การสร้างภาพสามมิติโดยใช้ OpenGL

3D Rendering with OpenGL

อ.วิทวัส วิทยชำนาญกุล (kwwithaw@kmitl.ac.th)

จุดประสงค์

1. เพื่อให้รู้จักกระบวนการสร้างภาพสามมิติซึ่งประกอบด้วย การสร้างวัตถุจากทรงเรขาคณิตพื้นฐาน การวางตำแหน่งของวัตถุ, แสง, และกล้อง การกำหนดคุณสมบัติพื้นผิววัตถุ การคำนวณหา Normal Vector ของพื้นผิว การกำหนดโมเดลของแหล่งกำเนิดแสง และการฉายภาพลงบนระนาบสองมิติ
2. เพื่อให้รู้จักการใช้คำสั่งในไลบรารี OpenGL ช่วยในการสร้างภาพสามมิติ

OpenGL คืออะไร

OpenGL เป็นซอร์ฟแวร์ไลบรารีที่ใช้ติดต่อกับฮาร์ดแวร์แสดงผล โดยมีคำสั่งพื้นฐานประมาณ 120 คำสั่งที่สามารถใช้กำหนดคุณลักษณะและควบคุมการทำงานของแอ๊พพลิเคชันสามมิติ ผู้พัฒนาโปรแกรมสามารถใช้ไลบรารี OpenGL ได้โดยไม่มีค่าลิขสิทธิ์

OpenGL ถูกออกแบบมาโดยไม่ยึดติดกับระบบ สามารถทำงานได้บนทุกๆแพล็ตฟอร์ม (Portability) เพื่อที่จะบรรลุเป้าหมายนี้ OpenGL จะไม่มีคำสั่งที่จัดการกับระบบปฏิบัติการเลย อีกทั้งยังไม่มีคำสั่งเพื่อรับอินพุทจากผู้ใช้อีกด้วย หน้าที่ทั้งสองอย่างนี้จึงตกอยู่กับผู้เขียนโปรแกรม อย่างไรก็ตามยังมีซอร์ฟแวร์เพิ่มเติมที่ช่วยจัดการงานทั้งสองนี้หากพัฒนาโปรแกรมบนระบบปฏิบัติการแบบ Windows (ดู GLUT: OpenGL Utility Toolkit) นอกจากนี้ OpenGL ยังไม่มีคำสั่งระดับสูงที่จะใช้วาดวัตถุสามมิติแบบซับซ้อนอย่างเช่นรถยนต์ อวัยวะ หรือโมเลกุล สิ่งที่ OpenGL เตรียมไว้ให้สำหรับสร้างรูปจำลองสามมิติคือรูปทรงเรขาคณิตอย่างง่ายได้แก่ จุด เส้น และรูปหลายเหลี่ยมซึ่งผู้ใช้งานจะต้องนำรูปทรงเหล่านี้มาประกอบกันเพื่อให้เกิดรูปทรงสามมิติที่ซับซ้อน (ไลบรารี Open Inventor ถูกสร้างขึ้นจากชุดคำสั่งของ OpenGL เพื่อช่วยให้ผู้ใช้สามารถกำหนดสร้างรูปทรงที่ซับซ้อนได้โดยง่าย)

คณะกรรมการที่ร่วมกันกำหนดมาตรฐานของ OpenGL หรือ OpenGL Architecture Review Board (ARB) ประกอบด้วยบริษัท 3DLabs, Apple, ATI, Dell, Evans & Sutherland, HP, IBM, Intel, Matrox, NVIDIA, SGI, และ Sun Microsystems ปัจจุบัน OpenGL ได้ถูกพัฒนามาจนถึงเวอร์ชัน 1.5

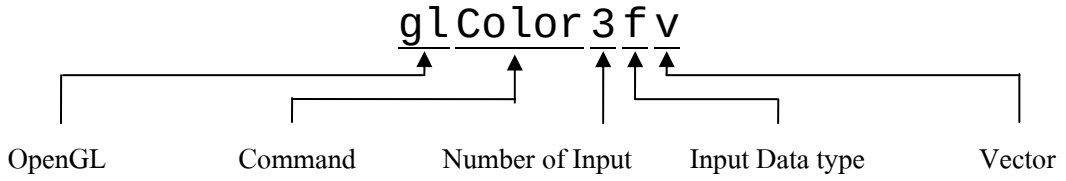
OpenGL vs. DirectX [1]

DirectX เป็นชุดพัฒนาซอฟต์แวร์เกมของไมโครซอฟท์ซึ่งสนับสนุนการทำงานหลายด้านทั้งภาพกราฟฟิกสองมิติ สามมิติ เสียง อุปกรณ์อินพุท และการเล่นพร้อมกันหลายคน DirectX มีการเปลี่ยนแปลงอย่างมากทุกครั้งที่มีการออกเวอร์ชันใหม่แต่ยังคงรองรับการทำงานของคำสั่งในเวอร์ชันเก่าด้วย อย่างไรก็ตามมันทำงานบนระบบปฏิบัติการของไมโครซอฟท์และ

เครื่องเล่น Xbox เท่านั้น สำหรับผู้เริ่มต้นแล้วมันค่อนข้างยากที่จะเรียนรู้ เกมสามมิติส่วนใหญ่เขียนขึ้นจาก DirectX

OpenGL ถูกควบคุมโดยคณะกรรมการ OpenGL Architecture Review Board ตัวไลบรารีของ OpenGL ต่างจาก DirectX เนื่องจากมันสนับสนุนการทำงานเฉพาะกราฟฟิกสองมิติและสามมิติ ผู้เขียนเกมต้องใช้ไลบรารีตัวอื่นช่วยเพื่อทำให้เกมสมบูรณ์ ทำให้มีเกมเพียงไม่มากที่เขียนด้วย OpenGL อย่างเช่น Quake, DOOM, Half Life, Unreal และ Call of Duty แต่ชุดคำสั่งของ OpenGL สามารถทำงานได้บนทุกแพลตฟอร์ม การทำงานค่อนข้างเสถียรและการเรียนรู้ค่อนข้างง่ายโดยมีหนังสือประกอบมากมาย สำหรับงานด้านวิจัยและพัฒนาแล้ว OpenGL ได้รับความนิยมอย่างสูง

คำสั่งและชนิดของข้อมูลของ OpenGL



การใช้งานคำสั่ง OpenGL จะต้อง include ไฟล์ header ชื่อ gl.h คำสั่งของ OpenGL จะขึ้นต้นด้วย `gl` อย่างเช่น `glClearColor()` ค่าคงที่จะขึ้นต้นด้วย `GL_` เช่น `GL_COLOR_BUFFER_BIT` สำหรับตัวลงท้ายหรือ suffix ของบางคำสั่งจะประกอบด้วยตัวเลขและตัวอักษรเช่น `glColor3f` ซึ่งเลข 3 บ่งบอกถึงจำนวนตัวแปรอินพุทคือสี RGB หากเป็น `glColor4f` จะหมายถึง RGBA ส่วน `f` หมายถึงอินพุทจะต้องเป็นข้อมูลชนิดเลขทศนิยม บางคำสั่งอาจรับข้อมูลอินพุทที่แตกต่างกันได้ถึง 8 ชนิด ตารางที่ 1 แสดง suffix ที่ใช้กำหนดชนิดของข้อมูลเทียบกับ ANSI C และ OpenGL Type Definition

ตารางที่ 1 Suffix ของคำสั่งและชนิดของข้อมูล

Suffix	Data Type	ANSI C Type Def.	OpenGL Type Def.
b	8-bit integer	signed char	GLbyte
s	16-bit integer	short	GLshort
i	32-bit integer	long	Glint, GLsizei
f	32-bit floating-point	float	GLfloat, GLclampf
d	64-bit floating-point	double	GLdouble, GLclampd
ub	8-bit unsigned integer	unsigned char	GLubyte, GLboolean
us	16- bit unsigned integer	unsigned short	GLushort
ui	32- bit unsigned integer	unsigned long	GLuint, GLenum
-	-	void	GLvoid

ดังนั้นคำสั่ง `glVertex2i(1, 3)` และ `glVertex2f(1.0, 3.0)` จึงทำหน้าที่เหมือนกันต่างกันที่คำสั่งแรกจะต้องกำหนดข้อมูลอินพุทเป็นเลขจำนวนเต็มและคำสั่งหลังกำหนดเป็นเลขทศนิยม

คำสั่งบางคำสั่งใน OpenGL มี suffix ตัวสุดท้ายเป็น `v` ซึ่งหมายความว่าคำสั่งนั้นรับข้อมูลอินพุทเป็นพอยน์เตอร์ที่ชี้ไปยังอาร์เรย์ของค่าแทนที่จะเป็นตัวเลข ตัวอย่างคำสั่งที่เหมือนกันแต่รับอินพุทต่างกัน

```
glColor3f(1.0, 0.0, 0.0);
```

```
float color_array[] = {1.0, 0.0, 0.0};
glColor3fv(color_array);
```

การทำงานของ OpenGL จะเป็นลักษณะ State Machine คือเมื่อสั่งให้โปรแกรมอยู่ในสถานะใด สถานะนั้นจะคงอยู่ตลอดจนกว่าจะมีการเปลี่ยนแปลง อย่างเช่นการตั้งค่าสีซึ่งเป็น State Variable เมื่อตั้งค่าสีไปครั้งหนึ่ง วัตถุที่ถูกวาดหลังจากนั้นจะมีสีนั้นตลอดจนกว่าจะมีคำสั่งเปลี่ยนสี

สำหรับผู้ที่สนใจสามารถอ่านเพิ่มเติมเกี่ยวกับ OpenGL และการเรนเดอร์ภาพสามมิติอย่างละเอียดที่ [2] และหนังสืออ้างอิงคำสั่งของ OpenGL ที่ [3]

GLU: OpenGL Utility Library

GLU เป็นการเอาชุดคำสั่งของ OpenGL ที่ถูกใช้บ่อยๆมารวมกันเป็นคำสั่งใหม่ซึ่งทำให้การเขียนโปรแกรมสะดวกขึ้นอย่างเช่นการทำ Mipmapping สำหรับ Texture Mapping, การแปลงโคออดิเนทแบบ Orthogonal และ Perspective, การแบ่งซอยรูปเหลี่ยมหรือ Polygon Tessellation, การเรนเดอร์รูปทรงพื้นฐานเช่นทรงกลม ทรงกระบอก และแผ่นจาน, และการวาดเส้นโค้งและพื้นผิวโค้งโดยใช้ Non-Uniform Rational B-Spline (NURBS) คำสั่งของ GLU จะขึ้นต้นด้วยคำว่า `glu` เสมอ การเรียกใช้จะต้อง include ไฟล์ header ชื่อ glu.h ข้อมูลเพิ่มเติมเกี่ยวกับคำสั่งของ GLU ที่ [4]

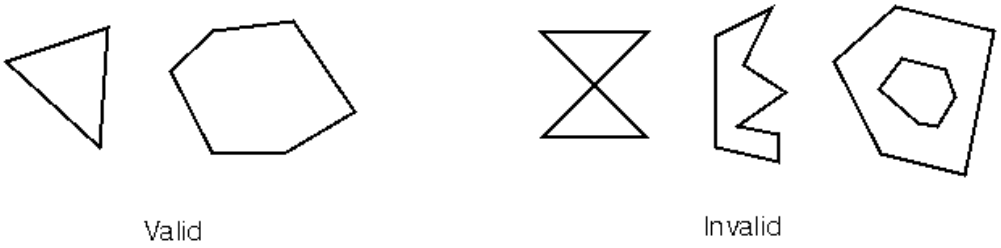
GLUT: OpenGL Utility Toolkit

GLUT เขียนขึ้นโดย Mark Kilgard เป็นเครื่องมือช่วยติดต่อระหว่างแอ๊พพลิเคชันของ OpenGL กับระบบปฏิบัติการแบบ Windows หรือเรียกว่าเป็น API (Application Programming Interface) ทำให้การเขียนโปรแกรมโดยใช้ OpenGL มีความซับซ้อนน้อยลง เหมาะสำหรับการพัฒนาโปรแกรมขนาดเล็กถึงขนาดกลาง คำสั่งของ GLUT จะขึ้นต้นด้วยคำว่า `glut` ค้นหาข้อมูลเพิ่มเติมเกี่ยวกับ GLUT ได้ที่ [5]

