实验 三　　基于着色器的摄像漫游与环视

一、 实验内容

- ✓ 绘制一个由细分四面体得到的三维镂垫（或三维迷宫）及其适当的场景
- ✓ 实现摄像漫游：通过鼠标来上下左右地移动摄像机，也可通过键盘 W、S、A 和 D 键（或向上、向下、向左和向右方向键）来前后左右地移动摄像机，要求避免用户穿过三维镂垫（或三维镂垫）表面，设置漫游加速键；
- ✓ 实现摄像机环视：通过鼠标或键盘旋转摄像机环顾四周，包括左右巡视、向上仰视和向下俯视。

二、 实验环境

Window8 操作系统、VS2013、OpenGL 库、Intel HD Graphics 4000 图形卡

注： 运行的话可能需要用 VS2013，并且配置一下 GLAUX 库

三、 实验方案与程序设计

✧ 目标任务

迷宫场景和照相机漫游的实现

✧ 所需知识

迷宫的绘制/摄像机的实现/键盘按键/图元的使用/坐标系变化

✧ 具体说明

1.GLSL3.x 以上版本废弃了 QUADS 图元，可采用 TRIANGL_STRIP 绘制面。
2.GLSL3.x 以上版本废弃了显示列表，采用 VAO/VBO 传递给 GPU 进行运算。
3.采用 Glulookat 函数可实现照相机，其实质是对模型视图矩阵进行变化。在本次实验中，实现了 Glulookat 函数中的模型变化和视图变化，并将变化后得到的 viewMatrix、projMatrix 传递给 GPU 进行顶点坐标系变化。

✧ 设计方法

从固定流水线做起，基于 CPU 实现本次实验，在此基础上实现了基于 GPU 的三维迷宫，逐步求精。采用面向对象的设计思路，定义 Camera 类、Maze 类、Texture 类，通过 Main 函数进行迷宫显示、碰撞检测、摄像机控制和各种交互操作。具体如下：

摄像机类：

```
class CCamera {
public:
    //摄像机类的构造函数
    CCamera();
    vec3 getPos();
    vec3 getPrePos();
```

```
    mat4 getProjmat();
    mat4 getViewjmat();
    //移动摄像机
    void MoveCamera(float speed);
    //旋转摄像机
    void RotateCamera(float angle);
    //上扬摄像机
    void UpDownCamera(float viewY);
    //设置模型视图矩阵
    void setCamera();
    //设置投影矩阵
    void BuildProjMatrix(float fov, float ratio, float nearP, float farP);
private:
    //投影矩阵
    float projMatrix[16];
    //模型视图矩阵
    float viewMatrix[16];
    //照相机旋转角度
    float angle;
    //摄像机的位置
    vec3 m_vPosition;
    //摄像机先前的位置
    vec3 m_vPreviousPos;
    //摄像机的视野
    vec3 m_vView;
    //摄像机的向上的位置
    vec3 m_vUpVector;
    //自定义照相机
    void crossProduct(float *a, float *b, float *res);
    void normalize(float *a);
    void setIdentityMatrix(float *mat, int size);
    void multMatrix(float *a, float *b);
    void setTranslationMatrix(float *mat, float x, float y, float z);
};
```

迷宫类：

```
class CMaze{
public:
    //构造函数
    CMaze();
    //迷宫初始化
    void MazeInit();
    //墙壁检测
    bool Wall(int x, int y);
    //三维迷宫显示
```

```
    void Maze3D(mat4 proj, mat4 view);
    //二维迷宫显示
    void Maze2D(mat4 proj, mat4 view);
private:
    //遍历生成迷宫
    bool OnOpen(int x, int y);
    void CloseIt(int x, int y);
    bool Neighbor(int x, int y, int w, int *nx, int *ny);
    bool Diagnal(int x, int y, int w, int *nx, int *ny);
    void Dw(int x, int y, int p);
    //绘制墙壁
    void DrawWalls(void);
    //绘制天空和地板
    void DrawSkyGround(void);
    //绘制当前位置
    void DrawBall(void);
};
```

