Report of Project 2

1. Algorithms Summary

In order to draw objects that appear to be realistic, besides modeling the imaging process of a camera/eye, we also need to simulate the interactions between light and surfaces to determine colors of pixels and how to render only visible surfaces as hidden surfaces are not visible. In this assignment, the main task is to implement the Phong illumination model and apply it to scenes using different shading models. Z-buffer is also used to eliminate hidden surfaces.

Z-Buffer Algorithm: We keep two buffers, one for the colors (frame buffer) and one for the depth (depth buffer)

- Frame buffer is initialized to be the background color;
- Depth buffer is initialized to the depth of the farthest points;
- For each surface to be drawn, we can scan-line it;
  - At each pixel, we only update its color in the frame buffer if the current point is closer than the previous one by comparing the depth value in the depth buffer.

Flat shading: We compute the illumination or color only at one point on the polygonal surface. Typically at the centroid of the polygon. It provides a reasonable approximation if the illumination does not change significantly in the polygonal surface.

Smoothe shading: For a triangular (or polygonal) mesh, the color or illumination is computed at each vertex. Then the colors are interpolated using barycentric coordinates (or linearly for a polygonal surface).

Phong shading: Instead of interpolating colors, we can interpolate the normals, we then use the normal at each pixel to compute the illumination at each pixel. We can also use barycentric coordinates for interpolation.

$$N_p = \alpha N_1 + \beta N_2 + \gamma N_3$$

$$c_r = c_{r,r}(c_{a,r} + \sum_{i=1}^{n} c_{l,r}^{(i)} \max(0, n \bullet l^{(i)})) + c_{p,r} \sum_{i=1}^{n} c_{l,r}^{(i)} \max(0, h^{(i)} \bullet n)^p$$

$$c_g = c_{r,g}(c_{a,g} + \sum_{i=1}^{n} c_{l,g}^{(i)} \max(0, n \bullet l^{(i)})) + c_{p,g} \sum_{i=1}^{n} c_{l,g}^{(i)} \max(0, h^{(i)} \bullet n)^p$$

$$c_b = c_{r,b}(c_{a,b} + \sum_{i=1}^{n} c_{l,b}^{(i)} \max(0, n \bullet l^{(i)})) + c_{p,b} \sum_{i=1}^{n} c_{l,b}^{(i)} \max(0, h^{(i)} \bullet n)^p$$

Phong Illumination Model:

Computing Depth Values:

$x_{min} = \text{floor}(x_i)$
$x_{max} = \text{ceiling}(x_i)$
$y_{min} = \text{floor}(y_i)$
$y_{max} = \text{ceiling}(y_i)$
for $y = y_{min}$ **to** y_{max} **do**
  for $x = x_{min}$ **to** x_{max} **do**
    $\alpha = f_{12}(x, y) / f_{12}(x_0, y_0)$
    $\beta = f_{20}(x, y) / f_{20}(x_1, y_1)$
    $\gamma = f_{01}(x, y) / f_{01}(x_2, y_2)$
    if ($\alpha > 0$ and $\beta > 0$ and $\gamma > 0$) **then**
      $z = \alpha z_0 + \beta z_1 + \gamma z_2$
      ...

2.The implementation

Some implementation are showed in last Project, such as draw floor, draw axis, matrix and vector calculating. So I will only show the implementation I do this time.

In lab3.c

Illumination:

```
void Illumination(double normal[],double diffuse[],double specular[],double illuPoint[])
{
	int i,l,d;
	illuColor[0] = illuColor[1] = illuColor[2] = 0;
	/*ambient*/
	for(i = 0; i < 3; i++){
		illuColor[i] += diffuse[i] * thescene.ambient[i];
	}

	for(l = 0; l < thescene.nlights; l++){
		for(d = 0; d < thescene.lights[l].ndirections; d++){
			double direction[3] = {thescene.lights[l].directions[d][0],

thescene.lights[l].directions[d][1],

thescene.lights[l].directions[d][2]};
			vecUnitization(direction,direction);

			double eye[3] = {vcamera.eye.xyzw[0]-illuPoint[0],vcamera.eye.xyzw[1]-illuPoint[1],vcamera.eye.xyzw[2]-illuPoint[2]};
			vecUnitization(eye,eye);
			double half[3] = {eye[0]+direction[0],eye[1]+direction[1],eye[2]+direction[2]};
			vecUnitization(half,half);

			/*diffuse*/
			for(i = 0; i < 3; i++){
				illuColor[i] += diffuse[i] * thescene.lights[l].light[i] * vecDotProduct(normal,direction);

			}
			/*specular*/
			for(i = 0; i < 3; i++){
				illuColor[i] += specular[i] * thescene.lights[l].light[i] * pow(vecDotProduct(normal,half),specular[3]);

			}
		}
	}

	for(i = 0; i < 3; i++){
		if(illuColor[i]>1)
			illuColor[i] = 1;
	}
}
```

Vector dot Product

```
double vecDotProduct(double firstVec[], double secondVec[])
{
	double product;
	product = firstVec[0]*secondVec[0]+firstVec[1]*secondVec[1]+firstVec[2]*secondVec[2];
```

```
	if(product < 0)
		product = 0;
	return product;
}
```