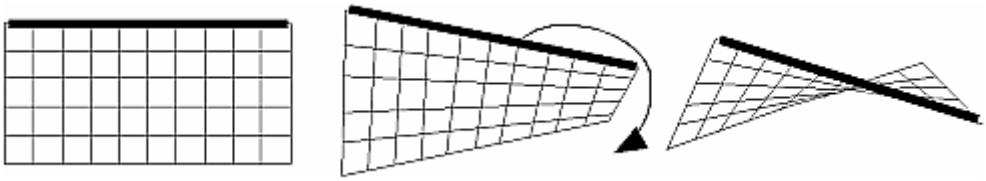
รูปทรงพื้นฐาน (Geometric Primitives)

รูปทรงพื้นฐานของ OpenGL ได้แก่ จุด, เส้น, และรูปเหลี่ยม โดยรูปทรงทั้งสามนี้ถูกกำหนดตำแหน่งโดยคู่ลำดับสามมิติ *(x, y, z)* หรือคู่ลำดับโฮโมจีเนียส *(x, y, z, w)* (โดยปกติแล้ว $w = 1$) ที่เรียกว่า vertex ซึ่งใช้ 1 vertex สำหรับจุด 2 vertex สำหรับเส้น และมากกว่า 2 vertex สำหรับรูปเหลี่ยม OpenGL ได้กำหนดลักษณะของรูปเหลี่ยมที่เหมาะสมโดยใช้ข้อบังคับสามข้อคือ ขอบของรูปเหลี่ยมจะต้องไม่ตัดกันเอง รูปทรงของรูปเหลี่ยมจะต้องไม่แหว่งเว้าเข้าไป และจะต้องไม่มีรูภายในรูปเหลี่ยม (ดูรูปที่ 1 ประกอบ) แต่ไม่มีข้อกำหนดสำหรับจำนวนมุมของรูปเหลี่ยม ข้อบังคับเหล่านี้ทำให้การแสดงผลมีความรวดเร็ว และหากใช้รูปเหลี่ยมที่ไม่เหมาะสมก็อาจทำให้การเรนเดอร์ภาพผิดพลาดได้ อย่างไรก็ตามรูปเหลี่ยมที่ไม่เหมาะสมสามารถแบ่งเป็นรูปเหลี่ยมย่อยๆได้โดยผู้ใช้หรือใช้ไลบรารี GLU ซึ่งมีคำสั่งที่ใช้ซอยรูปเหลี่ยมที่ซับซ้อนด้วย (Tessellation)

ข้อควรระวังสำหรับการเรนเดอร์รูปเหลี่ยมคือเมื่อ vertex ของรูปเหลี่ยมไม่ได้วางอยู่บนระนาบเดียวกัน การหมุนและการเปลี่ยนมุมมองอาจทำให้รูปเหลี่ยมนั้นกลายเป็นรูปเหลี่ยมที่ไม่เหมาะสมได้ ตัวอย่างในรูปที่ 2 เมื่อรูปสี่เหลี่ยมมี vertex ที่ไม่ได้อยู่บนระนาบเดียวกันโดยวางเหลื่อมกันเล็กน้อย เมื่อเกิดการหมุนจะทำให้รูปสี่เหลี่ยมนั้นมีลักษณะที่ไม่เหมาะสมคือเว้าเข้าไป ซึ่งจะทำให้การเรนเดอร์ผิดพลาด ปัญหานี้สามารถหลีกเลี่ยงได้หากใช้เฉพาะรูปสามเหลี่ยมเพราะจุดทั้งสามจะวางตัวอยู่ในระนาบเดียวกันแน่นอน



รูปที่ 1. รูปเหลี่ยมที่เหมาะสมและไม่เหมาะสม



รูปที่ 2. รูปเหลี่ยมที่ไม่ได้ระนาบเมื่อหมุนเปลี่ยนมุมมองจะทำให้เกิดส่วนเว้า

เส้นโค้งและพื้นผิวโค้งสำหรับ OpenGL แล้วไม่ถือว่าเป็นรูปทรงพื้นฐาน แต่สามารถประมาณได้โดยใช้เส้นตรงหรือพื้นผิวย่อยๆมาประกอบกัน ดังรูปที่ 3 และ 4

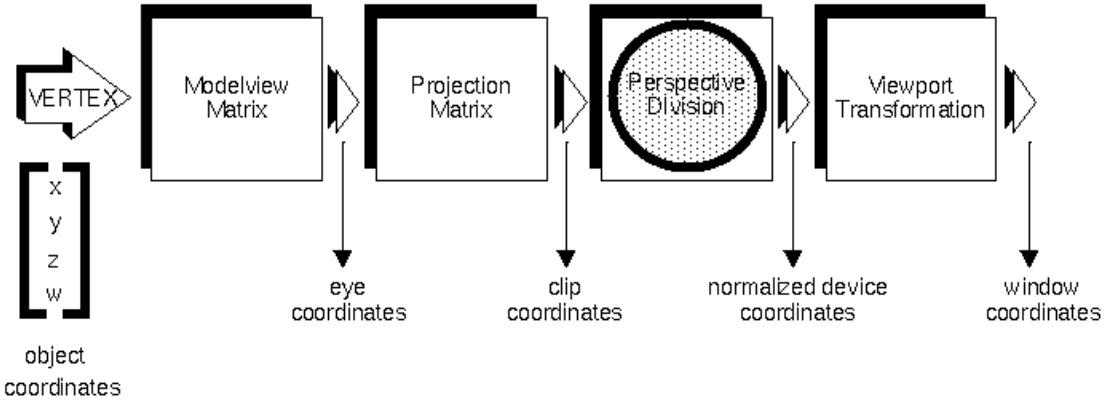
รูปที่ 3. เส้นโค้งที่เกิดจากการประกอบกันของเส้นตรง

รูปที่ 4. ทรงกลมที่เกิดจากการประกอบกันของพื้นผิวเรียบ

การจัดวางตำแหน่งวัตถุและการฉายภาพลงบนจอ

หากมี vertex v ซึ่งเป็นโคออดิเนท (x, y, z, w) และเมตริกการแปลง (Transformation Matrix) M ขนาด 4×4 การแปลงโคออดิเนทหรือ Transformation จะเขียนในรูปเมตริกได้คือ

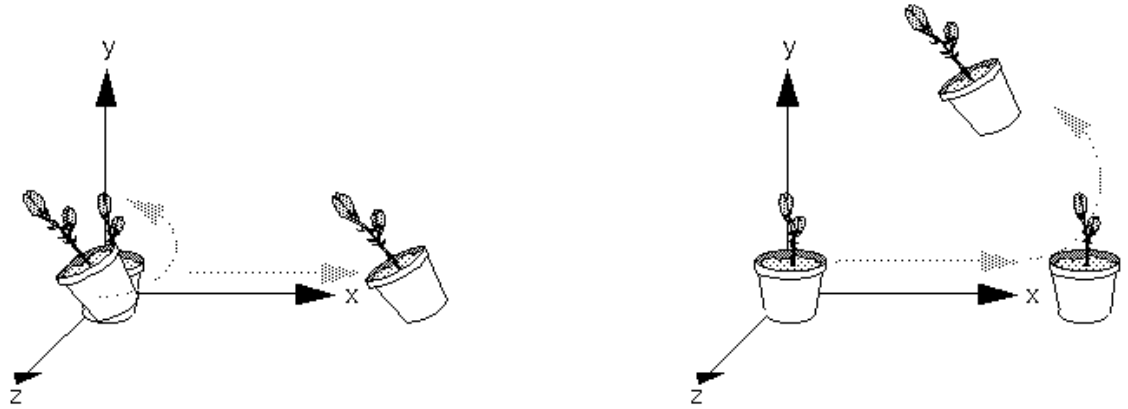
$$v' = Mv \tag{1}$$



รูปที่ 5. กระบวนการแปลงโคออดิเนทสามมิติของวัตถุไปเป็นพิกเซลบนจอภาพ

รูปที่ 5 แสดงกระบวนการแปลงโคออดิเนทสามมิติของวัตถุไปเป็นพิกเซลบนจอภาพโดยแต่ละขั้นตอนสามารถเทียบได้กับขั้นตอนการถ่ายภาพ ขั้นตอนทั้ง 4 สามารถอธิบายได้ดังนี้

- *Modelview Transformation* – รวมสองขั้นตอนคือ Viewing Transformation เป็นการเปลี่ยนมุมมองซึ่งเทียบได้กับการจัดตำแหน่งกล้องและ Modeling Transformation ประกอบด้วยการเลื่อน (Translation), การหมุน (Rotation), และการสเกลวัตถุ (Scaling) ซึ่งเทียบได้กับการจัดตำแหน่งวัตถุ การแปลงทั้งสองจะเปลี่ยน Object Coordinates เป็น Eye Coordinates



(a) (b)

รูปที่ 6. ลำดับที่แตกต่างกันเมื่อทำการหมุนและเลื่อนวัตถุ ให้ผลลัพธ์ที่แตกต่างกันมาก

การทำ Modelview Transformation นั้นค่อนข้างซับซ้อน สมมติว่ามีวัตถุวางอยู่บนจุดกำเนิด และต้องการหมุนวัตถุรอบแกน z ทวนเข็มนาฬิกาเป็นมุม 45 องศา ร่วมกับการเลื่อนวัตถุไปตามแกน $+x$ หากวัตถุถูก

หมุน 45 องศาก่อนที่จะเลื่อนไปตามแกน $+x$ วัตถุจะวางตัวอยู่บนแกน x (รูปที่ 6.a) แต่ถ้าหากเลื่อนวัตถุไปตามแกน $+x$ ก่อนที่จะหมุนวัตถุ วัตถุจะวางตัวอยู่บนเส้นตรง $y=x$ (รูปที่ 6.b) จะเห็นว่าลำดับของการ Transform มีความสำคัญมาก

- *Projection Transformation* – เป็นการกำหนดรูปร่างของปริมาตรการมองหรือ Viewing Volume เป็น Orthogonal หรือ Perspective และกำหนดขอบเขตของปริมาตรซึ่งมีผลทำให้บางส่วนของวัตถุถูกตัดออกจากฉาก เทียบได้กับการเลือกเลนส์กล้องและการปรับระยะวัตถุ ขั้นตอนนี้จะให้ Clip Coordinates
- *Perspective Division* – จะทำการหารค่าของโคออดิเนท (x, y, z, w) ด้วยค่าของ w ทำให้ได้ Normalized Device Coordinates
- *Viewport Transformation* – เป็นการเปลี่ยนโคออดิเนทของวัตถุไปเป็นโคออดิเนทของจอภาพซึ่งสามารถทำให้ภาพขยาย ยืดหรือหดได้ เทียบได้กับการปรับขนาดของภาพในขั้นตอนการอัดรูป ขั้นตอนนี้จะให้ Window Coordinates

หลังจากการแปลงโคออดิเนทจนได้ Window Coordinates แล้ว การเรนเดอร์ภาพลงบนหน้าจอจะใช้ทั้งโคออดิเนท (x, y, z) โดย (x, y) กำหนดจุดพิกเซลบนจอที่จะวาด และ z เป็นตัวเก็บค่าความลึกของ vertex โดยวัดระยะจากจอภาพ ค่าความลึกนี้มีประโยชน์ในการตัดส่วนที่ไม่จำเป็นออก เช่นถ้ามี vertex สองจุดที่มีค่าพิกัด x และ y เดียวกันแต่มีค่า z ต่างกัน ค่า z จะถูกใช้ในการเปรียบเทียบเพื่อตัดสินใจว่าพื้นผิวไหนถูกบังอยู่ และจะไม่วาดพื้นผิวส่วนนั้น โดยเรียกเทคนิคนี้ว่า Hidden-Surface Removal

แสง

แสงมีความสำคัญมาก ในการสร้างภาพสามมิติ วัตถุส่วนใหญ่จะไม่ดูเหมือนสามมิติจนกว่าจะมีการให้แสง ดังตัวอย่างทรงกลมในรูปที่ 7 จะเห็นว่าทรงกลมที่ยังไม่ให้แสงจะมีลักษณะเหมือนแผ่นจาน 2 มิติ