纹理类：

```
class GLTexture{
public:
    unsigned int texture[1];
    void Use();
    void BuildColorTexture(unsigned char r, unsigned char g, unsigned char b);
    void LoadBMP(char *name);
    GLTexture();
};
```

✧ 基本步骤

定义二维迷宫数组，链接成功后，递归地生成三维迷宫，其中 Z 值为 0 或 1，并且将顶点数据保存在顶点数组对象中。之后设置显示回调函数，变化摄像机进行模型变化和视图变化并且传递给 GPU。当收到按键消息后，在显示回调函数中修改摄像机的方向和位置，从而实现漫游。

✧ 迷宫生成：

注： 用 GPU 实现时，发现 GL_QUADS 和显示列表在现代图形设计中已经不支持采用，后面改为 GL_TRIANGL_STRIP+VAO/VBO 顶点数组/缓冲区对象实现

基于 CPU 实现

迷宫初始化：

```
void CMaze::MazeInit(){
  walllist = glGenLists(2);
  mazelist = walllist + 1;
  balllist = walllist + 2;
  glNewList(walllist, GL_COMPILE);
```

```
  DrawWalls();
  glEndList();
  glNewList(mazelist, GL_COMPILE);
  DrawTop();
  glEndList();
  glNewList(balllist, GL_COMPILE);
  DrawBall();
  glEndList();
}
```

遍历迷宫数组，生成迷宫：

```
void CMaze::Dw(int x, int y, int p) {
  int w = p;
  CloseIt(x, y);
  do{
    int x2 = 0;
    int y2 = 0;
    if (Neighbor(x, y, w, &x2, &y2)){
        if (OnOpen(x2, y2)) {
            Dw(x2, y2, (w + 3) % 4);
        }
        else {
            if ((w + 1) % 4 == p)
            {
                return;
            }
        }
    }
    else {
        float fx;
        float fy;
        if (Diagnal(x, y, w, &x2, &y2) && OnOpen(x2, y2)) {
            Dw(x2, y2, (w + 2) % 4);
        }
        texcoordX = (texcoordX<0.5) ? 1.0f : 0.0f;
        fx = (float)x + ((w == 1 || w == 2) ? 1.0f : 0.0f);
        fy = (float)y + ((w == 0 || w == 1) ? 1.0f : 0.0f);
        glTexCoord2f(texcoordX, 0.0f);
        glVertex3f(fx, fy, 0.0f);
        glTexCoord2f(texcoordX, 1.0f);
        glVertex3f(fx, fy, 1.0f);
    }
    w++; w %= 4;
  } while (w != p);
  return;
```

```
}
//绘制墙壁
void CMaze::DrawWalls() {
  texture[0].LoadBMP("Data/cc.bmp");
  texture[0].Use();
  glBegin(GL_QUAD_STRIP);
  glColor3f(1.0, 1.0, 1.0);
  glVertex3f(0.0f, 0.0f, 0.0f);
  glVertex3f(0.0f, 0.0f, 1.0f);
  Dw(0, 0, 0);
  glEnd();
}
//绘制天空和地板
void CMaze::DrawSkyGround() {
  for (int y = 0; y<MAZE_HEIGHT; y++) {
    for (int x = 0; x<MAZE_WIDTH; x++) {
      if (!Wall(x, y)) {
        texture[0].Use();
        glBegin(GL_QUADS);
        //天空
        glTexCoord2f(1.0f, 1.0f);
        glVertex3f(x + 0.0f, y + 0.0f, 0.0f);
        glTexCoord2f(0.0f, 1.0f);
        glVertex3f(x + 1.0f, y + 0.0f, 0.0f);
        glTexCoord2f(0.0f, 0.0f);
        glVertex3f(x + 1.0f, y + 1.0f, 0.0f);
        glTexCoord2f(1.0f, 0.0f);
        glVertex3f(x + 0.0f, y + 1.0f, 0.0f);
        glEnd();
        //地板
        glTexCoord2f(1.0f, 1.0f);
        glVertex3f(x + 0.0f, y + 0.0f, 1.0f);
        glTexCoord2f(0.0f, 1.0f);
        glVertex3f(x + 1.0f, y + 0.0f, 1.0f);
        glTexCoord2f(0.0f, 0.0f);
        glVertex3f(x + 1.0f, y + 1.0f, 1.0f);
        glTexCoord2f(1.0f, 0.0f);
        glVertex3f(x + 0.0f, y + 1.0f, 1.0f);
        glEnd();
      }
    }
  }
```