Rendering and flat shading, smooth shading, phong shading:

```
void setBuffer(COLOR_VERTEX colorVertices[MAXLINELENGTH],double matrixFinal[][4],int nM,double d[],double s[],int shading,double mInverse[][4])
{
	int k;
	for(k = 0; k < thescene.mesh[nM].npolygons; k++){
		COLOR_VERTEX vertices[3] = {colorVertices[thescene.mesh[nM].polygons[k].num[0]],

					colorVertices[thescene.mesh[nM].polygons[k].num[1]],

					colorVertices[thescene.mesh[nM].polygons[k].num[2]]};
		int i=0;
		for(i = 0; i < 3; i++){
	matrixApply(matrixFinal,vertices[i].xyzw);
		}
		toScreen(vertices[0].xyzw,vertices[1].xyzw,vertices[2].xyzw);

	triRendering(vertices[0].xyzw,vertices[1].xyzw,vertices[2].xyzw,vertices[0].rgba,vertices[1].rgba,vertices[2].rgba,d,s,shading,mInverse);

	}
}

void triRendering(double v0[],double v1[],double v2[],float c0[],float c1[],float c2[],double d[],double s[],int shading,double mInverse[][4])
{
	/*triangle constant*/
	int xmax,xmin,ymax,ymin;
	double l0_12,l1_02,l2_01;
	double x_incr_alpha,x_incr_beta,x_incr_gamma;
	double y_incr_alpha,y_incr_beta,y_incr_gamma;
	double alpha0,beta0,gamma0,flag_12,flag_02,flag_01;

	xmin=min(min(v0[0],v1[0]),v2[0]);
	xmax=max(max(v0[0],v1[0]),v2[0]);
	ymin=min(min(v0[1],v1[1]),v2[1]);
	ymax=max(max(v0[1],v1[1]),v2[1]);

	/*const*/
	l0_12 = decision(v1,v2,v0[0],v0[1]);
	l1_02 = decision(v0,v2,v1[0],v1[1]);
	l2_01 = decision(v0,v1,v2[0],v2[1]);

	x_incr_alpha = (v1[1]-v2[1])/l0_12;
	x_incr_beta = (v0[1]-v2[1])/l1_02;
	x_incr_gamma = (v0[1]-v1[1])/l2_01;

	y_incr_alpha = (v2[0]-v1[0])/l0_12;
	y_incr_beta = (v2[0]-v0[0])/l1_02;
	y_incr_gamma = (v1[0]-v0[0])/l2_01;
```

```
	alpha0 = decision(v1,v2,xmin,ymin)/l0_12;
	beta0 = decision(v0,v2,xmin,ymin)/l1_02;
	gamma0 = decision(v0,v1,xmin,ymin)/l2_01;

	/* (-1,-1) or (-2,-1) */
	flag_12 = decision(v1,v2,-1,-1);
	flag_02 = decision(v0,v2,-1,-1);
	flag_01 = decision(v0,v1,-1,-1);

	if(flag_12 == 0)
		flag_12 = decision(v1,v2,-2,-1);
	if(flag_02 == 0)
		flag_02 = decision(v0,v2,-2,-1);
	if(flag_01 == 0)
		flag_01 = decision(v0,v1,-2,-1);

	int i,x,y;
	double alpha,beta,gamma;
	for (y = ymin; y <= ymax; y++){
		alpha = alpha0;
		beta = beta0;
		gamma = gamma0;
		for (x = xmin; x <= xmax; x++){
			if(alpha >= 0 && beta >= 0 && gamma >= 0){
				if( (alpha > 0 || l0_12 * flag_12 >0) && (beta > 0 || l1_02 * flag_02 >0) && (gamma > 0 ||
l2_01 * flag_01 > 0)){
					double z = alpha * v0[2] + beta * v1[2] + gamma * v2[2];
					if(z < buffer[y*thescene.screen_w+x].z){
						if(shading == 0){
							for(i = 0; i < 3; i++){
								buffer[y*thescene.screen_w+x].rgba[i] = illuColor[i];

							}
							buffer[y*thescene.screen_w+x].z = z;
						}
						else if(shading == 1){
							for(i = 0; i < 3; i++){
								buffer[y*thescene.screen_w+x].rgba[i] = alpha * c0[i] + beta * c1[i]
+ gamma * c2[i];
							}
						 buffer[y*thescene.screen_w+x].z = z;
						}
						else{
							double normal[3];
							for(i = 0; i < 3; i++){
								normal[i] = alpha * c0[i] + beta * c1[i] + gamma * c2[i];
							}
							vecUnitization(normal,normal);

							double w = alpha * v0[3] + beta * v1[3] + gamma * v2[3];
							double p[4] = {x*w,y*w,z,w};

							matrixApply(mInverse,p);
							Illumination(normal,d,s,p);
							for(i = 0; i < 3; i++){
								buffer[y*thescene.screen_w+x].rgba[i] = illuColor[i];

							}
```

```
                                        buffer[y*thescene.screen_w+x].z = z;
                                    }
                                }
                            }
                        }
                        alpha += x_incr_alpha;
                        beta += x_incr_beta;
                        gamma += x_incr_gamma;

                    }
                    alpha0 += y_incr_alpha;
                    beta0 += y_incr_beta;
                    gamma0 += y_incr_gamma;
                }
            }
```

In render function() //shading

```
else if(ascene->identities[i].instr[j] == MESH_KEY){
                                double tM[4][4];
                                matrixInitial(tM);
                                matrixMultiply(mTransform,tM,0);
                                matrixMultiply(matrixFinal,tM,0);
                                int k,l,d;

                                /*apply transform matrix to all vertices in world coordinates*/
                                COLOR_VERTEX colorVertices[ascene->mesh[nM].nvertices];
                                for(k = 0; k < ascene->mesh[nM].nvertices; k++){
                                    colorVertices[k] = ascene->mesh[nM].vertices[k];
                                    matrixApply(mTransform,colorVertices[k].xyzw);
                                }

                                //flat shading
                                if(ascene->mesh[nM].shading == 0){
                                    for(k = 0; k < ascene->mesh[nM].npolygons; k++){
                                        COLOR_VERTEX vertices[3] =
{colorVertices[ascene->mesh[nM].polygons[k].num[0]],

                    colorVertices[ascene->mesh[nM].polygons[k].num[1]],

                    colorVertices[ascene->mesh[nM].polygons[k].num[2]]};

                                        double normal[3];

        triangleNormal(vertices[0].xyzw,vertices[1].xyzw,vertices[2].xyzw,normal);
                                        double center[3];
                                        center[0] = (vertices[0].xyzw[0] + vertices[1].xyzw[0] +
vertices[2].xyzw[0])/(double)3;
                                        center[1] = (vertices[0].xyzw[1] + vertices[1].xyzw[1] +
vertices[2].xyzw[1])/(double)3;
                                        center[2] = (vertices[0].xyzw[2] + vertices[1].xyzw[2] +
vertices[2].xyzw[2])/(double)3;

        Illumination(normal,ascene->mesh[nM].diffuse,ascene->mesh[nM].specular,center);
        int i;
                                        for(i = 0; i < 3; i++){
        matrixApply(matrixFinal,vertices[i].xyzw);
                                        }
                                        toScreen(vertices[0].xyzw,vertices[1].xyzw,vertices[2].xyzw);
```

```
triRendering(vertices[0].xyzw,vertices[1].xyzw,vertices[2].xyzw,0,0,0,0,0,0,mInverse);

                                    }
                                }

                                //phong shading
                                else if (ascene->mesh[nM].shading == 2){
                                    double vertex_normal[ascene->mesh[nM].nvertices][3];
                                    for(k = 0; k < ascene->mesh[nM].nvertices; k++){
                                    double composeNormal[3] = {0,0,0};
                                        for(l = 0; l < ascene->mesh[nM].npolygons; l++){
                                            for(d = 0; d < ascene->mesh[nM].polygons[l].nvertices; d++){
                                                if(ascene->mesh[nM].polygons[l].num[d] == k){
                                                COLOR_VERTEX vertices[3] =
{colorVertices[ascene->mesh[nM].polygons[l].num[0]],

                                colorVertices[ascene->mesh[nM].polygons[l].num[1]],

                                colorVertices[ascene->mesh[nM].polygons[l].num[2]]};
                                                double normal[3];

    triangleNormal(vertices[0].xyzw,vertices[1].xyzw,vertices[2].xyzw,normal);
                                                                    composeNormal[0] += normal[0];
                                                        composeNormal[1] += normal[1];
                                                        composeNormal[2] += normal[2];
                                                }
                                            }
                                        }
    vecUnitization(composeNormal,composeNormal);
                                    int i=0;
                                    for(i = 0; i < 3; i++){
                                        colorVertices[k].rgba[i] = composeNormal[i];
                                        }
                                    }
    setBuffer(colorVertices,matrixFinal,nM,ascene->mesh[nM].diffuse,ascene->mesh[nM].specular,2,mInverse);

                                }
                                //smooth shading
                                else{
                                   for(k = 0; k < ascene->mesh[nM].nvertices; k++){
                                     double composeNormal[3] = {0,0,0};
                                     for(l = 0; l < ascene->mesh[nM].npolygons; l++){
                                         for(d = 0; d < ascene->mesh[nM].polygons[l].nvertices; d++){

if(ascene->mesh[nM].polygons[l].num[d] == k){
                                                                COLOR_VERTEX vertices[3] =
{colorVertices[ascene->mesh[nM].polygons[l].num[0]],

                                colorVertices[ascene->mesh[nM].polygons[l].num[1]],

                                colorVertices[ascene->mesh[nM].polygons[l].num[2]]};
                                                 double normal[3];
                                triangleNormal(vertices[0].xyzw,vertices[1].xyzw,vertices[2].xyzw,normal);
                                                                    composeNormal[0] += normal[0];
                                                        composeNormal[1] += normal[1];
            composeNormal[2] += normal[2];
                                                }
```

```
                                        }
                                    }
        vecUnitization(composeNormal,composeNormal);
        Illumination(composeNormal,ascene->mesh[nM].diffuse,ascene->mesh[nM].specular,colorVertices[k].xyzw)
;

        colorVertices[k].rgba[0] = illuColor[0];
        colorVertices[k].rgba[1] = illuColor[1];
        colorVertices[k].rgba[2] = illuColor[2];
                                    }
        setBuffer(colorVertices,matrixFinal,nM,0,0,1,mInverse);
                                }
                                glLineWidth(ascene->mesh[nM].width);
                                glBegin(GL_POINTS);
                                for(k = 0; k < ascene->screen_h; k++){
                                    for(l = 0; l < ascene->screen_w; l++){
                                        if(buffer[k*(ascene->screen_w)+l].z < 9999){
                                glColor3f(buffer[k*(ascene->screen_w)  +  l].rgba[0],buffer[k*(ascene->screen_w)  +
l].rgba[1],buffer[k*(ascene->screen_w) + l].rgba[2]);
        glVertex2i(l,k);
                                        }
                                    }
                                }
                                glEnd();
                                nM++;

                    }
                }
        }
```