รูปที่ 7. ทรงกลมที่ไม่มีแสงตกกระทบและทรงกลมที่มีแสงตกกระทบ

เมื่อแสงตกกระทบวัตถุ ตาเราจะสามารถรับสีที่แตกต่างกันของวัตถุขึ้นอยู่กับพลังงานของโฟตอนที่ตกกระทบเซลล์ประสาทรูปกรวย โฟตอนเหล่านั้นมาจากแหล่งกำเนิดแสง บางส่วนของโฟตอนจะถูกดูดกลืนและส่วนที่เหลือจะถูกสะท้อนโดยพื้นผิวของวัตถุ พื้นผิวที่แตกต่างกันจะมีคุณสมบัติการสะท้อนแสงที่แตกต่างกัน พื้นผิวที่มันเงาจะสะท้อนแสงไปยังทิศทางที่เจาะจง พื้นผิวหยาบจะสะท้อนแสงให้กระจายไปในทุกทิศทาง แต่โดยปกติแล้วพื้นผิวส่วนใหญ่จะมีคุณสมบัติอยู่ระหว่างพื้นผิวสองแบบนี้

OpenGL จะคำนวณแสงโดยแยกแสงเป็น 3 องค์ประกอบคือแดง เขียวและน้ำเงิน (RGB) ดังนั้นสีของแสงจากแหล่งกำเนิดจะถูกกำหนดโดยปริมาณขององค์ประกอบเหล่านี้ และสีของพื้นผิววัตถุจะถูกกำหนดโดยเปอร์เซ็นต์ของแสงสีแดงเขียวและน้ำเงินที่ตกกระทบและถูกสะท้อนออกไป ตัวอย่างเช่นลูกบอลสีแดงจะดูดกลืนแสงสีเขียวและแสงสีน้ำเงินแต่จะสะท้อนแสงสีแดง เมื่ออยู่ในแสงขาวจะมองเห็นเป็นสีแดง แต่ถ้าอยู่ในแสงสีเขียวมันจะกลายเป็นสีดำ สมการคำนวณแสงของ OpenGL นั้นเป็นแค่การประมาณการแต่ให้ผลลัพธ์ที่ดีในเวลาคำนวณที่สั้น

แหล่งกำเนิดแสงใน OpenGL สามารถกำหนดให้มาได้จากหลายทิศทาง โดยสามารถกำหนดให้มาจากทิศทางที่แน่นอนได้หรือจะให้เป็นแสงที่สะท้อนไปมารอบๆฉาก ตัวอย่างเช่นเมื่อเราเปิดหลอดไฟในห้อง แสงส่วนใหญ่จะมาจากหลอดไฟนั้นโดยตรง แต่แสงบางส่วนจะสะท้อนไปมาในผนังห้องหลายรอบก่อนที่จะถึงตาเรา แสงที่สะท้อนไปมาในห้องนี้เรียกว่า Ambient Light ซึ่งถูกสมมติว่าไม่ทราบแหล่งกำเนิดที่แน่นอนแต่จะหายไปเมื่อเราปิดหลอดไฟนั้น

โมเดลของแหล่งกำเนิดแสงใน OpenGL จะถูกแบ่งเป็น 4 แบบใหญ่คือ Emitted, Ambient, Diffuse, และ Specular Light โดยโมเดลทั้ง 4 นี้จะถูกกำหนดองค์ประกอบสี RGB ที่แตกต่างกัน และถูกคำนวณแยกกันแล้วนำมารวมกันภายหลัง

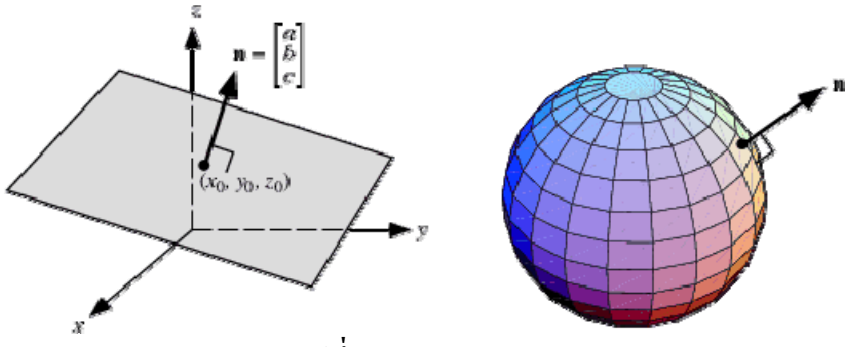
- *Emitted Light* – เป็นแสงที่ส่งออกมาจากตัววัตถุเอง
- *Ambient Light* – เป็นแสงที่กระจัดกระจายจากการสะท้อนจนไม่รู้แหล่งที่มา ทำให้ดูเหมือนกับมาจากทุกทิศทาง เมื่อแสงแบบนี้ตกกระทบวัตถุจะสะท้อนเท่าๆกันในทุกทิศทาง แสงจากหลอดไฟแบ็คไลท์ในห้องจะมีองค์ประกอบนี้อยู่มากเนื่องจากถูกสะท้อนหลายครั้งก่อนที่จะมาถึงตา ส่วนแสงจากสปอทไลท์จะมีองค์ประกอบนี้น้อยมากเนื่องจากแสงจะพุ่งไปในทิศทางเดียว
- *Diffuse Light* – เป็นแสงที่มาจากแหล่งกำเนิดโดยตรงแต่เมื่อตกกระทบวัตถุจะสะท้อนออกไปเท่ากันทุกทิศทาง
- *Specular Light* – มาจากแหล่งกำเนิดที่เจาะจงเช่นกันแต่จะสะท้อนไปในทิศทางที่เจาะจง คือมุมสะท้อนเท่ากับมุมตกกระทบ แสงจากเลเซอร์ที่ตกกระทบกระจกที่มีคุณภาพจะให้การสะท้อนเป็น Specular เกือบ 100 เปอร์เซ็นต์ วัตถุที่มันเงาเช่นพลาสติกจะมีคุณสมบัติของ Specular มากกว่าวัตถุที่หยาบเช่นชอล์ค

วัตถุมีโมเดลการสะท้อนแสงสามแบบคือ Ambient, Diffuse, และ Specular Reflectance ที่ถูกกำหนดสีไว้แตกต่างกัน Ambient Reflectance ของวัตถุจะใช้สำหรับคำนวณสีของวัตถุที่แสดงออกมาเมื่อถูก Ambient Light ตกกระทบ เช่นเดียวกับ Diffuse และ Specular Reflectance ซึ่งคำนวณคู่กับ Diffuse และ Specular Light ตามลำดับ โดยปกติแล้วสีของวัตถุที่ถูกกำหนดโดย Ambient และ Diffuse Reflectance จะเหมือนหรือคล้ายคลึงกัน ส่วน Specular Reflectance จะกำหนดเป็นสีขาวหรือเทา ซึ่งทำให้จุดที่สะท้อนเป็นสีของแหล่งกำเนิดแสง อย่างเช่นลูกบอลมันเงาสีแดงเมื่อถูกฉายด้วยแสงสีขาวพื้นที่ส่วนใหญ่จะเป็นสีแดงแต่จะมีส่วนหนึ่งที่มีมุมตกกระทบเท่ากับมุมสะท้อนซึ่งจะเป็นสีขาว

การกำหนดปริมาณสีสำหรับแสงจะกำหนดเป็นเปอร์เซ็นต์ของความเข้มสูงสุดของแต่ละสี ถ้าค่า RGB เป็น 1.0 ทั้งหมดจะได้แสงสีขาว ถ้าเป็น 0.5 ทั้งหมดจะได้สีขาวที่ความเข้มต่ำลงครึ่งหนึ่ง แต่ถ้า R=G=1.0, B=0.0 จะเป็นแสงสีเหลือง ส่วนการกำหนดปริมาณสีที่สะท้อนออกมาจากวัตถุจะกำหนดเป็นสัดส่วนของสีที่สะท้อนออกมา อย่างเช่นถ้า R=1.0, G=0.5, B=0.0 วัตถุจะสะท้อนแสงสีแดงทั้งหมด สะท้อนแสงสีเขียวเพียงครึ่งหนึ่งของแสงสีเขียวที่ตกกระทบและดูดกลืนแสงสีน้ำเงินทั้งหมด สมมติว่าแสงมีค่าความเข้มเป็น (LR, LG, LB) และวัตถุมีค่าการสะท้อนแสงเป็น (MR, MG, MB) แสงที่เข้าสู่ตาจะเป็น (LR*MR, LG*MG, LB*MB) ถ้าหากมีแสงจากสองแหล่งเข้าสู่ตาเป็น (R1, G1, B1) และ (R2, G2, B2) แสงรวมจะเป็น (R1+R2, G1+G2, B1+B2) หากค่าที่รวมได้มีค่ามากกว่า 1.0 ซึ่งสว่างเกินกว่าที่จอภาพจะแสดงผลได้ ค่านั้นจะถูกปัดลงเป็น 1.0

Normal Vector

Normal Vector เป็นเวคเตอร์ที่ชี้ไปในทิศทางตั้งฉากกับพื้นผิวของวัตถุ มีไว้สำหรับกำหนดลักษณะการวางตัวของพื้นผิว มันถูกใช้ในการคำนวณปริมาณแสงที่ตกกระทบพื้นผิวบริเวณนั้น OpenGL อนุญาตให้มีการกำหนด Normal Vector เฉพาะที่จุด Vertex ของพื้นผิวเท่านั้น

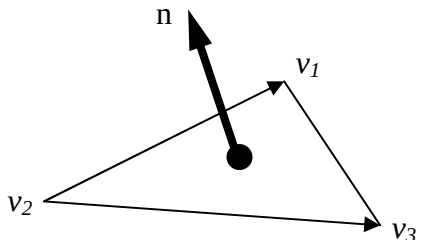


รูปที่ 8. Normal Vector

หากมีสมการทางคณิตศาสตร์ที่ใช้อธิบายพื้นผิว เราสามารถหาค่า Normal Vector ที่แต่ละจุดบนพื้นผิวได้ สมมติสมการพื้นผิวเป็น $z = f(x, y)$ จะหา Normal Vector ที่จุด (x_0, y_0) ได้จาก

$$n = \begin{bmatrix} f_x(x_0, y_0) \\ f_y(x_0, y_0) \\ -1 \end{bmatrix} \tag{2}$$

เมื่อ $f_x = \partial f/\partial x$ และ $f_y = \partial f/\partial y$ เป็นอนุพันธ์ย่อยของฟังก์ชัน $f(x, y)$ [6]



รูปที่ 9. Normal Vector ของรูปสามเหลี่ยมที่มีจุดยอดที่ v_1, v_2, และ v_3

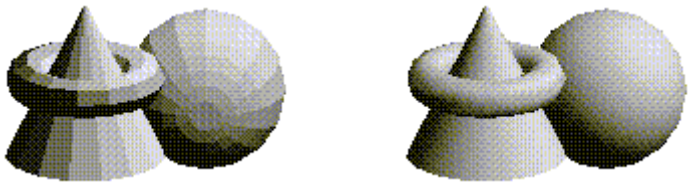
แต่หากพื้นผิวประกอบด้วยรูปเหลี่ยมและไม่มีสมการอธิบายแล้วจะมีสูตรการคำนวณ Normal Vector คือ (ดูรูปที่ 9 ประกอบ)

$$n=[v_1-v_2]\times[v_2-v_3]=\begin{bmatrix}x_1-x_2\\y_1-y_2\\z_1-z_2\end{bmatrix}\times\begin{bmatrix}x_2-x_3\\y_2-y_3\\z_2-z_3\end{bmatrix}=\begin{bmatrix}(y_1-y_2)(z_2-z_3)-(z_1-z_2)(y_2-y_3)\\(z_1-z_2)(x_2-x_3)-(x_1-x_2)(z_2-z_3)\\(x_1-x_2)(y_2-y_3)-(y_1-y_2)(x_2-x_3)\end{bmatrix} \quad (3)$$