基于 GPU 实现

顶点着色器：

```
in vec4 vPosition;
in vec4 vColor;
in vec2 vTexCoord;
out vec4 color;
out vec2 texCoord;
uniform mat4 viewMatrix;
uniform mat4 projMatrix;
void main(){
	texCoord = vTexCoord;
	color = vColor;
	gl_Position = projMatrix *viewMatrix * vPosition;
}
```

片元着色器：

```
in vec4 color;
in vec2 texCoord;
out vec4 fColor;
uniform sampler2D texture;
void main(){
	fColor = color* texture2D(texture, texCoord );
}
```

遍历迷宫数组，生成迷宫

```
void CMaze::Dw(int x, int y, int p) {
  int w = p;
  CloseIt(x, y);
  do{
    int x2 = 0;
    int y2 = 0;
    if (Neighbor(x, y, w, &x2, &y2)){
        if (OnOpen(x2, y2)) {
            Dw(x2, y2, (w + 3) % 4);
        }
        else {
            if ((w + 1) % 4 == p)
            {
                return;
            }
        }
    }
    else {
        float fx;
        float fy;
        if (Diagnal(x, y, w, &x2, &y2) && OnOpen(x2, y2)) {
```

```
            Dw(x2, y2, (w + 2) % 4);
        }
        texcoordX = (texcoordX<0.5) ? 1.0f : 0.0f;
        fx = (float)x + ((w == 1 || w == 2) ? 1.0f : 0.0f);
        fy = (float)y + ((w == 0 || w == 1) ? 1.0f : 0.0f);
        wall_texcoords[NumVerticesW] = vec2(texcoordX, 0.0f);
        wall_points[NumVerticesW++] = vec4(fx, fy, 0.0f, 1.0f);
        wall_texcoords[NumVerticesW] = vec2(texcoordX, 1.0f);
        wall_points[NumVerticesW++] = vec4(fx, fy, 1.0f, 1.0f);
    }
    w++; w %= 4;
  } while (w != p);
  return;
}
//绘制墙壁
void CMaze::DrawWalls() {
  for (int i = 0; i < 500; i++){
    wall_colors[i] = color4(1.0, 1.0, 1.0, 1.0);
  }
  wall_texcoords[NumVerticesW] = vec2(0.0f, 0.0f);
  wall_points[NumVerticesW++] = vec4(0.0f, 0.0f, 0.0f, 1.0f);
  wall_texcoords[NumVerticesW] = vec2(0.0f, 1.0f);
  wall_points[NumVerticesW++] = vec4(0.0f, 0.0f, 1.0f, 1.0f);
  Dw(0, 0, 0);
}
//绘制天空和地板
void CMaze::DrawSkyGround() {
  for (int i = 0; i < 500; i++){
    ground_colors[i] = color4(1.0, 1.0, 1.0, 1.0);
    sky_colors[i] = color4(1.0, 1.0, 1.0, 1.0);
   }
  for (int y = 0; y < MAZE_HEIGHT; y++) {
    for (int x = 0; x < MAZE_WIDTH; x++) {
        if (!Wall(x, y)) {
            //地板
            ground_texcoords[NumVerticesG] = vec2(0.0f, 0.0f);
            ground_points[NumVerticesG++] = vec4(x + 1.0f, y + 0.0f, 0.0f, 1.0f);
            ground_texcoords[NumVerticesG] = vec2(0.0f, 1.0f);
            ground_points[NumVerticesG++] = vec4(x + 1.0f, y + 1.0f, 0.0f, 1.0f);
            ground_texcoords[NumVerticesG] = vec2(1.0f, 0.0f);
            ground_points[NumVerticesG++] = vec4(x + 0.0f, y + 0.0f, 0.0f, 1.0f);
            ground_texcoords[NumVerticesG] = vec2(1.0f, 1.0f);
            ground_points[NumVerticesG++] = vec4(x + 0.0f, y + 1.0f, 0.0f, 1.0f);
            //天空
```

```
            sky_texcoords[NumVerticesS] = vec2(0.0f, 0.0f);
            sky_points[NumVerticesS++] = vec4(x + 1.0f, y + 0.0f, 1.0f, 1.0f);
            sky_texcoords[NumVerticesS] = vec2(0.0f, 1.0f);
            sky_points[NumVerticesS++] = vec4(x + 1.0f, y + 1.0f, 1.0f, 1.0f);
            sky_texcoords[NumVerticesS] = vec2(1.0f, 0.0f);
            sky_points[NumVerticesS++] = vec4(x + 0.0f, y + 0.0f, 1.0f, 1.0f);
            sky_texcoords[NumVerticesS] = vec2(1.0f, 1.0f);
            sky_points[NumVerticesS++] = vec4(x + 0.0f, y + 1.0f, 1.0f, 1.0f);
        }
      }
    }
}

//绘制场景
void CMaze::Maze3D(mat4 projMatrix, mat4 viewMatrix){
 //设置模型矩阵和视图矩阵
  glUniformMatrix4fv(projMatrixLoc, 1, false, projMatrix);
  glUniformMatrix4fv(viewMatrixLoc, 1, false, viewMatrix);
  texture[1].Use();
   //墙壁数据
  glBindVertexArray(vao[0]);
  glDrawArrays(GL_TRIANGLE_STRIP, 0, NumVerticesW);
   //天空数据
  glBindVertexArray(vao[1]);
  glDrawArrays(GL_TRIANGLE_STRIP, 0, NumVerticesG);
   //地板数据
  glBindVertexArray(vao[2]);
  glDrawArrays(GL_TRIANGLE_STRIP, 0, NumVerticesS);
}
```

