValue)
          )
        (setf (gethash (complex i j) (data-points-of self) ) dp)

        )
       )
      )
          )
      )



#| example bivariate pdf
(setf bbb (make-instance 'bivariate-normal-pdf))
(init-bivariate-normal-pdf bbb -0.2 0.5 -0.2 0.4 0.5)
(setf functionObject bbb)

(setf ds (make-instance 'glDataSet))
(setf size (make-Range :min 61 :max 61))
(setf xrange (make-Range :min -1.5 :max 1.0) )
(setf yrange (make-Range :min -1.5 :max 1.0) )
(collect-function-points ds functionObject size xRange yRange)

(setf fw (make-instance 'glSurfaceWindow :glds ds))

;;; visual control
(toggle-surface fw)
(toggle-wireframe fw)

;;; rotation
(switch-direction fw)
(setf (rotate-axis-of fw) :x)
(setf (rotate-axis-of fw) :y)
(setf (rotate-axis-of fw) :z)
|#
```