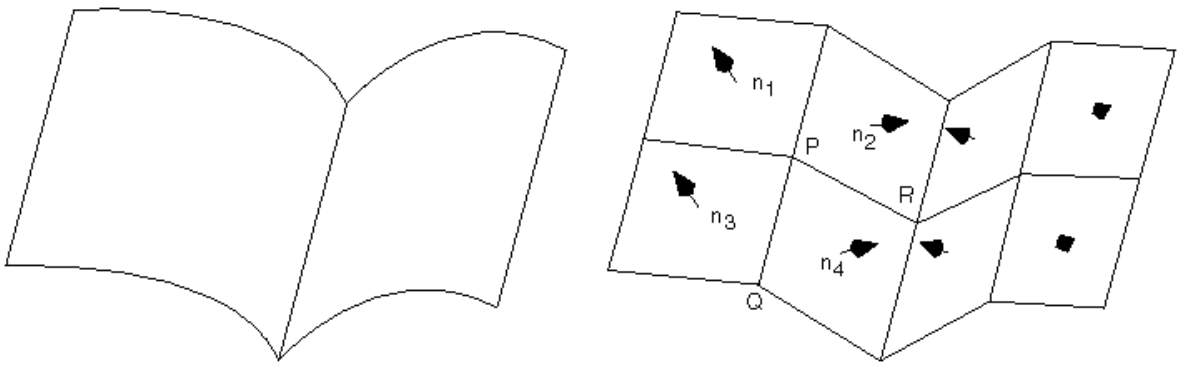
เมื่อ $v_1=(x_1, y_1, z_1)$, $v_2=(x_2, y_2, z_2)$, และ $v_3=(x_3, y_3, z_3)$ เป็นจุดใดๆบนรูปเหลี่ยมที่ไม่วางอยู่บนเส้นตรงเดียวกัน Normal Vector ที่ได้จากการคำนวณทั้งสองวิธีจะต้องถูก Normalize ให้มีขนาดเป็น 1 ก่อนโดยใช้สูตร

$$n=\begin{bmatrix}\dfrac{x}{\sqrt{x^2+y^2+z^2}}\\\dfrac{y}{\sqrt{x^2+y^2+z^2}}\\\dfrac{z}{\sqrt{x^2+y^2+z^2}}\end{bmatrix} \quad (4)$$

สำหรับพื้นผิวเรียบแล้วทุกๆจุดจะมี Normal Vector ชี้ไปในทิศทางเดียวกันหมด แต่สำหรับพื้นผิวโค้งซึ่งประกอบด้วยพื้นผิวเรียบย่อยๆหากกำหนดให้ Normal Vector ตั้งฉากกับพื้นผิวเรียบย่อยๆแล้ว เมื่อแสดงผลจะมองเห็นรอยต่อของพื้นผิวได้ชัดเจนเนื่องจากทิศทางของ Normal Vector ไม่ต่อเนื่องที่บริเวณรอยต่อของพื้นผิวดังรูปที่ 10 ซึ่งปัญหานี้สามารถแก้ได้โดยใช้การเฉลี่ยค่า Normal Vector ที่รอยต่อ ตัวอย่างในรูปที่ 11 ที่จุด P ค่า Normal Vector ควรเป็นค่าเฉลี่ยจาก Normal Vector สี่จุดรอบๆหรือ $n_p = n_1+n_2+n_3+n_4$



รูปที่ 10. เรนเดอร์โดยใช้ Normal Vector ของรูปเหลี่ยม(ซ้าย) และใช้ Normal Vector จริง (ขวา)



รูปที่ 11. การเฉลี่ยค่า Normal Vector ที่จุด Vertex

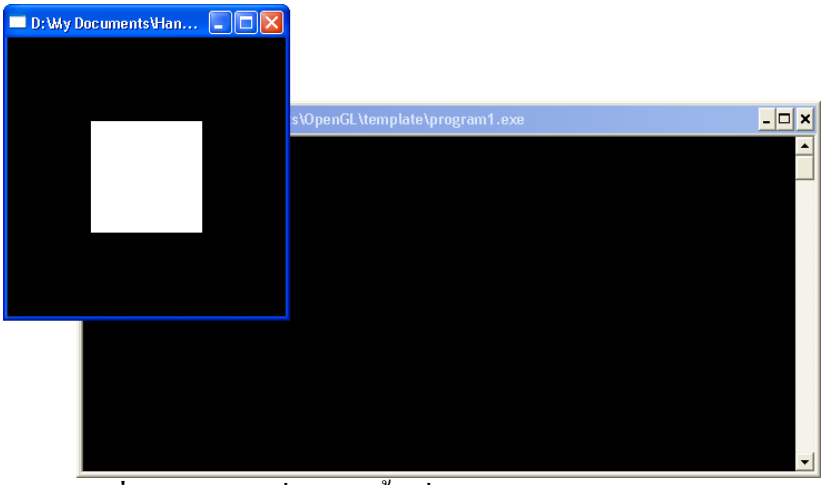
การทดลอง

ในการทดลองจะทำการเขียนโปรแกรมภาษา C ที่ใช้ OpenGL ทำงานบนระบบปฏิบัติการ Windows โดยมี GLUT เป็นตัวช่วยติดต่อกับระบบปฏิบัติการและผู้ใช้ การทดลองประกอบด้วย 4 ส่วนหลักคือ

- Part A – Introduction to GL Programming โปรแกรมเริ่มต้นสำหรับการเขียน OpenGL
- Part B – Drawing วาดรูปทรงพื้นฐานของ OpenGL
- Part C – Viewing ควบคุมมุมมองวัตถุ เปลี่ยนแปลงขนาดรูปทรงและการจัดวางวัตถุ
- Part D – Lighting ให้แสงกับวัตถุโดยกำหนดโมเดลของแสงและโมเดลการสะท้อนแสงของวัตถุ

Part A **Introduction to GL Programming:** ในส่วนนี้เป็นคำสั่งที่ใช้เตรียม Windows สำหรับ OpenGL, การตั้งค่าพารามิเตอร์พื้นฐานต่างๆ, การฉายวัตถุลงบนหน้าจอ, และการรับคำสั่งจากคีย์บอร์ด โดยคำสั่งและฟังก์ชันทั้งหมดนี้รวมอยู่ในไฟล์ template.cpp (ดูการ Setup GLUT ในภาคผนวก)

<u>คำสั่ง</u> รันโปรแกรมและสังเกตผลที่เกิดขึ้น จากนั้นทดลองปรับขนาดหน้าต่าง



รูปที่ 12. หน้าต่างที่ปรากฏขึ้นเมื่อคอมไพล์และรันโปรแกรม

List1

```
1:    #include <windows.h>
2:    #include <GL/glut.h>
3:    #define SIZEW 50.0
4:    int i;
```

- include ไฟล์ที่จำเป็นเข้ามาในโปรแกรมได้แก่ windows.h และ glut.h โดย glut.h ได้รวม gl.h และ glu.h ไว้ด้วยแล้วจึงไม่จำเป็นต้อง include ซ้ำอีก
- `#define SIZEW 50.0` กำหนดขอบเขตของวัตถุ (ใช้ใน List4)

List2

```
1:    void init(void)
2:    {
3:      glEnable(GL_DEPTH_TEST);
4:      glClearColor(0.0, 0.0, 0.0, 0.0);
5:    }
```

ฟังก์ชัน `init()` ใช้สำหรับตั้งค่าที่จำเป็นต่างๆใน OpenGL

▪ `glEnable(GL_DEPTH_TEST);` เรียกว่า Hidden-Surface Removal เป็นการเปิด Depth Buffer ซึ่งใช้เก็บความลึก *z* ที่แต่ละพิกเซลของหน้าจอเพื่อเปรียบเทียบป้องกันการวาดวัตถุซ้อนทับกันผิดลำดับ

▪ `glClearColor(GLfloat red, GLfloat green, GLfloat blue, GLfloat alpha);` เป็นการตั้งค่าสีพื้นหลัง โดย `(0.0,0.0,0.0,0.0)` เป็นสีดำและ `(1.0,1.0,1.0,0.0)` เป็นสีขาว

List3

```
1:    void display(void)
2:    {
3:      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
4:      glPushMatrix();
5:        glRectf(-20.0, -20.0, 20.0, 20.0);
6:      glPopMatrix();
7:      glutSwapBuffers();
8:    }
```

ฟังก์ชัน `display()` จะถูกเรียกใช้ทุกครั้งที่ Windows ต้องการวาดหน้าจอใหม่ ในการทดลองหัวข้อต่อไปจะเพิ่มคำสั่งเข้ามาในฟังก์ชันนี้เป็นหลัก

▪ `glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);` เป็นการเคลียร์ค่าใน Color Buffer ซึ่งเก็บรูปที่แสดงออกทางหน้าจอให้เป็นสีพื้นหลังตามที่กำหนดใน `glClearColor();` และเคลียร์ค่าใน Depth Buffer

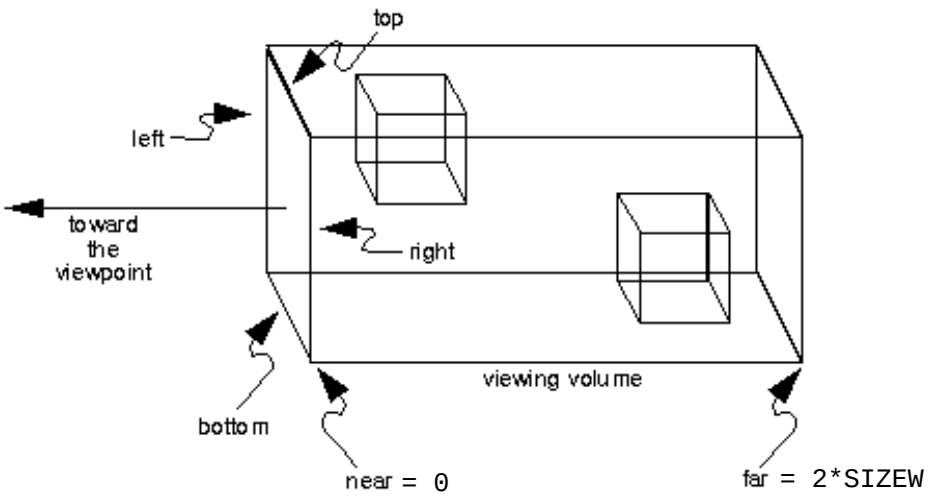
▪ คำสั่งที่ใช้วาดรูปสองหรือสามมิติจะแทรกระหว่าง `glPushMatrix();` ซึ่งใช้สำหรับคัดลอกเมตริกทรานฟอร์ม (Transformation Matrix) ปัจจุบันลงใน stack และ `glPopMatrix();` สำหรับดึงเมตริกที่เก็บไว้ออกมาจาก stack (รายละเอียดของ `glPushMatrix();` และ `glPopMatrix();` อยู่ใน PartC.2)

▪ `glRectf(GLfloat x1, GLfloat y1, GLfloat x2, GLfloat y2);` วาดสี่เหลี่ยมที่มีจุดมุมอยู่ที่พิกัด *(x1, y1)* และ *(x2, y2)* บนระนาบ $z = 0$

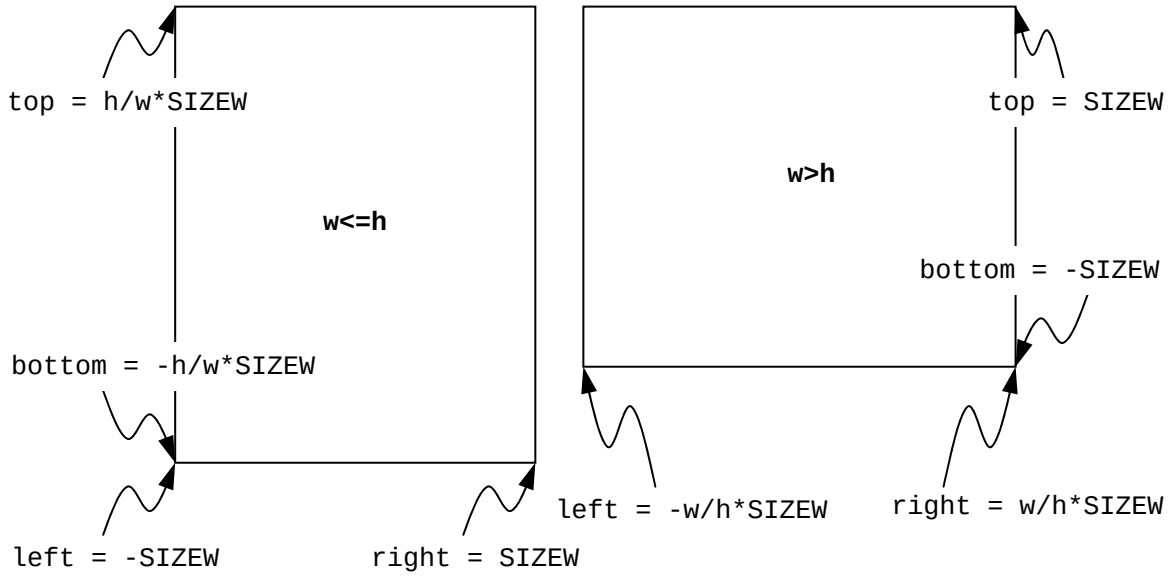
▪ `glutSwapBuffers();` เป็นคำสั่งแสดงสิ่งที่วาดออกทางหน้าจอ