✧ **其他说明:**

相机漫游

通过定义相机当前视角、视点位置、视线方向，接受用户当前操作获取角度变化和速度变化，进行 sin、cos 三角变化从而实现相机参数的更新。在 CPU 版本，将视点位置和视线方向作为 glulookat 的前六个参数；在 GPU 版本，将视点位置和视线方向作为自定义 setCamera 的参数，进行模型视图变化，获取矩阵传递给 GPU 后进行顶点变化。

```
//构造函数
CCamera::CCamera()
{
  this->angle = 90.0f;
   //初始化摄像机位置、方向、向上方向
  vec3 vZero = vec3(13.5f, 0.3f , -16.5f);
  vec3 vView = vec3(13.5f, 0.3f, -15.5f);
  vec3 vUp = vec3(0.0, 1.0, 0);
```

```
  m_vPreviousPos = m_vPosition = vZero;
  m_vView = vView;
  m_vUpVector = vUp;
}

//根据一定的速度前后移动摄像机
void CCamera::MoveCamera(float speed)
{
   float rad = angle*(PI_ / 180.0);
   //移动摄像机的位置坐标 X
   m_vPosition.x += cos(rad) * speed;
   //移动摄像机的位置坐标 Z
   m_vPosition.z += sin(rad) * speed;
}

// 根据一定的角度左右旋转摄像机
void CCamera::RotateCamera(float angle)
{
  this->angle += angle;
  float rad = this->angle*(PI_ / 180.0);
  m_vView.x = m_vPosition.x + cos(rad);
  m_vView.z = m_vPosition.z + sin(rad);
}
// 根据一定的角度上下旋转摄像机
void CCamera::UpDownCamera(float viewY)
{
  m_vView.y += viewY;
}
//   更新摄像机方向和位置
void CCamera::Update(int positionX, int positionY , bool flag)
{
  if (flag == true){
     m_vPosition.x = positionX;
     m_vPosition.z = positionY;
  }
  RotateCamera(0);
}
```