Variable: mInverse(Inverse matrix)

```
            double persInverse[4][4];
            double orthoInverse[4][4];
            matrixInitial(persInverse);
            matrixInitial(orthoInverse);

            persInverse[0][0] = (double)1/nearPers;
            persInverse[1][1] = (double)1/nearPers;
            persInverse[2][2] = 0;
            persInverse[2][3] = 1;
            persInverse[3][2] = (double)-1/(nearPers*farPers);
            persInverse[3][3] = (nearPers+farPers)/(nearPers*farPers);

            orthoInverse[0][0] = right;
            orthoInverse[1][1] = top;
            orthoInverse[2][2] = (nearPers-farPers)/(double)2;
            orthoInverse[2][3] = (nearPers+farPers)/(double)2;

            matrixMultiply(mInverse,persInverse,1);
            matrixMultiply(mInverse,orthoInverse,1);

            double vpInverse[4][4];
            matrixInitial(vpInverse);
            vpInverse[0][0] = (double)2/ascene->screen_w;
            vpInverse[0][3] = (double)(1-ascene->screen_w)/ascene->screen_w;
            vpInverse[1][1] = (double)2/ascene->screen_h;
            vpInverse[1][3] = (double)(1-ascene->screen_h)/ascene->screen_h;
            matrixMultiply(mInverse,vpInverse,1);
```