List4

```
1:    void reshape(int w, int h)
2:    {
3:      glViewport(0, 0, w, h);
4:      glMatrixMode(GL_PROJECTION);
5:      glLoadIdentity();
6:      if(w<=h && w>0)
7:        glOrtho(-SIZEW,SIZEW,(GLdouble)-h/w*SIZEW,
          (GLdouble)h/w*SIZEW,0.0,2*SIZEW);
8:      else if(h>0)
9:        glOrtho((GLdouble)-w/h*SIZEW,(GLdouble)w/h*SIZEW,
          -SIZEW,SIZEW,0.0,2*SIZEW);
10:     glMatrixMode(GL_MODELVIEW);
11:     glLoadIdentity();
12:     glTranslatef(0.0,0.0,-SIZEW);
13:   }
```



รูปที่ 13. ขอบเขตการมองเห็นสำหรับการฉายภาพแบบขนาน



รูปที่ 14. ขอบเขตซ้ายขวาและบนล่างสำหรับหน้าต่างที่มีความกว้างน้อยกว่าความสูง (ซ้าย) และหน้าต่างที่มีความกว้างมากกว่าความสูง (ขวา) ค่าของ SIZEW ถูกกำหนดใน List1 ส่วน h และ w คือความสูงและความกว้างของหน้าต่างมีหน่วยเป็นพิกเซล

ฟังก์ชัน `reshape()` จะถูกเรียกใช้เมื่อหน้าต่างของ OpenGL ถูกปรับขนาดหรือถูกเปิดครั้งแรก

- `glViewport(GLint x, GLint y, GLint width, GLint height);` เป็นการจัดพิกัดของ OpenGL ให้เท่ากับหน้าต่างของ Windows (Viewport Transformation)
- `glMatrixMode(GL_PROJECTION);` เปลี่ยนโหมดของเมตริกซ์จาก `GL_MODELVIEW` เป็น `GL_PROJECTION` ทำให้การแปลงเมตริกซ์หลังจากนี้จะมีผลกับการที่วัตถุถูกฉายลงบนระนาบ 2 มิติโดยไม่มีผลกระทบกับตัววัตถุ ส่วนคำสั่ง `glMatrixMode(GL_MODELVIEW);` จะเป็นการเปลี่ยนโหมดของเมตริกซ์กลับไปเป็น `GL_MODELVIEW`

▪ `glLoadIdentity();` แทนเมตริกที่มีอยู่ด้วยเมตริกเอกลักษณ์ เนื่องจากเมตริกการแปลงตัวใหม่จะถูกคูณกับเมตริกที่มีอยู่ หากไม่เคลียร์เมตริกเก่าทิ้งจะเกิดความผิดพลาดขึ้นได้

▪ `glOrtho2D(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far);` ฉายภาพสามมิติลงบนจอภาพโดยใช้การแปลงเมตริกแบบขนาน (Orthogonal Transformation) โดยกำหนดขอบเขตของวัตถุหรือ Clipping Plane ดังรูปที่ 13 และ 14 วัตถุที่วาดทั้งหมดจะต้องมีพิกัดไม่เกินขอบเขตนี้จึงจะมองเห็นได้ในจอภาพ หากเกินกว่านี้จะถูกตัดออก (Clipped)

▪ `glTranslatef(0.0, 0.0, -SIZEW)` ดู Part C.1

List5

```
1:    void keyboard(unsigned char key, int x, int y)
2:    {
3:      switch(key){
4:        case 27:
5:          exit(0);
6:          break;
7:        case 'f':
8:          glutFullScreen();
9:          break;
10:       case 'w':
11:         glutReshapeWindow(250, 250);
12:         break;
13:     }
14:   }
```

ฟังก์ชั่น `keyboard()` ถูกเรียกใช้งานทุกครั้งที่มีการกดปุ่มบนคีย์บอร์ด

▪ กดปุ่ม escape (ASCII Code 27) จะออกจากโปรแกรม

▪ กดปุ่ม f เรียกฟังก์ชัน `glutFullScreen();` ใช้แสดงหน้าต่างของ OpenGL แบบเต็มจอ

▪ กดปุ่ม w เรียกฟังก์ชัน `glutReshapeWindow(w, h);` ปรับหน้าจอให้มีขนาด $w \times h$

List6

```
1:    void main (int argc, char** argv)
2:    {
3:      glutInit(&argc, argv);
4:      glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);
5:      glutInitWindowSize(250, 250);
6:      glutCreateWindow(argv[0]);
7:      init();
8:      glutReshapeFunc(reshape);
9:      glutKeyboardFunc(keyboard);
10:     glutDisplayFunc(display);
11:     glutIdleFunc(display);
12:     glutMainLoop();
13:   }
```

ใน `main()` บรรทัดที่ 3 ถึง 7 เป็นการตั้งพารามิเตอร์ของ OpenGL บรรทัดที่ 8 ถึง 11 ประกาศฟังก์ชั่นที่ทำหน้าที่ต่างๆ และบรรทัดที่ 12 เริ่มรอรับคำสั่งจากผู้ใช้

Part B **Drawing:** อธิบายถึงคำสั่งที่ใช้ในการวาดรูปทรงเรขาคณิตพื้นฐานของ OpenGL และ GLU

B.1 **Drawing Primitives:** วาดรูปทรงพื้นฐานของ OpenGL ได้แก่จุด เส้น และรูปทรงเหลี่ยม

<u>คำสั่ง</u> แก้ template.cpp ให้ Display Function เป็น List7 จากนั้นรันโปรแกรมและสังเกตผลที่เกิดขึ้น ทดลองเปลี่ยนสี, ตำแหน่งของ vertex, แบบของรูปทรง, และลักษณะของรูปทรงที่สร้างขึ้น

List7

```
void display(void)
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glPushMatrix();
    glBegin(GL_POLYGON);
      glColor3f(0.0, 0.0, 1.0);
      glVertex3f(0.0, 0.0, 0.0);
      glColor3f(1.0, 0.0, 0.0);
      glVertex3f(30.0, 0.0, 0.0);
      glColor3f(0.0, 1.0, 0.0);
      glVertex3f(0.0, 30.0, 0.0);
    glEnd();
  glPopMatrix();
  glutSwapBuffers();
}
```

▪ `glColor3f(GLfloat red, GLfloat green, GLfloat blue) ;` ใช้กำหนดสีของ Vertex โดยกำหนดจากแม่สีสามสีคือแดงเขียวน้ำเงิน มีค่าจาก 0.0 ถึง 1.0 โดยปกติแล้วถ้าหากแต่ละ Vertex ถูกกำหนดสีที่ต่างกัน สีภายในรูปเหลี่ยมจะเกิดจากการประมาณค่า

▪ `glVertex{234}{sifd}[v](TYPEcoords);` กำหนดตำแหน่ง Vertex ของรูปทรง ซึ่งสามารถกำหนดได้ 3 แบบคือ *(x, y), (x, y, z),* และ *(x, y, z, w)* ขึ้นอยู่กับการเลือกคำสั่ง ในที่นี้ใช้ *(x, y, z)* การเรียกคำสั่ง glVertex() จะต้องอยู่ระหว่าง `glBegin();` และ `glEnd();`

▪ `glBegin(GLenum mode);` ประกาศจุดเริ่มต้นของชุดคำสั่ง `glVertex();` ที่ใช้วาดรูปทรง ชนิดของรูปทรงระบุโดย mode ซึ่งเป็นได้ 10 แบบตามรูปที่ 15 โดยคำสั่งจะปิดท้ายด้วย `glEnd();` เสมอ

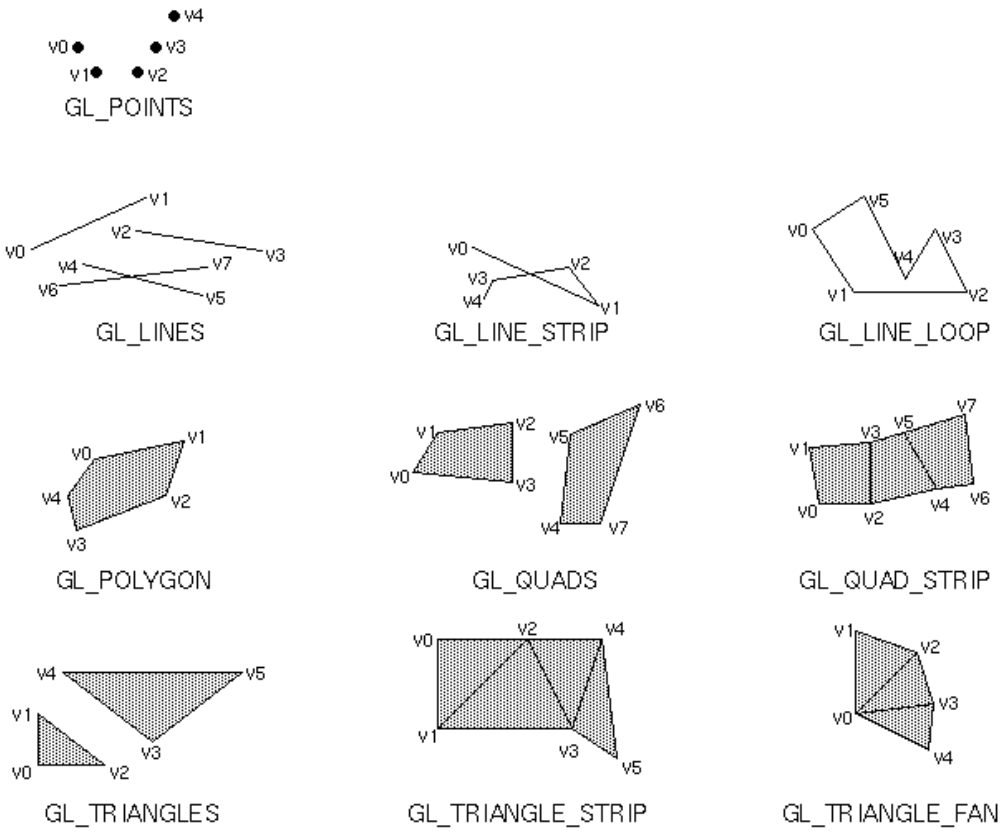
▪ นอกจากนี้ยังมีคำสั่งที่ใช้กำหนดลักษณะของจุด, เส้น, และรูปเหลี่ยมที่สร้างขึ้นด้วย โดยจะต้องกำหนดลักษณะเหล่านี้ก่อนคำสั่ง `glBegin();` คำสั่งเหล่านี้ไดแก่

▪ `glPointSize(GLfloat size);` ใช้กำหนดขนาดของจุด โดยปกติจะมีค่าเป็น 1.0

▪ `glLineWidth(GLfloat width);` กำหนดความกว้างของเส้น ปกติเท่ากับ 1.0

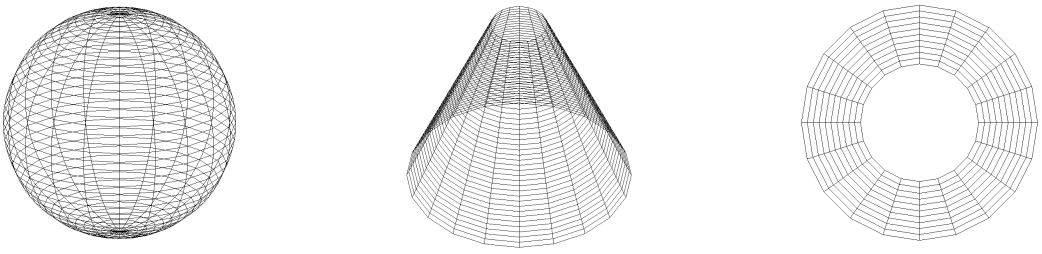
▪ `glPolygonMode(GLenum face, GLenum mode);` กำหนดลักษณะของรูปเหลี่ยมโดย `face` หมายถึงด้านหน้าหรือหลัง และ `mode` คือวิธีการวาดรูปเหลี่ยมด้านนั้นๆ (สองด้านวาดแตกต่างกันได้) สามารถกำหนด `face` ได้เป็น `GL_FRONT_AND_BACK`, `GL_FRONT`, หรือ `GL_BACK` และ `mode` เป็น `GL_POINT`, `GL_LINE`, หรือ `GL_FILL` ซึ่งหมายถึงการวาดเป็นจุด เส้นโครงร่าง หรือเติมเต็ม

▪ `glFrontFace(GLenum mode);` ใช้กำหนดด้านของรูปเหลี่ยม โดยปกติ `mode` จะเป็น `GL_CCW` ซึ่งหมายถึงด้านที่ `Vertex` วนทวนเข็มนาฬิกาถือเป็นด้านหน้า หากกำหนดเป็น `GL_CW` จะกลับกัน



