Introduction to Computer Graphics Homework 1

0616309 王祥任

Abstraction

本次作業一要使用 C++，OpenGL，GLUT 等來完成簡單的日、月、地公轉和自轉的模擬 3D 影像模型。

Program Structure

以下將針對程式的架構進行簡略的說明。

```
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
    glutInitWindowSize(width, height);
    glutInitWindowPosition(0, 0);
    glutCreateWindow("HW1");

    lighting();
    glutDisplayFunc(display);

    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboard);
    glutIdleFunc(idle);
    glutMainLoop();
    return 0;
}
```

在 main 函式中，依流程依序為：

1. 啟動 Glut 工具庫。
2. 設定顯示的模式，使其具有雙 Buffer、RGBA 顯色、Depth Buffer。
3. 設定視窗寬、高和初始生成位置(0,0)。
4. 建立名為「HW1」的視窗。
5. 設定光源。
6. 設定各種 Callback 函式(顯示、視窗大小變動、鍵盤輸入、空閒)。
7. 呼叫 glutMainLoop 函式以適時處理輸入及呼叫對應的 Callback。

```
void lighting()
{
    // enable lighting
    glEnable(GL_LIGHTING);
    //Add directed light
    GLfloat diffuseColor[] = { 1.0f, 1.0f, 1.0f, 1.0f };
    GLfloat ambientColor[] = { 0.5f, 0.5f, 0.5f, 1.0f };
    GLfloat position[] = { 0.0f, 10.0f, 0.0f, 1.0f };
    glEnable(GL_LIGHT0);                              //open light0
    glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuseColor);   //set diffuse color of light0
    glLightfv(GL_LIGHT0, GL_AMBIENT, ambientColor);   //set ambient color of light0
    glLightfv(GL_LIGHT0, GL_POSITION, position);      //set position of light0

    // For Color and Shade // WACODE
    glColorMaterial(GL_FRONT, GL_DIFFUSE);
    glEnable(GL_COLOR_MATERIAL);
}
```

在 lighting 函式中，依流程依序為：

1. 啟動光照及光源「GL_LIGHT0」。
2. 設定光源的漫反光及環境光，並設定光源位置。
3. 設定並啟動「GL_Color_Material」以確保單色材質可以有正常的光影。

在 display 函式中，依流程依序為：

1. 設定 ModelView、Viewport、Projection Matrix。
2. 啟動 cull face, depth test, normalization。
3. 清空 Buffer。
4. 旋轉、移動坐標系以正確繪製日、月、地、地軸。
5. 交換 Buffer 以顯示。

reshape 函式會根據視窗的變動改變 width、height 兩個全域變數，以根據視窗大小適時調整 Viewport Matrix 和 Projection Matrix。

keyboard 函式會根據鍵盤輸入調整某些全域變數，以達到使地球降維、暫停自、公轉等。

idle 函式可以當成在每一幀都會常態觸發的回呼函數，本次主要用作計算時間來調整星體的位置。

Implement － Draw Planet without glutsolidshpere()

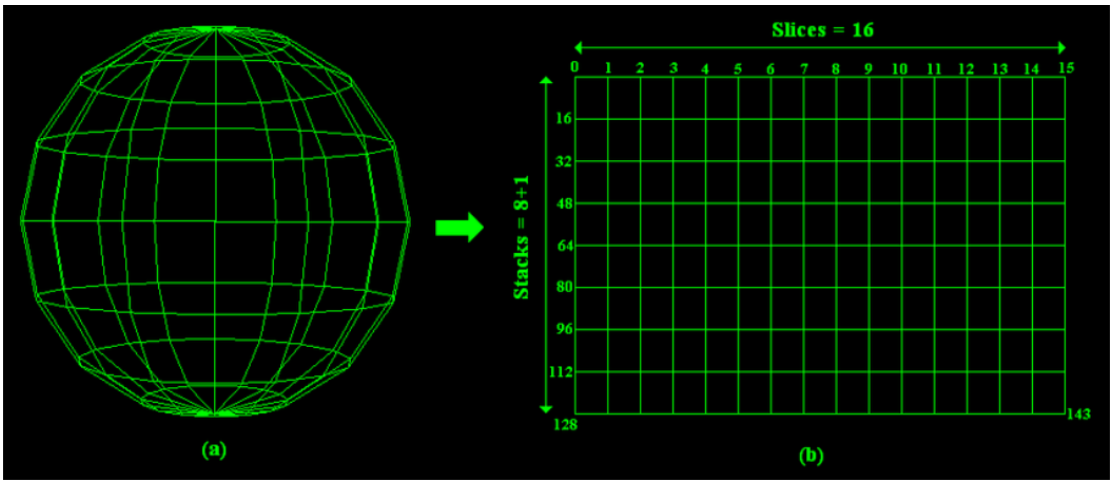


Figure 1: https://www.researchgate.net/figure/a-The-construction-of-a-sphere-with-8-stacks-16-slices-and-144-vertices-Vertices_fig13_266873735