SSD.c(in red):

```
#include <stdio.h>
#define SSD_UTIL_SOURCE_CODE
#include "SSD_util.h"
#define MAXLINELENGTH 1000
#define MAXLABELLEN          16

struct ssd_keyword keyword_table[] =
   {{SCREEN_KEY, "screen", 5},
    {COLOR_KEY, "color", 3},
    {LINE_KEY, "line", 1},
    {VERTEX_KEY, "vertex", 3},
    {POLYLINE_KEY, "polyline", 2},
    {CIRCLE_KEY, "circle", 3},
    {ARC_KEY, "arc", 5},
    {SAVE_KEY, "save", 1},
    {TRIANGLE_KEY, "triangle",0},

    {EYE_KEY, "eye",3},
    {GAZE_KEY, "gaze",3},
    {UPVECTOR_KEY, "upvector",3},
    {ORTHO_KEY, "ortho",4},
    {PERSP_KEY, "perspective",3},
    {FLOOR_KEY, "floor",5},
    {AXIS_KEY, "axis",2},
    {IDENTITY_KEY, "identity",0},
    {TRANSLATE_KEY, "translate",3},
    {ROTATE_KEY, "rotate",3},
    {SCALE_KEY, "scale",3},
    {MESH_KEY, "mesh",2},
    {LIGHT_KEY, "light",3},
    {AMBIENT_KEY, "ambient",3},
    {SHADING_KEY, "shading",1},
    {DIFFUSE_KEY, "diffuse",3},
    {SPECULAR_KEY, "specular",3},
    {-1,"unknown", 0}};

int match_Keyword(char *keyword, int *npara)
{
   int i;
   *npara = 0;
   for (i=0; keyword_table[i].key_id != -1; i++) {
      if (strcmp(keyword_table[i].name, keyword) == 0) {
         *npara = keyword_table[i].npara;
         return keyword_table[i].key_id;
      }
   }
   return -1;
}

int readAndParse(FILE *inFilePtr, char *keyword, char *arg0,
      char *arg1, char *arg2, char *arg3, char *arg4)
{
   static char line[MAXLINELENGTH];
   char *ptr = line;

   /* delete prior values */
```

```
   keyword[0] = arg0[0] = arg1[0] = arg2[0] = arg3[0] = arg4[0] = '\0';
   if (feof(inFilePtr)) return(0);
   /* read the line from the SSD file */
   while (1) {
      if (fgets(line, MAXLINELENGTH, inFilePtr) == NULL) {
         /* reached end of file */
         return(0);
      }

      if (feof(inFilePtr)==0) {
         /* check for line too long (by looking for a \n) */
         if (strchr(line, '\n') == NULL) {
       /* line too long */
       printf("error: line too long\n");
       exit(1);
         }
      }
      if (line[0] != '#') break;
   }
   /*
    * Parse the line.
    */
         if(line[0] == 32){
   sscanf(line,

"%*[\t\n ]%[^\t\n ]%*[\t\n ]%[^\t\n ]%*[\t\n ]%[^\t\n ]%*[\t\n ]%[^\t\n ]%*[\t\n ]%[^\t\n ]%*[\t\n ]%[^\t\n ]%*[\t\
n ]",
        keyword, arg0, arg1, arg2, arg3, arg4);
        }
        else{
   sscanf(line,
   "%[^\t\n ]%*[\t\n ]%[^\t\n ]%*[\t\n ]%[^\t\n ]%*[\t\n ]%[^\t\n ]%*[\t\n ]%[^\t\n ]%*[\t\n ]%[^\t\n ]%*[\t\n ]",
        keyword, arg0, arg1, arg2, arg3, arg4);
        }
   return strlen(line);
}

int Read_SSD_Scene(char *fname, SCENE *ascene, CAMERA *acamera, char *saved_fname)
{
   char keyword[MAXLINELENGTH],    arg0[MAXLINELENGTH],
      arg1[MAXLINELENGTH], arg2[MAXLINELENGTH], arg3[MAXLINELENGTH];
   char arg4[MAXLINELENGTH];
        int ident = 0;
   FILE *fp;

   /* We first set all the default values */
   RGB_COLOR fcolor, vcolor;
   int ind, ii, num_ver, key_id, key_id1, npara;
   int shading = 0;
   double diffuse[3] = {0,0,0};
   double specular[3] = {0,0,0};

   ascene->screen_w = 600;
   ascene->screen_h = 400;
   /* The default color is white */
   ascene->bcolor.rgba[0] = 1.0;
   ascene->bcolor.rgba[1] = 1.0;
   ascene->bcolor.rgba[2] = 1.0;
```

```
ascene->bcolor.rgba[3] = 1.0;
fcolor.rgba[0] = 0.0; fcolor.rgba[1] = 0.0;
fcolor.rgba[2] = 0.0; fcolor.rgba[3] = 1.0;
ascene->nlines = 0;
ascene->npolylines = 0;
ascene->ntriangles = 0;
//
ascene->nidentities = 0;
ascene->pjType = 0; //0:orthographic 1:perspective
ascene->isAxis = 0; //0 represents noAxis
ascene->nlights =0;
int ntranslate = 0;
int nrotate = 0;
int nscale = 0;
int nmesh = 0;
fp = fopen(fname,"rb");
if (fp == NULL) {
   fprintf(stderr,"%s:%d: Can not open SSD file <%s>.\n",
                __FILE__, __LINE__, fname);
   return -1;
}
while (readAndParse(fp, keyword, arg0, arg1, arg2, arg3, arg4) > 0) {
   if (keyword[0] == '\0') {
      /* We simply all blank lines */
      continue;
   }
   key_id = match_Keyword(keyword, &npara);
   switch(key_id) {
   case LINE_KEY:
      ascene->nlines++;
      break;
   case POLYLINE_KEY:
      ascene->npolylines++;
      break;
   case TRIANGLE_KEY:
      ascene->ntriangles++;
      break;

   case IDENTITY_KEY:
      ascene->nidentities++;
      break;
   case TRANSLATE_KEY:
      ntranslate++;
      break;
   case SCALE_KEY:
      nscale++;
      break;
   case ROTATE_KEY:
      nrotate++;
      break;
   case MESH_KEY:
      nmesh++;
      break;
   case LIGHT_KEY:
      ascene->nlights++;
      break;
   }
}
```

```
printf("There are %d lines, %d polylines, and %d triangles in %s.\n",
      ascene->nlines, ascene->npolylines, ascene->ntriangles,
      fname);
/* We rewind the file to the very beginning to read the file again */
rewind(fp);
ascene->lines = (LINE *)malloc(sizeof(LINE) * ascene->nlines);
ascene->polylines = (POLYLINE *)malloc(sizeof(POLYLINE) *
                                ascene->npolylines);
ascene->triangles = (TRIANGLE *)malloc(sizeof(TRIANGLE) *
                                ascene->ntriangles);
ascene->identities = (IDENTITY *)malloc(sizeof(IDENTITY) * ascene->nidentities);
ascene->translate = (TRANSLATE *)malloc(sizeof(TRANSLATE) * ntranslate);
ascene->scale = (SCALE *)malloc(sizeof(SCALE) * nscale);
ascene->rotate = (ROTATE *)malloc(sizeof(ROTATE) * nrotate);
ascene->mesh = (MESH *)malloc(sizeof(MESH) * nmesh);
ascene->lights = (LIGHT *)malloc(sizeof(LIGHT) * ascene->nlights);

ascene->nlines = 0;
ascene->npolylines = 0;
ascene->ntriangles = 0;
ascene->nlights = 0;
ascene->nidentities = 0;
ntranslate = 0;
nscale = 0;
nrotate = 0;
nmesh = 0;
int instr_ind = 0;
int brk = 0;
int readAndParseResult = 0;

while (readAndParse(fp, keyword, arg0, arg1, arg2, arg3, arg4) > 0) {
   if (keyword[0] == '\0') {
      /* We simply all blank lines */
      continue;
   }
   key_id = match_Keyword(keyword, &npara);
   switch(key_id) {
   case SCREEN_KEY:
     ascene->screen_w = atoi(arg0);
     ascene->screen_h = atoi(arg1);
     ascene->bcolor.rgba[0] = atoi(arg2)/255.0;
     ascene->bcolor.rgba[1]    = atoi(arg3)/255.0;
     ascene->bcolor.rgba[2]    = atoi(arg4)/255.0;
     ascene->bcolor.rgba[3] = 1.0;
     break;

   case COLOR_KEY:
      /* We read the color */
      fcolor.rgba[0] = atoi(arg0)/255.0;    fcolor.rgba[1] = atoi(arg1)/255.0;
      fcolor.rgba[2] = atoi(arg2)/255.0;
      break;
   case LINE_KEY:
      ind = ascene->nlines;
      ascene->lines[ind].width = atof(arg0);
      /* We set the default colors */
      vcolor = fcolor;
      num_ver = 0;
      while (readAndParse(fp, keyword, arg0, arg1, arg2, arg3, arg4) > 0) {
```

```
            key_id1 = match_Keyword(keyword, &npara);
            switch(key_id1) {
            case VERTEX_KEY:
               ascene->lines[ind].vertices[num_ver].xyzw[0] = atof(arg0);
           ascene->lines[ind].vertices[num_ver].xyzw[1] = atof(arg1);
               ascene->lines[ind].vertices[num_ver].xyzw[2] = atof(arg2);
           memcpy(ascene->lines[ind].vertices[num_ver].rgba,
                vcolor.rgba, sizeof(vcolor.rgba));

#if defined(DEBUG_FLAG)
               printf("Point %d %d with color %6.4f %6.4f %6.4f\n",
                        (int)ascene->lines[ind].vertices[num_ver].xyzw[0],
                (int)ascene->lines[ind].vertices[num_ver].xyzw[1],
                ascene->lines[ind].vertices[num_ver].rgba[0],
                ascene->lines[ind].vertices[num_ver].rgba[1],
                ascene->lines[ind].vertices[num_ver].rgba[2]);
#endif
               num_ver ++;
               break;
            case COLOR_KEY:
           vcolor.rgba[0] = atoi(arg0)/255.0; vcolor.rgba[1] = atoi(arg1)/255.0;
               vcolor.rgba[2] = atoi(arg2)/255.0;
               break;
            default:
               printf("%s:%d Line (%s %s %s %s %s %s) ignored.\n",
                        __FILE__, __LINE__, keyword, arg0, arg1, arg2,
                        arg3, arg4);
            }
            if (num_ver == 2) {
               break;
            }
         }
         ascene->nlines++;
         break;
      case POLYLINE_KEY:
         ind = ascene->npolylines;
         ascene->polylines[ind].nvertices = atoi(arg0);
         ascene->polylines[ind].width = atof(arg1);
         ascene->polylines[ind].vertices =
       (COLOR_VERTEX *)malloc(sizeof(COLOR_VERTEX) *
                              ascene->polylines[ind].nvertices);
         vcolor = fcolor;
         num_ver = 0;
         while (readAndParse(fp, keyword, arg0, arg1, arg2, arg3, arg4) > 0) {
            key_id1 = match_Keyword(keyword, &npara);
            switch(key_id1) {
            case VERTEX_KEY:
               ascene->polylines[ind].vertices[num_ver].xyzw[0] = atof(arg0);
           ascene->polylines[ind].vertices[num_ver].xyzw[1] = atof(arg1);
               ascene->polylines[ind].vertices[num_ver].xyzw[2] = atof(arg2);
           memcpy(ascene->polylines[ind].vertices[num_ver].rgba,
                vcolor.rgba, sizeof(vcolor.rgba));

#if defined(DEBUG_FLAG)
               printf("Point %d %d with color %6.4f %6.4f %6.4f\n",
                        (int)ascene->polylines[ind].vertices[num_ver].xyzw[0],
                (int)ascene->polylines[ind].vertices[num_ver].xyzw[1],
                ascene->polylines[ind].vertices[num_ver].rgba[0],
```

```
                ascene->polylines[ind].vertices[num_ver].rgba[1],
                ascene->polylines[ind].vertices[num_ver].rgba[2]);
#endif
                num_ver ++;
                break;
              case COLOR_KEY:
             vcolor.rgba[0] = atoi(arg0)/255.0; vcolor.rgba[1] = atoi(arg1)/255.0;
                vcolor.rgba[2] = atoi(arg2)/255.0;
                break;
              default:
                printf("%s:%d Line (%s %s %s %s %s) ignored.\n",
                        __FILE__, __LINE__, keyword, arg0, arg1, arg2,
                        arg3, arg4);
              }
              if (num_ver >= ascene->polylines[ind].nvertices) {
                break;
              }
            }
            ascene->npolylines++;
            break;

          case TRIANGLE_KEY:
            ind = ascene->ntriangles;
            vcolor = fcolor;
            num_ver = 0;
            while (readAndParse(fp, keyword, arg0, arg1, arg2, arg3, arg4) > 0) {
              key_id1 = match_Keyword(keyword, &npara);
              switch(key_id1) {
              case VERTEX_KEY:
                ascene->triangles[ind].vertices[num_ver].xyzw[0] = atof(arg0);
             ascene->triangles[ind].vertices[num_ver].xyzw[1] = atof(arg1);
                ascene->triangles[ind].vertices[num_ver].xyzw[2] = atof(arg2);
             memcpy(ascene->triangles[ind].vertices[num_ver].rgba,
                  vcolor.rgba, sizeof(vcolor.rgba));
#if defined(DEBUG_FLAG)
                printf("Point %d %d with color %6.4f %6.4f %6.4f\n",
                        (int)ascene->triangles[ind].vertices[num_ver].xyzw[0],
                (int)ascene->triangles[ind].vertices[num_ver].xyzw[1],