รูปที่ 15. รูปทรงพื้นฐานแบบต่างๆของ OpenGL

B.2 **Drawing Quadric Primitives:** วาดรูปทรงแบบ Quadric ที่มีใน GLU ได้แก่รูปทรงกลม ทรงกรวย และแผ่นจาน



a) **b)** **c)**

รูปที่ 16. Quadric Primitive a) Sphere, b) Cylinder, c) Disk

<u>คำสั่ง</u> แก้ template.cpp ให้ Header เป็น List8, Init Function เป็น List9, และ Display Function เป็น List10 จากนั้นรันโปรแกรมและสังเกตผลที่เกิดขึ้น เปลี่ยนชนิดของรูปทรงและลักษณะการวาด

List8

```
#include <windows.h>
#include <GL/glut.h>
#define SIZEW 50.0
GLUquadricObj *theObj;
int i;
```

List9

```
1:    void init(void)
2:    {
3:      glEnable(GL_DEPTH_TEST);
4:      glClearColor(0.0, 0.0, 0.0, 0.0);
5:      theObj = gluNewQuadric();
6:    }
```

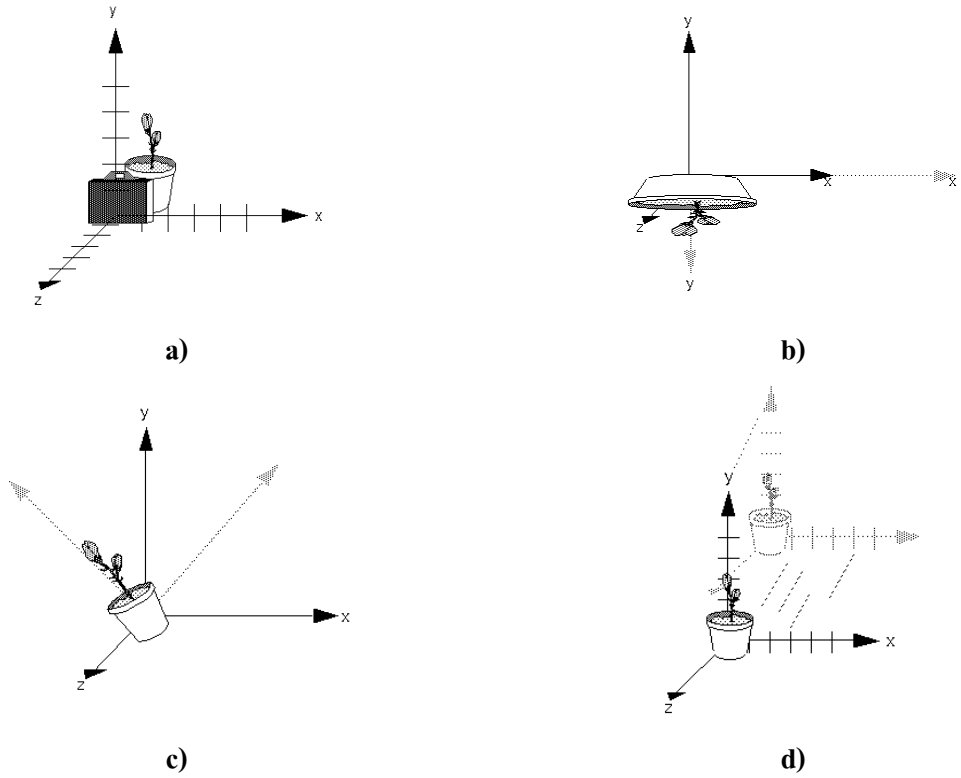
List10

```
1:    void display(void)
2:    {
3:      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
4:      glPushMatrix();
5:        glRotatef(++i%360, 0.0, 1.0, 1.0);
6:        gluQuadricDrawStyle(theObj, GLU_LINE);
7:        gluSphere(theObj, 40.0, 50, 20);
8:      glPopMatrix();
9:      glutSwapBuffers();
10:   }
```

- `glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z);` ดู Part C:
- GLU มีคำสั่งที่ใช้วาดรูปทรงพิเศษได้แก่ `gluSphere();` สำหรับทรงกลม `gluCylinder();` สำหรับทรงกรวย และ `gluDisk();` สำหรับแผ่นจาน โดยจะเรียกรูปทรงเหล่านี้ว่า Quadric และจะต้องใช้คำสั่งต่อไปนี้ประกอบ
- `GLUquadricObj *theObj;` ประกาศตัวแปร Quadric ชื่อ theObj
- `theObj = gluNewQuadric();` สร้าง Quadric Object ให้ `theObj`
- `gluDeleteQuadric(theObj);` ลบ Quadric Object ชื่อ theObj ซึ่งไม่ได้ใช้งานแล้ว
- `gluQuadricDrawStyle(theObj, GLenum drawStyle);` กำหนดลักษณะการวาดรูปทรงของ `theObj` โดย `drawStyle` มี 3 แบบหลักคือ `GLU_FILL, GLU_LINE, GLU_POINT` ซึ่งวาดวัตถุเป็นทรงทึบ เส้น และจุดตามลำดับ โดยค่าเริ่มต้นจะเป็น `GLU_FILL`
- `gluSphere(theObj, GLdouble radius, GLint slices, Glint stacks);` ใช้วาดทรงกลมศูนย์กลางที่จุดกำเนิด มีรัศมี `radius` โดย `stacks` เป็นจำนวนชั้นเทียบได้กับเส้นละติจูด และ `slices` เป็นจำนวนเส้นตรงที่มาต่อเป็นเส้นโค้งซึ่งเทียบได้กับเส้นลองจิจูด
- `gluCylinder(theObj, GLdouble baseRadius, GLdouble topRadius, GLdouble height, GLint slices, GLint stacks);` ใช้วาดรูปทรงกรวยมีแกนกลางอยู่บนแกน z โดยฐานอยู่ที่ `z=0` มีรัศมี `baseRadius` และยอดที่ `z = height` มีรัศมี `topRadius` ส่วน `stacks` และ `slices` มีความหมายเหมือนกันกับทรงกลม
- `gluDisk(theObj, GLdouble innerRadius, GLdouble outerRadius, GLint slices, Glint loops);` ใช้วาดแผ่นจานบนระนาบ `z=0` โดย `outerRadius` คือรัศมีภายนอก, `innerRadius` คือรัศมีภายใน, `slices` คือจำนวนเส้นตรงที่มาต่อกันเป็นวงกลมและ `loops` คือจำนวนวงแหวน ด้าน `+z` ถือเป็นด้านบนของแผ่นจาน

Part C **Viewing:** อธิบายคำสั่งที่ใช้ในการจัดตำแหน่งวัตถุและกล้องหรือ Modelview Transformation และเทคนิคในการควบคุมการเคลื่อนไหวของวัตถุไปยังตำแหน่งที่ต้องการ

C.1 **Modelview Transformation** คำสั่งในการเคลื่อนย้ายวัตถุและเปลี่ยนรูปทรง



รูปที่ 17. a) กล้องและวัตถุมีตำแหน่งเริ่มต้นอยู่ที่จุดกำเนิด, b) `glScalef(2.0, -0.5, 1.0)`, c) `glRotatef(45.0, 0.0, 0.0, 1.0)`, d) `glTranslatef()`

<u>**คำสั่ง**</u> แก้ template.cpp ให้ Display Function เป็น List11 จากนั้นรันโปรแกรมและสังเกตผลที่เกิดขึ้น ทดลองเปลี่ยนชนิดการแปลงและเปลี่ยนขนาดของพารามิเตอร์ที่ใช้

List11

```
1:    void display(void)
2:    {
3:      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
4:      glPushMatrix();
5:        glScalef(0.8, 1.2, 1.0);
6:        glutWireTeapot(20.0);
7:      glPopMatrix();
8:      glutSwapBuffers();
9:    }
```

▪ `glutWireTeapot(GLdouble size);` วาดกาน้ำชาเป็นเส้นโครงร่าง คำสั่งนี้อยู่ใน GLUT ปกติใช้สำหรับทดสอบคำสั่งต่างๆ

▪ `glScalef(GLfloat x, GLfloat y, GLfloat z);` ทำให้วัตถุมีการยืด หด หรือสะท้อนตามแนวแกน x, y, และ z โดยสถานะปกติมีค่าเท่ากับ 1.0 หากมากกว่า 1.0 จะเป็นการยืด น้อยกว่า 1.0 เป็นการหด และน้อยกว่า 0.0 เป็นการสะท้อน (ดูรูปที่ 17.b)

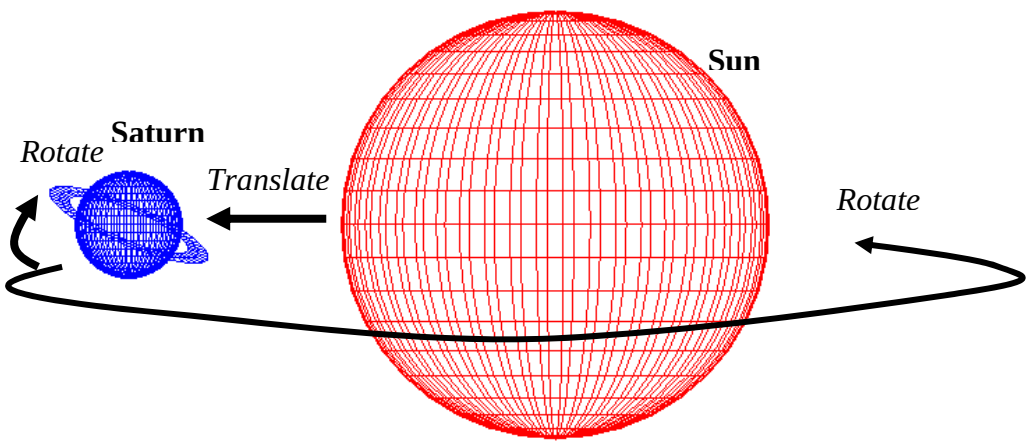
▪ `glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z);` หมุนวัตถุรอบแกน *x, y,* หรือ *z* ทวนเข็มนาฬิกาโดยมุมเท่ากับ `angle` มีหน่วยเป็นองศา (ดูรูปที่ 17.c)

▪ `glTranslatef(GLfloat x, GLfloat y, GLfloat z);` เลื่อนวัตถุตามแนวแกน **ระวัง** หากเลื่อนวัตถุเกินขอบเขตของค่า SIZEW จะทำให้วัตถุอยู่นอกปริมาตรการมองและวัตถุนั้นจะไม่ถูกเรนเดอร์ แก้ไขโดยการเพิ่มค่า SIZEW ใน List1 หรือลดระยะการเลื่อนลง (ดูรูปที่ 17.d)

▪ `glTranslatef(0.0, 0.0, -SIZEW);` ใน List4 ของ Part A ใช้ดึงวัตถุถอยออกมาจากจุดกำเนิด

▪ หมายเหตุ: หากการทำ Modelview Transformation มีมากกว่า 1 ครั้ง การ Transform จะเกิดจากหลังมาหน้า เช่นหากสั่ง `glRotate();` ตามด้วย `glTranslate();` คำสั่งหลัง คือ `glTranslate();` จะถูกกระทำก่อน

C.2 **Building the Solar System** จำลองระบบสุริยะโดยมีดวงอาทิตย์เป็นศูนย์กลางหมุนรอบตัวเองและดาวเสาร์ซึ่งหมุนรอบดวงอาทิตย์เป็นวงกลม



รูปที่ 18. Solar System

<u>คำสั่ง</u> แก้ template.cpp ให้ Display Function เป็น List12 ส่วน Header เป็น List8, Init Function เป็น List9 จากนั้นรันโปรแกรมและสังเกตผลที่เกิดขึ้น