矩阵变化

通过定义模型矩阵和视图矩阵，对其进行模型变化和视图变化，以实现 glulookat 的功能，从而避免了一堆入栈和出栈的操作，也不需要在顶点着色器中使用内置变量 GLModelViewProjection。

```
//投影矩阵变化
void CCamera::BuildProjMatrix(float fov, float ratio, float nearP, float farP)
{
```

```
  float f = 1.0f / tan(fov * (M_PI / 360.0));
  setIdentityMatrix(projMatrix, 4);
  projMatrix[0] = f / ratio;
  projMatrix[1 * 4 + 1] = f;
  projMatrix[2 * 4 + 2] = (farP + nearP) / (nearP - farP);
  projMatrix[3 * 4 + 2] = (2.0f * farP * nearP) / (nearP - farP);
  projMatrix[2 * 4 + 3] = -1.0f;
  projMatrix[3 * 4 + 3] = 0.0f;
}

//视图矩阵变化
void CCamera::setCamera()
{
  GLfloat    posX = m_vPosition.x;
  GLfloat    posY = m_vPosition.y;
  GLfloat    posZ = m_vPosition.z;
  GLfloat    lookAtX = m_vView.x;
  GLfloat    lookAtY = m_vView.y;
  GLfloat    lookAtZ = m_vView.z;

  float dir[3], right[3], up[3];
  up[0] = 0.0f;      up[1] = 1.0f;      up[2] = 0.0f;
  dir[0] = (lookAtX - posX);
  dir[1] = (lookAtY - posY);
  dir[2] = (lookAtZ - posZ);
  normalize(dir);

  crossProduct(dir, up, right);
  normalize(right);
  crossProduct(right, dir, up);
  normalize(up);

  float aux[16];
  viewMatrix[0] = right[0];
  viewMatrix[4] = right[1];
  viewMatrix[8] = right[2];
  viewMatrix[12] = 0.0f;
  viewMatrix[1] = up[0];
  viewMatrix[5] = up[1];
  viewMatrix[9] = up[2];
  viewMatrix[13] = 0.0f;
  viewMatrix[2] = -dir[0];
  viewMatrix[6] = -dir[1];
  viewMatrix[10] = -dir[2];
  viewMatrix[14] = 0.0f;
```

```
  viewMatrix[3] = 0.0f;
  viewMatrix[7] = 0.0f;
  viewMatrix[11] = 0.0f;
  viewMatrix[15] = 1.0f;

  setTranslationMatrix(aux, -posX, -posY, -posZ);
  multMatrix(viewMatrix, aux);
}
// ----------------------------------------------------
//矩阵操作类
//
//矩阵叉乘
void CCamera::crossProduct(float *a, float *b, float *res)
{
  res[0] = a[1] * b[2] - b[1] * a[2];
  res[1] = a[2] * b[0] - b[2] * a[0];
  res[2] = a[0] * b[1] - b[0] * a[1];
}

//归一化
void CCamera::normalize(float *a)
{
  float mag = sqrt(a[0] * a[0] + a[1] * a[1] + a[2] * a[2]);
  a[0] /= mag;
  a[1] /= mag;
  a[2] /= mag;
}

//重置
void CCamera::setIdentityMatrix(float *mat, int size)
{
  // fill matrix with 0s
  for (int i = 0; i < size * size; ++i)
    mat[i] = 0.0f;

  // fill diagonal with 1s
  for (int i = 0; i < size; ++i)
    mat[i + i * size] = 1.0f;
}

//矩阵相乘
void CCamera::multMatrix(float *a, float *b)
{
  float res[16];
  for (int i = 0; i < 4; ++i) {
    for (int j = 0; j < 4; ++j) {
```

```
            res[j * 4 + i] = 0.0f;
            for (int k = 0; k < 4; ++k) {
                res[j * 4 + i] += a[k * 4 + i] * b[j * 4 + k];
            }
        }
    }
    memcpy(a, res, 16 * sizeof(float));
}

// 平移
void CCamera::setTranslationMatrix(float *mat, float x, float y, float z)
{
    setIdentityMatrix(mat, 4);
    mat[12] = x;
    mat[13] = y;
    mat[14] = z;
}
```