                ascene->triangles[ind].vertices[num_ver].rgba[0],
                ascene->triangles[ind].vertices[num_ver].rgba[1],
                ascene->triangles[ind].vertices[num_ver].rgba[2]);
#endif
                num_ver ++;
                break;
              case COLOR_KEY:
             vcolor.rgba[0] = atoi(arg0)/255.0; vcolor.rgba[1] = atoi(arg1)/255.0;
                vcolor.rgba[2] = atoi(arg2)/255.0;
                break;
              default:
                printf("%s:%d Line (%s %s %s %s %s) ignored.\n",
                        __FILE__, __LINE__, keyword, arg0, arg1, arg2,
                        arg3, arg4);
              }
              if (num_ver >= 3) {
                break;
              }
            }
            ascene->ntriangles++;
```

```
    break;


case EYE_KEY:
    acamera->eye.xyzw[0] = atof(arg0);
    acamera->eye.xyzw[1] = atof(arg1);
    acamera->eye.xyzw[2] = atof(arg2);
    break;

case GAZE_KEY:
    acamera->gaze.xyzw[0] = atof(arg0);
    acamera->gaze.xyzw[1] = atof(arg1);
    acamera->gaze.xyzw[2] = atof(arg2);
    break;

case UPVECTOR_KEY:
    acamera->upVector.xyzw[0] = atof(arg0);
    acamera->upVector.xyzw[1] = atof(arg1);
    acamera->upVector.xyzw[2] = atof(arg2);
    break;


case ORTHO_KEY:
    ascene->ortho.right = atof(arg0);
    ascene->ortho.top = atof(arg1);
    ascene->ortho.near = atof(arg2);
    ascene->ortho.far = atof(arg3);
    break;

case PERSP_KEY:
    ascene->pjType = 1;
    ascene->persp.near = atof(arg1);
    ascene->persp.far = atof(arg2);
    ascene->persp.angle = atof(arg0);
    break;

case FLOOR_KEY:
    ascene->floor.size = atof(arg0);
    ascene->floor.xmin = atof(arg1);
    ascene->floor.xmax = atof(arg2);
    ascene->floor.ymin = atof(arg3);
    ascene->floor.ymax = atof(arg4);
    ascene->floor.color = fcolor;
    break;

case AXIS_KEY:
    ascene->isAxis = 1;
    ascene->axis.width = atof(arg0);
    ascene->axis.length = atof(arg1);
    break;

case AMBIENT_KEY:
  ascene->ambient[0] = atof(arg0);
  ascene->ambient[1] = atof(arg1);
  ascene->ambient[2] = atof(arg2);
  break;

case SHADING_KEY:
```

```
            shading = atof(arg0);
          break;

        case DIFFUSE_KEY:
          diffuse[0] = atof(arg0);
          diffuse[1] = atof(arg1);
          diffuse[2] = atof(arg2);
          break;

        case SPECULAR_KEY:
          specular[0] = atof(arg0);
          specular[1] = atof(arg1);
          specular[2] = atof(arg2);
                specular[3] = atof(arg3);
          break;

        case LIGHT_KEY:
                {
          ascene->lights[ascene->nlights].light[0] = atof(arg0);
          ascene->lights[ascene->nlights].light[1] = atof(arg1);
          ascene->lights[ascene->nlights].light[2] = atof(arg2);

                int result_readAndParse = readAndParse(fp, keyword, arg0, arg1, arg2, arg3, arg4);
                key_id = match_Keyword(keyword, &npara);
                     ascene->lights[ascene->nlights].ndirections = 0;
                     while(key_id == VERTEX_KEY)
                     {
                          ascene->lights[ascene->nlights].directions[ascene->lights[ascene->nlights].ndirections][0]
= atof(arg0);
                          ascene->lights[ascene->nlights].directions[ascene->lights[ascene->nlights].ndirections][1]
= atof(arg1);

        ascene->lights[ascene->nlights].directions[ascene->lights[ascene->nlights].ndirections++][2] = atof(arg2);
                          result_readAndParse = readAndParse(fp, keyword, arg0, arg1, arg2, arg3, arg4);
                       key_id = match_Keyword(keyword, &npara);
                     }
                     fseek(fp,-result_readAndParse,SEEK_CUR);
                     ascene->nlights++;
                }
          break;

        case IDENTITY_KEY:
           readAndParseResult = readAndParse(fp, keyword, arg0, arg1, arg2, arg3, arg4);
             while (readAndParseResult > 0) {
               key_id1 = match_Keyword(keyword, &npara);
               switch(key_id1) {
             case TRANSLATE_KEY:
               ascene->translate[ntranslate].xyz[0] = atof(arg0);
               ascene->translate[ntranslate].xyz[1] = atof(arg1);
               ascene->translate[ntranslate++].xyz[2] = atof(arg2);
               ascene->identities[ascene->nidentities].instr[instr_ind++] = TRANSLATE_KEY;
               break;
             case ROTATE_KEY:
               ascene->rotate[nrotate].angle = atof(arg0);
               ascene->rotate[nrotate].xyz[0] = atof(arg1);
               ascene->rotate[nrotate].xyz[1] = atof(arg2);
               ascene->rotate[nrotate++].xyz[2] = atof(arg3);
               ascene->identities[ascene->nidentities].instr[instr_ind++] = ROTATE_KEY;
```

```
                break;
            case SCALE_KEY:
                ascene->scale[nscale].xyz[0] = atof(arg0);
                ascene->scale[nscale].xyz[1] = atof(arg1);
                ascene->scale[nscale++].xyz[2] = atof(arg2);
                ascene->identities[ascene->nidentities].instr[instr_ind++] = SCALE_KEY;
                break;
            case LIGHT_KEY:
        {
            ascene->lights[ascene->nlights].light[0] = atof(arg0);
            ascene->lights[ascene->nlights].light[1] = atof(arg1);
            ascene->lights[ascene->nlights].light[2] = atof(arg2);

            int result_readAndParse = readAndParse(fp, keyword, arg0, arg1, arg2, arg3, arg4);
            key_id = match_Keyword(keyword, &npara);
            ascene->lights[ascene->nlights].ndirections = 0;
            while(key_id == VERTEX_KEY){
                ascene->lights[ascene->nlights].directions[ascene->lights[ascene->nlights].ndirections][0]
atof(arg0);
                ascene->lights[ascene->nlights].directions[ascene->lights[ascene->nlights].ndirections][1]
atof(arg1);
                ascene->lights[ascene->nlights].directions[ascene->lights[ascene->nlights].ndirections++][2]
atof(arg2);
                result_readAndParse = readAndParse(fp, keyword, arg0, arg1, arg2, arg3, arg4);
                key_id = match_Keyword(keyword, &npara);
            }
            fseek(fp,-result_readAndParse,SEEK_CUR);
            ascene->nlights++;
        }
            break;
            case AMBIENT_KEY:
                ascene->ambient[0] = atof(arg0);
                ascene->ambient[1] = atof(arg1);
                ascene->ambient[2] = atof(arg2);
                break;

            case SHADING_KEY:
                shading = atof(arg0);
                break;

            case DIFFUSE_KEY:
                diffuse[0] = atof(arg0);
                diffuse[1] = atof(arg1);
                diffuse[2] = atof(arg2);
                break;

            case SPECULAR_KEY:
                specular[0] = atof(arg0);
                specular[1] = atof(arg1);
                specular[2] = atof(arg2);
                specular[3] = atof(arg3);
                break;

            case MESH_KEY:
                {
                    FILE *off = fopen(arg0,"rb");
              ascene->mesh[nmesh].width = atof(arg1);
              ascene->mesh[nmesh].shading = shading;
```

```
            ascene->mesh[nmesh].diffuse[0] = diffuse[0];
            ascene->mesh[nmesh].diffuse[1] = diffuse[1];
            ascene->mesh[nmesh].diffuse[2] = diffuse[2];
            ascene->mesh[nmesh].specular[0] = specular[0];
            ascene->mesh[nmesh].specular[1] = specular[1];
            ascene->mesh[nmesh].specular[2] = specular[2];
            ascene->mesh[nmesh].specular[3] = specular[3];
            readAndParse(off, keyword, arg0, arg1, arg2, arg3, arg4);
            readAndParse(off, keyword, arg0, arg1, arg2, arg3, arg4);
            ascene->mesh[nmesh].nvertices = atoi(keyword);
            ascene->mesh[nmesh].vertices = (COLOR_VERTEX *)malloc(sizeof(COLOR_VERTEX) *
ascene->mesh[nmesh].nvertices);

            ascene->mesh[nmesh].npolygons = atoi(arg0);
            ascene->mesh[nmesh].polygons = (POLYGON *)malloc(sizeof(POLYGON) *
ascene->mesh[nmesh].npolygons);

            int mm;
            for(mm = 0; mm < ascene->mesh[nmesh].nvertices; mm++){
                readAndParse(off, keyword, arg0, arg1, arg2, arg3, arg4);
                ascene->mesh[nmesh].vertices[mm].xyzw[0] = atof(keyword);
                ascene->mesh[nmesh].vertices[mm].xyzw[1] = atof(arg0);
                ascene->mesh[nmesh].vertices[mm].xyzw[2] = atof(arg1);
                ascene->mesh[nmesh].vertices[mm].xyzw[3] = 1;
                if(arg4[0] == '\0'){
                    ascene->mesh[nmesh].vertices[mm].rgba[0] = fcolor.rgba[0];
                    ascene->mesh[nmesh].vertices[mm].rgba[1] = fcolor.rgba[1];
                    ascene->mesh[nmesh].vertices[mm].rgba[2] = fcolor.rgba[2];

                }
                else{
                    ascene->mesh[nmesh].vertices[mm].rgba[0] = atof(arg2);
                    ascene->mesh[nmesh].vertices[mm].rgba[1] = atof(arg3);
                    ascene->mesh[nmesh].vertices[mm].rgba[2] = atof(arg4);
                    }
                }
                for(mm = 0; mm < ascene->mesh[nmesh].npolygons; mm++){
                    readAndParse(off, keyword, arg0, arg1, arg2, arg3, arg4);
                    int index[5] = {atoi(arg0),atoi(arg1),atoi(arg2),atoi(arg3),atoi(arg4)};
                    ascene->mesh[nmesh].polygons[mm].nvertices = atoi(keyword);
                    int nn;
                    for(nn = 0; nn < ascene->mesh[nmesh].polygons[mm].nvertices;nn++){
                        ascene->mesh[nmesh].polygons[mm].num[nn] = index[nn];
                            }
                }

            nmesh++;
            ascene->identities[ascene->nidentities].instr[instr_ind++] = MESH_KEY;
            }
            break;
        case COLOR_KEY:
            fcolor.rgba[0] = atoi(arg0)/255.0;    fcolor.rgba[1] = atoi(arg1)/255.0;
            fcolor.rgba[2] = atoi(arg2)/255.0;
            break;
        case IDENTITY_KEY:
            case SAVE_KEY:
            case LINE_KEY:
                brk = 1;
```

```
                    fseek(fp,-readAndParseResult,SEEK_CUR);
                    break;
                default:
                    printf("%s:%d Line (%s %s %s %s %s %s) ignored.\n",
                        __FILE__, __LINE__, keyword, arg0, arg1, arg2,
                        arg3, arg4);
                        }
            if(brk == 1)
            break;
            readAndParseResult = readAndParse(fp, keyword, arg0, arg1, arg2, arg3, arg4);
                    }
        ascene->identities[ascene->nidentities].inStr_num = instr_ind;
        ascene->nidentities++;
        instr_ind = 0;
        brk = 0;
        break;

    case SAVE_KEY:
        strcpy(saved_fname, arg0);
        break;
    default:
        printf("%s:%d Keyword (%s) and the line (%s %s %s %s %s) ignored.\n",
                __FILE__, __LINE__, keyword, arg0, arg1, arg2, arg3, arg4);

    }
  }
  fclose(fp);
  return 0;
}
```