List12

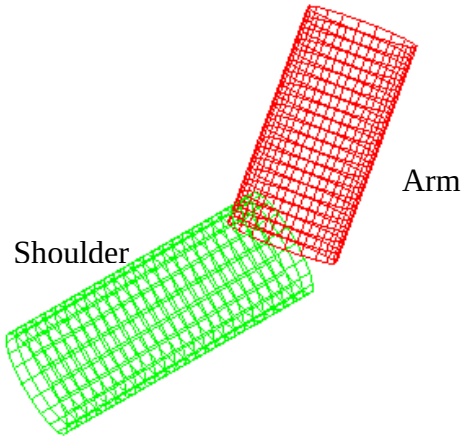
```
1:    void display(void)
2:    {
3:      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
4:      glPushMatrix();
5:        gluQuadricDrawStyle(theObj,GLU_LINE);
6:        glRotatef(++i%360, 0.0, 1.0, 0.0);
7:        glPushMatrix();                //Position A
8:          glTranslatef(40.0, 0.0, 0.0);
9:          glColor3f(0.0, 0.0, 1.0);
10:         glPushMatrix();              //Position B
11:           glRotatef(90.0, 1.0, 0.0, 0.5);
12:           gluDisk(theObj, 6.0, 8.0, 50, 2);
13:         glPopMatrix();               //Position B
14:         glRotatef(90.0, 1.0, 0.0, 0.0);
15:         gluSphere(theObj, 5.0, 50, 20);
16:       glPopMatrix();                 //Position A
17:       glColor3f(1.0, 0.0, 0.0);
18:       glRotatef(90.0, 1.0, 0.0, 0.0);
19:       gluSphere(theObj, 20.0, 50, 20);
20:     glPopMatrix();
21:     glutSwapBuffers();
22:   }
```

▪ `glPushMatrix();` ใช้คัดลอกเมตริกปัจจุบันลงใน Stack คล้ายกับการจำตำแหน่งหรือสถานะปัจจุบันไว้

▪ `glPopMatrix();` ใช้ลบเมตริกปัจจุบันทิ้ง แล้วเอาเมตริกอันบนสุดใน Stack มาแทนที่ เทียบได้กับการกลับไปยังตำแหน่งหรือสถานะเดิม

▪ การทำงานทั้งหมดของโปรแกรมนี้สามารถอธิบายได้ดังนี้ บรรทัดที่ 5 กำหนดลักษณะของ Quadric ที่จะวาดเป็นแบบเส้น บรรทัดที่ 6 ทำการหมุนวัตถุทั้งหมดโดยเพิ่มมุมขึ้นเรื่อยๆทุกครั้งที่มีการเรียกใช้ฟังก์ชันนี้ บรรทัดที่ 7 จำตำแหน่งปัจจุบันไว้ (ตำแหน่ง A) จากนั้นเลื่อนไปทางแกน x 40 หน่วยในบรรทัดที่ 8 เพื่อที่จะวาดดาวเสาร์และวงแหวนของดาว บรรทัดที่ 9 เปลี่ยนสีที่จะวาดเป็นสีน้ำเงิน จากนั้นจำตำแหน่งปัจจุบันไว้อีกครั้งหนึ่งในบรรทัดที่ 10 (ตำแหน่ง B) บรรทัดที่ 11-12 วาดวงแหวนที่ถูกจับหมุนเอียง 45 องศา จากนั้นกลับไปยังตำแหน่ง B เพื่อวาดทรงกลมของดาวเสาร์ที่ถูกหมุนขึ้นตั้งตรงแล้ว (บรรทัดที่ 14-15) ต่อมาบรรทัดที่ 16 จะกลับไปยังตำแหน่ง A หรือตำแหน่งเริ่มต้นเพื่อวาดดวงอาทิตย์ (บรรทัดที่ 17-19)

C.3 **Robot Arm** ควบคุมการเคลื่อนไหวของแขนหุ่นยนต์ซึ่งประกอบด้วยสองส่วนคือ Shoulder และ Arm ที่จะเคลื่อนไหวเมื่อมีผู้ใช้กดคีย์ 4, 8, 6, และ 2



รูปที่ 19. Robot Arm

<u>คำสั่ง</u> แก้ template.cpp ให้ Header เป็น List13, Display Function เป็น List14, Keyboard Function เป็น List15 ส่วน Init Function เป็น List9 จากนั้นรันโปรแกรมและสังเกตผลที่เกิดขึ้น

List13

```
#include <windows.h>
#include <GL/glut.h>
#define SIZEW 50.0
GLUquadricObj *theObj;
int i, arm, shoulder;
```

List14

```
void display(void)
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glPushMatrix();
    gluQuadricDrawStyle(theObj,GLU_LINE);
    glRotatef(75.0, 0.0, 1.0, 0.0);
    glRotatef(shoulder, 1.0, 0.0, 0.0);
    glPushMatrix();
      glTranslatef(0.0, 0.0, 25.0);
      glRotatef(arm, 1.0, 0.0, 0.0);
      glColor3f(1.0, 0.0, 0.0);
      gluCylinder(theObj, 5.0, 5.0, 20.0, 20, 20);
    glPopMatrix();
    glColor3f(0.0, 1.0, 0.0);
    gluCylinder(theObj, 5.0, 5.0, 25.0, 20, 25);
  glPopMatrix();
  glutSwapBuffers();
}
```

List15

```
void keyboard(unsigned char key, int x, int y)
{
  switch(key){
    case 27:
      exit(0);
      break;
    case 'f':
      glutFullScreen();
      break;
    case 'w':
      glutReshapeWindow(250, 250);
      break;
    case '8':
      shoulder = --shoulder%360;
      break;
    case '2':
      shoulder = ++shoulder%360;
      break;
    case '4':
      arm = --arm%360;
      break;
    case '6':
      arm = ++arm%360;
      break;
  }
}
```

Part D **Lighting:** อธิบายคำสั่งที่ใช้ในการกำหนดคุณสมบัติของแหล่งกำเนิดแสงและพื้นผิววัตถุรวมทั้งกำหนด Normal Vector ให้กับแต่ละ Vertex บนพื้นผิว

D.1 **Light & Material Properties** กำหนดคุณสมบัติของแหล่งกำเนิดแสงและพื้นผิววัตถุ

<u>คำสั่ง</u> แก้ template.cpp ให้ Init Function เป็น List16 และ Display Function เป็น List17 จากนั้นรันโปรแกรมและสังเกตผลที่เกิดขึ้น ทดลองเปลี่ยนตำแหน่งและสีของแหล่งกำเนิดแสงรวมทั้งคุณสมบัติการสะท้อนแสงของพื้นผิววัตถุ ทดลองปิดแหล่งกำเนิดแสงและสังเกตความแตกต่าง

List16

```
void init(void)
{
  float mat_specular[] = {1.0, 1.0, 1.0, 1.0};
  float mat_shininess[] = {50.0};
  float light_position[] = {1.0, 1.0, 1.0, 0.0};
  float light_ambient[] = {0.5, 0.5, 0.5, 1.0};
  glLightfv(GL_LIGHT0, GL_POSITION, light_position);
  glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
  glEnable(GL_LIGHT0);
  glEnable(GL_LIGHTING);
  glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
  glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
  glColorMaterial(GL_FRONT_AND_BACK,
    GL_AMBIENT_AND_DIFFUSE);
  glEnable(GL_COLOR_MATERIAL);
  glEnable(GL_DEPTH_TEST);
  glClearColor(0.0, 0.0, 0.0, 0.0);
}
```

List17

```
1:    void display(void)
2:    {
3:      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
4:      glPushMatrix();
5:        glColor3f(0.1, 0.5, 0.8);
6:        glutSolidTeapot(30.0);
7:      glPopMatrix();
8:      glutSwapBuffers();
9:    }
```

▪ `glutSolidTeapot(GLdouble size);` วาดกาน้ำชา ปกติแล้วจะต้องกำหนด Normal Vector ให้กับแต่ละ Vertex บนพื้นผิวของวัตถุ แต่คำสั่งนี้ได้รวมคำสั่งกำหนด Normal Vector ไว้เรียบร้อยแล้วจึงไม่ต้องกำหนดซ้ำอีก (คำสั่งสร้าง Quadric Primitive ได้รวมการกำหนด Normal Vector ไว้แล้วเช่นกัน)

▪ `glLight{if}v(GLenum light, GLenum pname, TYPEparam);` กำหนดสมบัติของแหล่งกำเนิดแสง `light` โดยแหล่งกำเนิดแสงสามารถมีได้อย่างต่ำ 8 แหล่ง (อาจจะมีได้มากกว่านี้ขึ้นอยู่กับระบบ) เรียกเป็น `GL_LIGHT0, GL_LIGHT1,..., GL_LIGHT7` ส่วน `pname` คือชื่อของพารามิเตอร์ที่ตั้งค่าได้ (ดูตารางที่ 2) ค่าของพารามิเตอร์นั้นกำหนดโดยอาร์เรย์ `param`

ตารางที่ 2 พารามิเตอร์ของแหล่งกำเนิดแสงและค่าเริ่มต้น

Parameter Name (pname)	Default Value (param)	Meaing
GL_AMBIENT	{0.0, 0.0, 0.0, 1.0}	ambient RGBA intensity of light
GL_DIFFUSE	{1.0, 1.0, 1.0, 1.0}	diffuse RGBA intensity of light
GL_SPECULAR	{1.0, 1.0, 1.0, 1.0}	specular RGBA intensity of light
GL_POSITION	{0.0, 0.0, 1.0, 1.0}	(*x*, *y*, *z*, *w*) position of light
GL_SPOT_DIRECTION	{0.0, 0.0, -1.0}	(*x*, *y*, *z*) direction of spotlight
GL_SPOT_EXPONENT	0.0	spotlight exponent
GL_SPOT_CUTOFF	180.0	spotlight cutoff angle
GL_CONSTANT_ATTENUATION	1.0	constant attenuation factor
GL_LINEAR_ATTENUATION	0.0	linear attenuation factor
GL_QUADRATIC_ATTENUATION	0.0	quadratic attenuation factor

***หมายเหตุ** ค่าเริ่มต้นของ `GL_DIFFUSE` และ `GL_SPECULAR` ในตารางใช้กับ `GL_LIGHT0` เท่านั้น แหล่งกำเนิดแสงตัวอื่นจะมีค่าเริ่มต้นเป็น `{0.0, 0.0, 0.0, 1.0}`

▪ `glLightfv(GL_LIGHT0, GL_POSITION, light_position);` เป็นการจัดตำแหน่งแสงจากแหล่งกำเนิด `GL_LIGHT0` เป็น *(x, y, z, w) = (1.0, 1.0, 1.0, 0.0)* จะเห็นว่าค่าสุดท้ายคือ *w* เป็น 0.0 ซึ่งทำให้แหล่งกำเนิดแสงมีตำแหน่งที่อนันต์และชี้มายังจุด *(1.0, 1.0, 1.0)* เรียกแหล่งกำเนิดแสงแบบนี้ว่า Directional Light Source หากค่า *w* ไม่เท่ากับศูนย์จะเป็น Positional Light Source มีตำแหน่งที่ *(x, y, z)* และส่งแสงออกมาทุกทิศทาง

▪ `glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);` ตั้งค่า Ambient Light ของ `GL_LIGHT0` เท่ากับ {0.5, 0.5, 0.5, 1.0}

▪ glEnable(GL_LIGHTING); ทำให้มีการคำนวณสีของวัตถุโดยคำนึงถึงแสงที่ให้และคุณสมบัติของพื้นผิววัตถุ

▪ glEnable(GL_LIGHT0); เป็นการเปิดแสงจากแหล่งกำเนิด GL_LIGHT0

▪ การเพิ่มแหล่งกำเนิดแสงจะมีผลกระทบกับประสิทธิภาพในการคำนวณเนื่องจาก OpenGL จะต้องคำนวณปริมาณแสงจากทุกๆแหล่งกำเนิดที่ตกลงบนแต่ละ vertex

ตารางที่ 3 พารามิเตอร์ของพื้นผิวและค่าเริ่มต้น

Parameter Name (pname)	Default Value (param)	Meaing
GL_AMBIENT	{0.2, 0.2, 0.2, 1.0}	ambient color of material
GL_DIFFUSE	{0.8, 0.8, 0.8, 1.0}	diffuse color of material
GL_AMBIENT_AND_DIFFUSE		ambient and diffuse color of material
GL_SPECULAR	{0.0, 0.0, 0.0, 1.0}	specular color of material
GL_SHININESS	0.0	specular exponent
GL_EMISSION	{0.0, 0.0, 0.0, 1.0}	Emissive color of material

▪ glMaterial{if}v(GLenum face, GLenum pname, TYPEparam); กำหนดสมบัติของพื้นผิววัตถุ เมื่อ face คือด้านของวัตถุที่จะกำหนดคุณสมบัติได้แก่ GL_FRONT, GL_BACK, หรือ GL_FRONT_AND_BACK ส่วน pname คือชื่อของพารามิเตอร์ที่ตั้งค่าได้ (ดูตารางที่ 3) ค่าของพารามิเตอร์นั้นกำหนดโดยอาร์เรย์ param