我們其實可以將球體思考為超高面數的多面體，而我們要做的，其實就是將這些表面依照特定的角度去繪製出來就行了。

類似地圖上經緯線的投影，每一個面的四個頂點都是由兩條相鄰經線還有兩

條相鄰緯線互相交錯而成。而兩條相鄰經線間所夾的多個相黏的面便可以視作一面帶(QUAD_STRIP)，而這些面帶接合在一起，外觀上即為面數較低的粗糙球體。

只要將經線數量增加(即 Slices 增加)還有緯線數量增加(即 Stacks 增加)，便能提高這球體的面數，最終便能繪製出具有高面數的擬似光滑球體。

```
int horizon = slices;
int vertical = stacks * 2;

double horizonTheta = double(PI*double(2.0) / double(horizon));
double verticalTheta = double(PI*double(2.0) / double(vertical));

// Draw a "Sphere"
// Or we can say drawing many adjacent "QUAD_STRIP"s
for (int i = 0; i < horizon; i++) {
    // Draw a "QUAD_STRIP"
    glBegin(GL_QUAD_STRIP);
    for (int j = 0; j <= vertical; j++) {
        // Since we have to "Close" the quad_strip
        // we have to overlap the start edge

        // p1 xyz norm
        double p1x = cos(j * verticalTheta) * cos(i * horizonTheta);
        double p1y = sin(j * verticalTheta);
        double p1z = cos(j * verticalTheta) * sin(i * horizonTheta);
        // p2 xyz norm
        double p2x = cos(j * verticalTheta) * cos(double(i+1) * horizonTheta);
        double p2y = sin(j * verticalTheta);
        double p2z = cos(j * verticalTheta) * sin(double(i+1) * horizonTheta);

        // draw p1
        glColor3f(R, G, B);
        glNormal3f(p1x, p1y, p1z); //Specify Vertex's normal vector. Without this, No Shadering
        glVertex3f(radius*p1x, radius*p1y, radius*p1z);

        // draw p2
        glColor3f(R, G, B);
        glNormal3f(p2x, p2y, p2z); //Specify Vertex's normal vector. Without this, No Shadering
        glVertex3f(radius*p2x, radius*p2y, radius*p2z);
    }
    glEnd();
}
```

實作上的細節是以兩層迴圈來旋轉角度，外迴圈執行 Slices 次，內迴圈執行 2*Stacks+1 次(多一次以確保環的貼合)。

內迴圈一次只繪製兩相鄰經線和一條緯線交錯構成的兩個點，下一次的內迴圈則是同一組經線和另一條相鄰緯線交錯構成的兩個點，以上四點便能構成一個面(QUAD)；

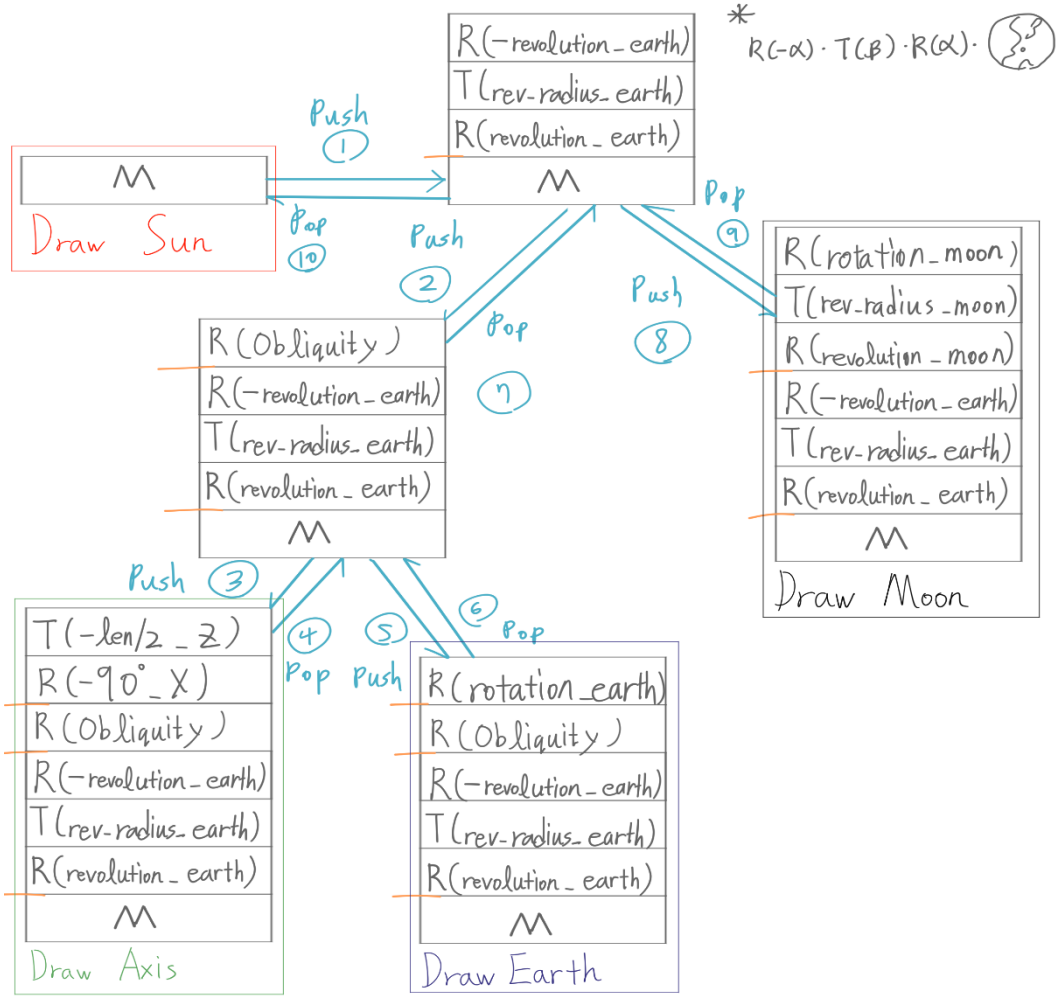
而當外迴圈跑完一次，便完成了一條以複數面構成的環(QUAD_STRIP)的繪製(雖然實際上是由兩條大圓航線，即四條經線所夾構成的，但為了方便解說頂點的繪製，所以將其解釋為兩條相鄰經線所夾)。