SSD.h file(in red):

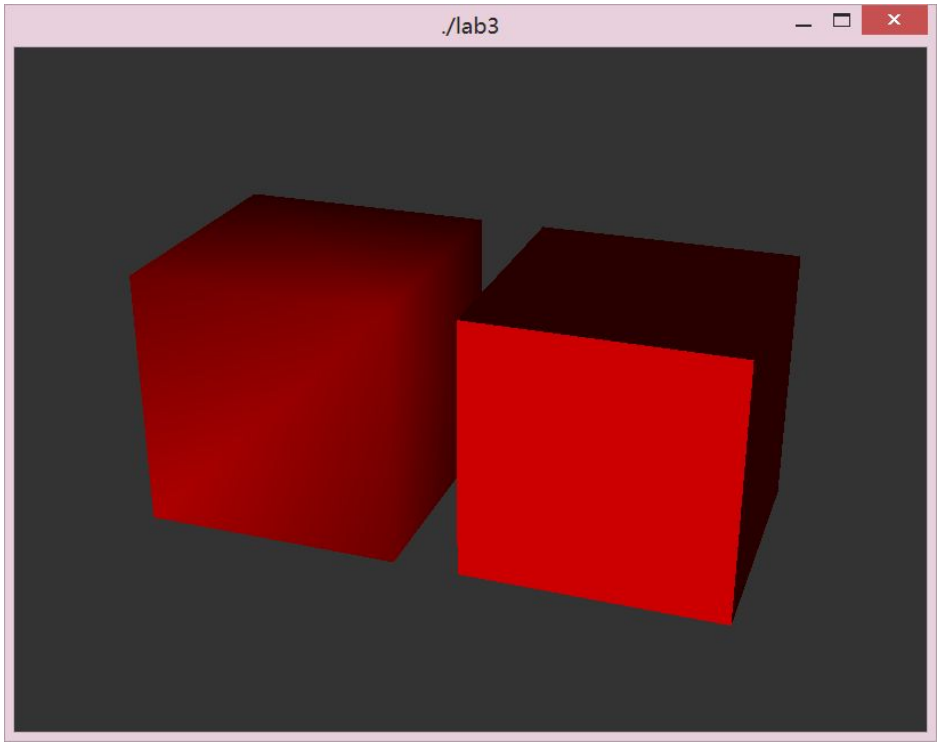
```
#if !defined(SSD_UTIL_H_H)
#define SSD_UTIL_H_H
#include <stdlib.h>
#include <string.h>

#define SCREEN_KEY      0
#define COLOR_KEY       1
#define LINE_KEY        2
#define VERTEX_KEY      3
#define POLYLINE_KEY 4
#define CIRCLE_KEY      5
#define ARC_KEY         6
#define SAVE_KEY        7
#define TRIANGLE_KEY 8
#define EYE_KEY         9
#define GAZE_KEY       10
#define UPVECTOR_KEY           11
#define ORTHO_KEY      12
#define PERSP_KEY       13
#define FLOOR_KEY      14
#define AXIS_KEY       15
#define IDENTITY_KEY      16
#define TRANSLATE_KEY    17
#define ROTATE_KEY         18
#define SCALE_KEY          19
```