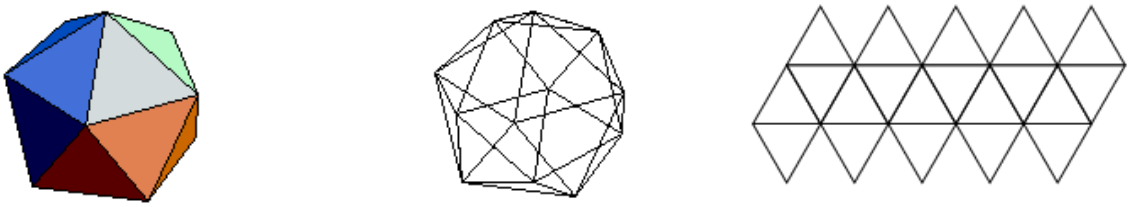
▪ glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular); กำหนดสีของ Specular Reflactance ให้เป็นสีขาว {1.0, 1.0, 1.0, 1.0}

▪ glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess); กำหนดขนาดและความสว่างของ Specular Reflectance ให้เป็น 50.0 สามารถกำหนดค่าอยู่ในช่วง [0.0, 128.0]

▪ glColorMaterial(GLenum face, GLenum mode); ให้สีของ mode เปลี่ยนตามคำสั่ง glColor*(); ทันที โดย mode สามารถเป็น GL_AMBIENT, GL_DIFFUSE, GL_AMBIENT_AND_DIFFUSE, GL_SPECULAR, หรือ GL_EMISSION และ face เป็น GL_FRONT, GL_BACK, หรือ GL_FRONT_AND_BACK

▪ glEnable(GL_COLOR_MATERIAL); อนุญาตการใช้งานคำสั่ง glColorMaterial();

D.2 **Normal Vector** กำหนด Normal Vector ให้กับ Vertex ของรูป Icosahedron (รูปเหลี่ยม 20 หน้า)



a) **b)** **c)**

รูปที่ 20. a) Icosahedron, b) เส้นโครงร่าง, และ c) รูปสามเหลี่ยมที่ประกอบขึ้นมา [7]

<u>คำสั่ง</u> แก้ template.cpp ให้ Header เป็น List18 และ Display Function เป็น List19 ส่วน Init Function ใช้ List16 จากนั้นรันโปรแกรมและสังเกตผลที่เกิดขึ้น

List18

```
1:    #include <windows.h>
2:    #include <GL/glut.h>
3:    #define SIZEW 1.5
4:    #define X .525731112119133606
5:    #define Z .850650808352039932
6:    int i, j;
7:    float vdata[12][3] = {
8:      {-X,0.0,Z}, {X,0.0,Z}, {-X,0.0,-Z}, {X,0.0,-Z},
9:      {0.0,Z,X}, {0.0,Z,-X}, {0.0,-Z,X}, {0.0,-Z,-X},
10:     {Z,X,0.0}, {-Z,X,0.0}, {Z,-X,0.0}, {-Z,-X,0.0}};
11:   int tindices[20][3] = {
12:     {0,1,4}, {0,4,9}, {9,4,5}, {4,8,5}, {4,1,8},
13:     {8,1,10}, {8,10,3}, {5,8,3}, {5,3,2}, {2,3,7},
14:     {7,3,10}, {7,10,6}, {7,6,11}, {11,6,0}, {0,6,1},
15:     {6,10,1}, {9,11,0}, {9,2,11}, {9,5,2}, {7,11,2}};
```

List19

```
1:    void display(void)
2:    {
3:      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
4:      glPushMatrix();
5:        glRotatef(++i%360, 1.0, 1.0, 1.0);
6:        glColor3f(0.3, 0.9, 0.4);
7:        for(j=0; j<20; j++){
8:          glBegin(GL_POLYGON);
9:            glNormal3fv(vdata[tindices[j][0]]);
10:           glVertex3fv(vdata[tindices[j][0]]);
11:           glNormal3fv(vdata[tindices[j][1]]);
12:           glVertex3fv(vdata[tindices[j][1]]);
13:           glNormal3fv(vdata[tindices[j][2]]);
14:           glVertex3fv(vdata[tindices[j][2]]);
15:         glEnd();
16:       }
17:     glPopMatrix();
18:     glutSwapBuffers();
19:   }
```

▪ Icosahedron ที่สร้างขึ้นประกอบด้วยรูปสามเหลี่ยมด้านเท่า 20 ชิ้นใช้ Vertex ร่วมกัน 12 จุด พิกัด *(x, y, z)* สำหรับ Vertex ทั้ง 12 จุดถูกกำหนดโดยตัวแปร `vdata[12][3]` ซึ่งใช้ตัวเลข `X` และ `Z` ที่ทำให้ระยะจากจุดกำเนิดถึงแต่ละ Vertex มีค่าเป็น 1.0 ตัวอย่างเช่น Vertex แรกมีพิกัดที่ `{-X,0.0,Z}` จะมีระยะห่างจากจุดกำเนิดเท่ากับ $\sqrt{X^2+Z^2} = \sqrt{0.52573^2+0.85065^2} = 1.0$

▪ ตัวแปร `tindices[20][3]` ใช้ควบคุมการเชื่อมต่อแต่ละ Vertex เข้าด้วยกันเพื่อให้เกิดเป็นสามเหลี่ยม 20 ชิ้น เช่นสามเหลี่ยมอันแรกใช้ Vertex อันที่ 0, 1, และ 4 พิกัดของสามเหลี่ยมทั้งหมดจะถูกกำหนดแบบวนทวนเข็มนาฬิกาเพื่อให้ด้านนอกของ Icosahedron เป็นด้านหน้า

▪ เนื่องจาก Icosahedron ที่สร้างขึ้นมีลักษณะเป็นทรงกลม Normal Vector ที่ใช้จะมีทิศทางเดียวกับเวคเตอร์ที่ชี้จากจุดกำเนิดถึงแต่ละ Vertex ดังนั้นเราสามารถใช้พิกัดของ Vertex เป็น Normal Vector ได้เลย

▪ `glNormal3fv(GLfloat *v);` ใช้ตั้งค่า Normal Vector สำหรับ Vertex ที่ถูกกำหนดตามมาด้วยคำสั่ง `glVertex3fv(GLfloat *v);`

ภาคผนวก

I. การ Setup OpenGL

ระบบปฏิบัติการรวมถึง IDE รุ่นใหม่ ๆ สนับสนุนการทำงานของ OpenGL โดยได้รวมไฟล์ที่จำเป็นสำหรับ compile, link, และ run โปรแกรมที่ใช้ OpenGL ไว้ด้วยแล้ว หากต้องการใช้ OpenGL ร่วมกับ Windows โดยไม่ใช้ GLUT เป็น API สำหรับ Visual C++ สามารถดูตัวอย่างการตั้งค่าเริ่มต้นของโปรแกรมได้ที่ [8] และสำหรับ Borland C++ Builder ที่ [9]

II. การ Setup GLUT

<u>สำหรับ Visual C++ [10]</u>

1. ดาวน์โหลดไฟล์ glut-3.7.6-bin.zip จาก http://www.xmission.com/~nate/glut/glut-3.7.6-bin.zip ไฟล์นี้ประกอบด้วยไฟล์ย่อย 3 ไฟล์คือ glut.h, glut32.lib, และ glut32.dll
2. คัดลอกไฟล์ glut.h ไปไว้ที่ C:\Program Files\Microsoft Visual Studio\VC\Include\GL
3. คัดลอกไฟล์ glut32.lib ไปไว้ที่ C:\Program Files\Microsoft Visual Studio\VC\Lib
4. คัดลอกไฟล์ glut32.dll ไปไว้ที่ C:\Windows\system32
5. เปิดโปรแกรม Visual C++ สร้าง project ใหม่ โดยเลือกให้เป็น Win32 Console Application และเป็น empty project
6. ไปที่เมนู Project > Add New Item จากนั้นเลือก C++ File (.cpp) และตั้งชื่อไฟล์ แล้วกด add
7. ดาวน์โหลดไฟล์ template.cpp จาก http://www.kmitl.ac.th/~kwwithaw/Files/template.cpp และคัดลอกโค้ดในไฟล์นี้ไปวางที่ไฟล์ .cpp ที่เพิ่งสร้างขึ้นใหม่ กด F5 หากถูกต้องโปรแกรม OpenGL จะรันขึ้นมา

<u>สำหรับ Borland C++ Builder [11]</u>

1. ดาวน์โหลดไฟล์ glut.zip จาก http://home.clara.net/paulyg/download/glut.zip ไฟล์นี้ประกอบด้วยไฟล์ย่อย 3 ไฟล์คือ glut.h, glut32.lib, และ glut32.dll
2. คัดลอกไฟล์ glut.h ไปไว้ที่ C:\Program Files\Borland\CBuilder5\Include\Gl
3. คัดลอกไฟล์ glut32.lib ไปไว้ที่ C:\Program Files\Borland\CBuilder5\Lib
4. คัดลอกไฟล์ glut32.dll ไปไว้ที่ C:\Windows\system32
5. ดาวน์โหลดไฟล์ template.cpp จาก http://www.kmitl.ac.th/~kwwithaw/Files/template.cpp
6. เปิดไฟล์ template.cpp ขึ้น (หากเปิด C++Builder อยู่ให้เลือก File -> Close All ก่อน) แล้วตอบ Yes เพื่อสร้างโปรเจ็ค
7. เลือก Project -> Add to Project เพื่อ Add ไลบราลีไฟล์ glut32.lib ในข้อ 3 เข้ามาในโปรเจ็ค
8. สามารถเพิ่มคำสั่ง OpenGL ตามที่ต้องการได้ การคอมไพล์และรันให้กด F9

<u>**ระวัง**</u> ไฟล์ glut*.* สำหรับ Visual C++ และ Borland C++ Builder ไม่สามารถใช้แทนกันได้ หากใช้ผิดไฟล์ เมื่อคอมไพล์โปรแกรมจะเกิดความผิดพลาดขึ้น

เอกสารอ้างอิง

[1] S. Heiman, **DXvsOGL**, http://members.cox.net/scottheiman/dxvsogl.htm, 2004.

[2] M. Woo, J. Neider, T. Davis, D. Shreiner, and OpenGL Architecture Review Board, **OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL, Version 1.2 ,** 3rd Ed., Addison-Wesley Pub Co. 1999.

[3] C. Frazier, R. Kempf, and OpenGL Architecture Review Board, **OpenGL(R) Reference Manual: The Official Reference Document to OpenGL, Version 1.1**, 2nd Ed., Addison-Wesley Pub Co. 1997.

[4] **GLU, GLX, & DRI**, http://www.opengl.org/resources/libraries/glx.html, 2004.

[5] **GLUT – The OpenGL Utility Toolkit**, http://www.opengl.org/resources/libraries/glut.html, 2004.

[6] E. W. Weisstein, **Normal Vector**, http://mathworld.wolfram.com/NormalVector.html, 2004.

[7] E. W. Weisstein, **Icosahedron,** http://mathworld.wolfram.com/Icosahedron.html, 2004.

[8] **OpenGL with Visual C++ 2008 Express Edition**, http://www.mrmoen.com/2008/03/30/opengl-with-visual-c-express-edition/, 2009.

[9] **Setting up OpenGL in C++Builder**, http://bdn.borland.com/article/0,1410,10528,00.html, 2009.

[10] N. Robins, **GLUT for Win32**, http://www.xmission.com/~nate/glut.html, 2004.

[11] P. Groves, **Paul's OpenGL Page**, http://home.clara.net/paulyg/ogl.htm, 2004.

การสร้างภาพสามมิติโดยใช้ OpenGL

ชื่อ_____เลขประจำตัว_____ชั้นปี_____

<u>คำสั่ง</u> เขียนอธิบายสิ่งที่ได้ทดลองและผลที่เกิดขึ้นในแต่ละการทดลอง<u>ด้วยความเข้าใจของตนเอง</u>

Part A – Introduction to GL Programming

_____

_____

_____

_____

_____

Part B.1 – Drawing Primitives

_____

_____

_____

_____

_____

_____

Part B.2 – Drawing Quadric Primitives

_____

_____

_____

_____

_____

_____

Part C.1 – Modelview Transformation

_____

_____

_____

_____

_____

_____

Part C.2 – Building the Solar System

Part C.3 – Robot Arm (อธิบายการทำงานของโปรแกรม)

Part D.1 – Light & Material Properties

Part D.2 – Normal Vector

ข้อเสนอแนะสำหรับเอกสารประกอบการทดลองชุดนี้