外迴圈跑完後，便會有多組(具體而言大約是 Slices/2 組)相鄰併合的複數面構成的環們(QUAD_STRIPs)，就外觀上而言便是一光滑的球體。

Implement － Revolution and Rotation

本次的實作要求以 glPushMatrix()、glPopMatrix()來輔助公轉和自轉的實作。個人是把 Push 當成存檔的動作，而 Pop 就是回到最新一次的存檔時間點，使座標的旋轉、移動回到上一階段。所以針對單一物件獨立的旋轉(例如自轉)或是位移等，最好是以 Push 和 Pop 上下包裹，以確保物件的坐標系正確調整後還能回到上一階段的坐標系繼續繪製其他星體。

具體而言，是以如下樹狀圖並輔以 Push、Pop 來完成自、公轉的坐標系移動。



M：基本前置的 Matrix。

Revolution_Earth：地球繞日公轉的角度。

Revolution_Radius_Earth：地球繞日公轉的半徑。

Rotation_Earth：地球自轉的角度

Obliquity：地球公轉軸的偏斜。

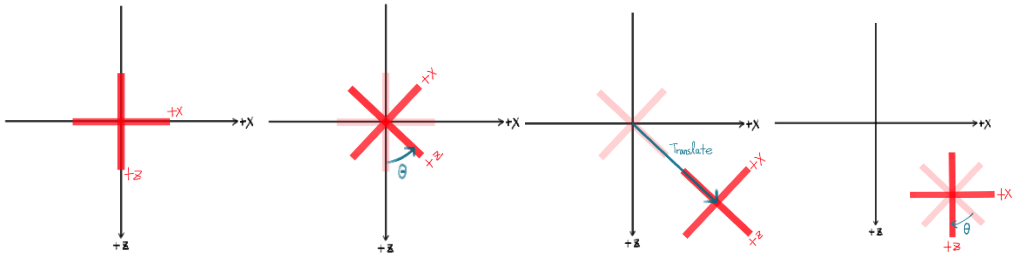
Revolution_Moon：月球繞地公轉的角度。

Revolution_Radius_Moon：月球繞地公轉的半徑。

Rotation_Moon：月球自轉的角度。

而自、公轉角度的計算由三個星體一天的旋轉角度和三個全域變數(日、月、地的時間)決定，由 Idle 函數固定每一幀往前一定的時間，藉此重新計算角度，達到隨著時間前進，自、公轉也有所行進的結果。

Discussion — How it rotate and translate



在我們使用 glRotatef 和 glTranslatef 的時候，我們必須理解到他是直接針對 Local 坐標系做旋轉和偏移，並不是如同常規所想的物件的偏移或是旋轉是參考 Global 坐標系。上圖給出簡單的地球公轉的座標轉移的範例，我們可以看到在旋轉完地球所屬的坐標系後，再使用 glTranslatef 沿著 Z 軸移動，並不是相對 Global 坐標系(黑色線)，而是相對於 Local 坐標系即地球所屬的坐標系(紅色線)。

Conclusion

本次作業可以學到在 OpenGL 中最基本的旋轉、位移等操作，並了解其實作上和平時理解的差異，以及了解 OpenGL 是如何繪製物體，並以大量平面繪製出球體。