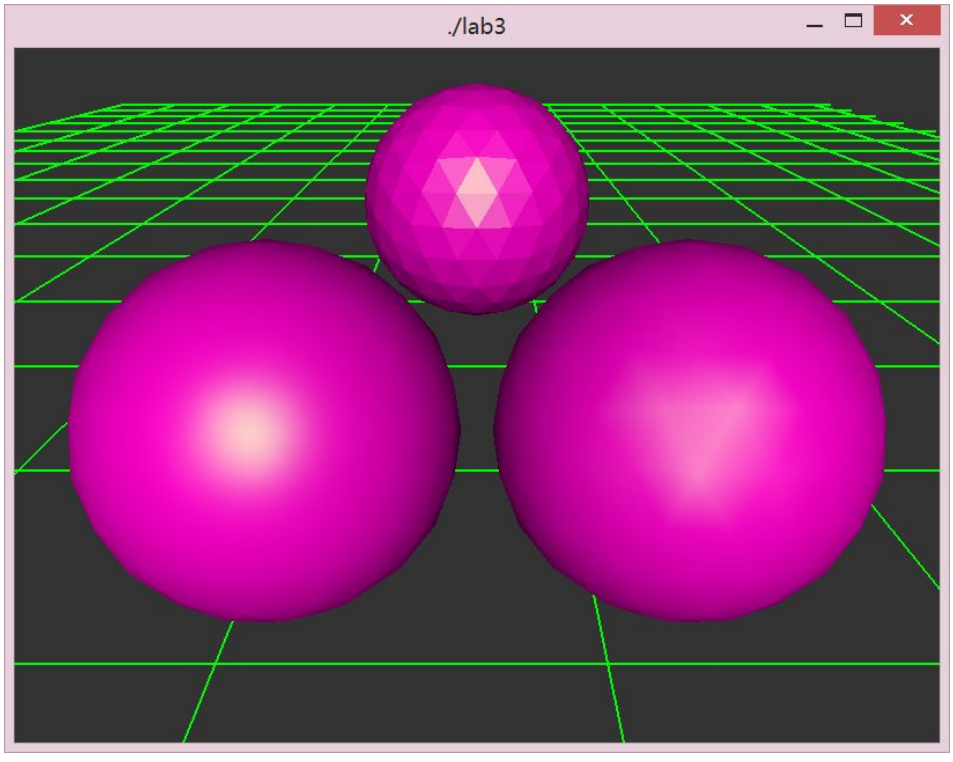
```
#define MESH_KEY            20
#define LIGHT_KEY           21
#define AMBIENT_KEY         22
#define SHADING_KEY         23
#define DIFFUSE_KEY         24
#define SPECULAR_KEY        25

struct ssd_keyword {
   /* Keyword table entry to be used for reading SSD */
   int key_id;
   char name[32];
   int npara;
};

typedef struct {
   double xyzw[4];
} VERTEX;

typedef struct {
   float rgba[4];
} RGB_COLOR;

typedef struct {
   double xyzw[4];
   float    rgba[4];
} COLOR_VERTEX;

typedef struct {
   double width;
   COLOR_VERTEX vertices[2];
} LINE;

typedef struct {
   double width;
   int       nvertices;
   COLOR_VERTEX *vertices;
} POLYLINE;

typedef struct {
   COLOR_VERTEX vertices[3];
} TRIANGLE;

typedef struct {
   int nvertices;
   int num[50];
} POLYGON;

typedef struct {
   double xyz[3];
} Vector;

typedef struct {
   double near;
   double far;
   double angle;
} PERSP;

typedef struct {
```

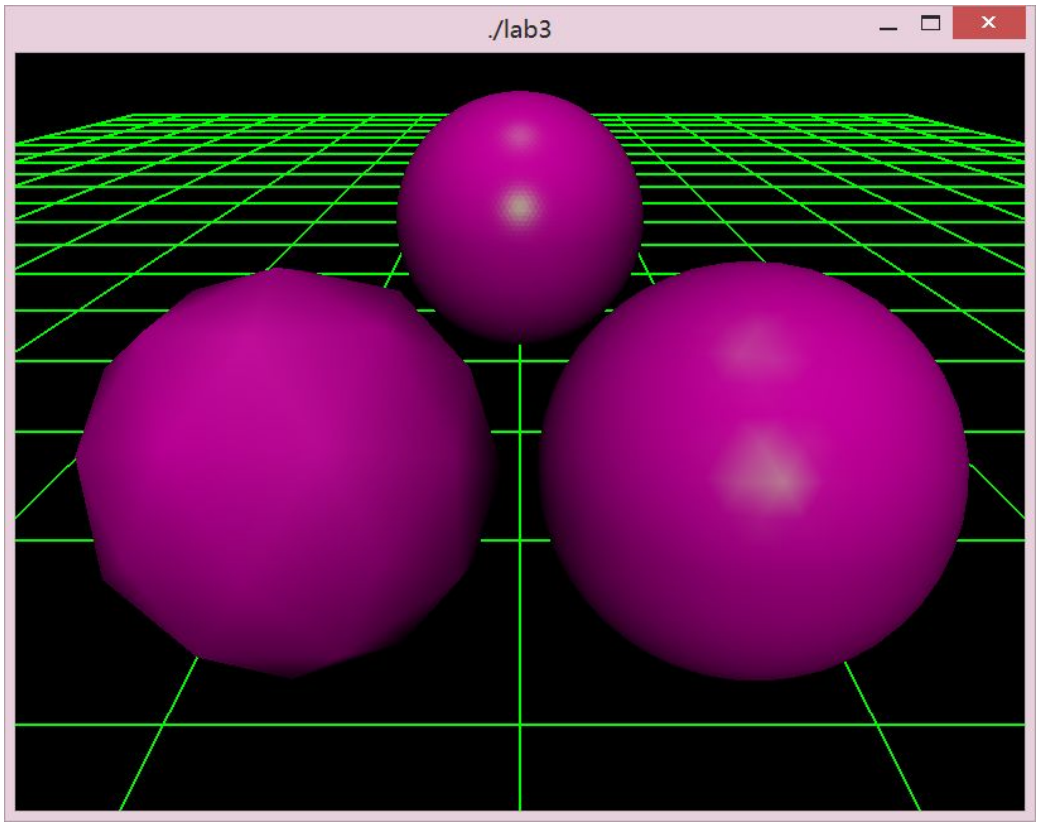
```
    double right;
    double top;
    double near;
    double far;
} ORTHO;

typedef struct {
    double xmin;
    double xmax;
    double ymin;
    double ymax;
    double size;
    RGB_COLOR color;
} FLOOR;

typedef struct {
    double width;
    double length;
} AXIS;

typedef struct {
    int inStr_num;
    int instr[50];
} IDENTITY;

typedef struct {
    double xyz[3];
} TRANSLATE;

typedef struct {
    double angle;
    double xyz[3];
} ROTATE;

typedef struct {
    double xyz[3];
} SCALE;

typedef struct {
    double width;
    int shading;
    double diffuse[3];
    double specular[4];
    int nvertices;
    COLOR_VERTEX *vertices;
    int npolygons;
    POLYGON *polygons;

} MESH;

typedef struct {
        double light[3];
        int ndirections;
        double directions[10][3];
} LIGHT;


typedef struct {
```

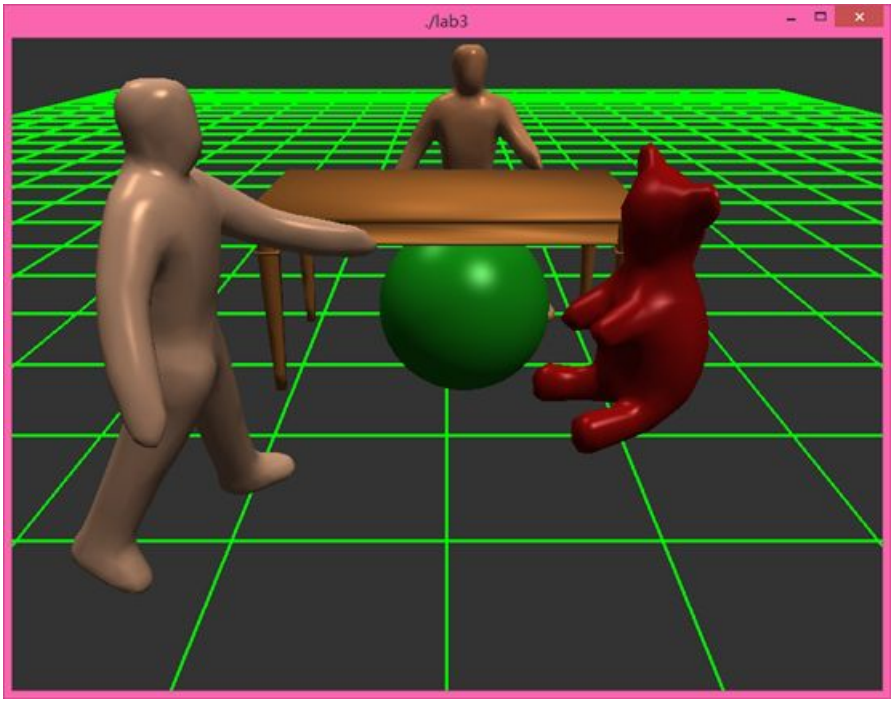
```
    int screen_w, screen_h;
    RGB_COLOR bcolor; /* The background color for the window */
    int    nlines; /* Number of lines */
    LINE *lines;
    int    npolylines; /* Number of the polylines */
    POLYLINE *polylines;
    int    ntriangles;
    TRIANGLE *triangles;

    int pjType;
    ORTHO ortho;
    PERSP persp;
    FLOOR floor;
    int isAxis;
    AXIS axis;
    int nidentities;
    IDENTITY *identities;
    TRANSLATE *translate;
    ROTATE *rotate;
    SCALE *scale;
    MESH *mesh;
    int nlights;
    LIGHT *lights;
    double ambient[3];

} SCENE;

typedef struct {
    //VERTEX position;
    VERTEX eye;
    VERTEX gaze;
    VERTEX upVector;

} CAMERA;

#if defined(SSD_UTIL_SOURCE_CODE)
#define EXTERN_FLAG
#else
#define EXTERN_FLAG extern
extern struct ssd_keyword keyword_table[];
#endif

EXTERN_FLAG
int match_Keyword(char *keyword, int *npara);

EXTERN_FLAG
int readAndParse(FILE *inFilePtr, char *keyword, char *arg0,
                 char *arg1, char *arg2, char *arg3, char *arg4);
EXTERN_FLAG
int Read_SSD_Scene(char *fname, SCENE *ascene, CAMERA *acamera, char *saved_fname);
#undef    EXTERN_FLAG

#endif
```

3. Result(screen shots)


./lab3


./lab3





4.Conclude

From this project, I learned more about illumination and shading. From the result I got, it could be found that, flat shading does not give a smooth surface perception which called Mach Band. And smooth shading does not handle specular highlights well, especially when highlights are small. The phong shading is the best among them.

In this project, there are many parameters need to calculate. So it is easy to get confused and debug. At the beginning, the highlight on the left ball in the second picture has a little deviation and I spent a lot time looking for the mistake. But finally, I found the bug is that I define a variable twice. Thus the bug got correct.