碰撞检测

对移动相机前后所处的视点位置进行比较，并且取整判断是否碰到墙壁，是的话前进到墙壁前，不是的话则正常前进。

```
//墙壁检测
void Forward(float bf = 0.2f)
{
    vec3 pos = camera.getPos();
    vec3 pre = camera.getPrePos();
    int x = ((int)pre.x);
    int y = abs(((int)pre.z));
    GLfloat px = pos.x;
    GLfloat py = abs(pos.z);
    bool flag = false;
    if ((px> x + 1.0f - bf) && maze.Wall(x + 1, y)) {
        px = (float)(x)+1.0f - bf;
        flag = true;
    }
    if ((py> y + 1.0f - bf) && maze.Wall(x, y + 1)) {
        py = (float)(y)+1.0f - bf;
        flag = true;
    }
    if ((px< x + bf )&& maze.Wall(x - 1, y)) {
        px = (float)(x)+bf;
        flag = true;
    }
    if ((py< y + bf )&& maze.Wall(x, y - 1)) {
```

```
    py = (float)(y)+bf;
    flag = true;
  }
  camera.Update(px, py, flag);
}
```

交互操作

```
//鼠标点击回调
void onMouseClick(GLint button, GLint state, GLint x, GLint y)
{
  //第一次鼠标按下时,记录鼠标在窗口中的初始坐标
  if (state == GLUT_DOWN)
    mouth_X = x, mouth_Y = y;
}

//鼠标移动回调
void onMouseMove(GLint x, GLint y){
  //判断八个方向
  if (x < mouth_X && fabs(y - mouth_Y) <= 5){
    camera.RotateCamera(-kAngle / 5);
  }
  else if (x > mouth_X && fabs(y - mouth_Y) <= 5) {
    camera.RotateCamera(kAngle / 5);
  }
  else if (fabs(x - mouth_X) <= 5 && y > mouth_Y){
    camera.UpDownCamera(-upAngle);
  }
  else if (fabs(x - mouth_X) <= 5 && y < mouth_Y)    {
    camera.UpDownCamera(upAngle);
  }
  else if (x > mouth_X && y > mouth_Y) {
    camera.RotateCamera(kAngle / 5), camera.UpDownCamera(-upAngle);
  }
  else if (x < mouth_X && y < mouth_Y) {
    camera.RotateCamera(-kAngle / 5), camera.UpDownCamera(upAngle);
  }
  else if (x > mouth_X && y < mouth_Y) {
    camera.RotateCamera(kAngle / 5), camera.UpDownCamera(upAngle);
  }
  else if (x < mouth_X && y > mouth_Y) {
    camera.RotateCamera(-kAngle / 5), camera.UpDownCamera(-upAngle);
  }
  mouth_X = x, mouth_Y = y;
  glutPostRedisplay();
}
```

```
//方向键回调
void SpecialKeys(int key, int x, int y)
{
  if (key == GLUT_KEY_UP) {
    camera.MoveCamera(kSpeed);
  }
  if (key == GLUT_KEY_DOWN) {
    camera.MoveCamera(-kSpeed);
  }
  if (key == GLUT_KEY_LEFT) {
    camera.RotateCamera(-kAngle);
  }
  if (key == GLUT_KEY_RIGHT) {
    camera.RotateCamera(kAngle);
  }
  Forward(0.2f);
  glutPostRedisplay();
}

//特殊键回调
void Keyboard(unsigned char key, int x, int y)
{
  switch (key)
  {
  case '1':
    Is3D = true;
    break;
  case '2':
    Is3D = false;
    break;
  case '3':
    kSpeed *= 2;       //加速
    break;
  case '4':
    kSpeed /= 2;       //减速
    break;
  default:break;
  }
  glutPostRedisplay();
}
```

四、 实验结果与分析（包括功能说明、操作说明）

- ✧ 按住键盘方向键可实现相机的前进、后退、左看和右看。
- ✧ 移动鼠标可从八个方向实现相机视线的调整，包括左看、右看、上看和下看。
- ✧ 点击键盘 1 键选择三维显示，2 键选择二维显示，3 键实现漫游加速，4 键实现漫游减速。

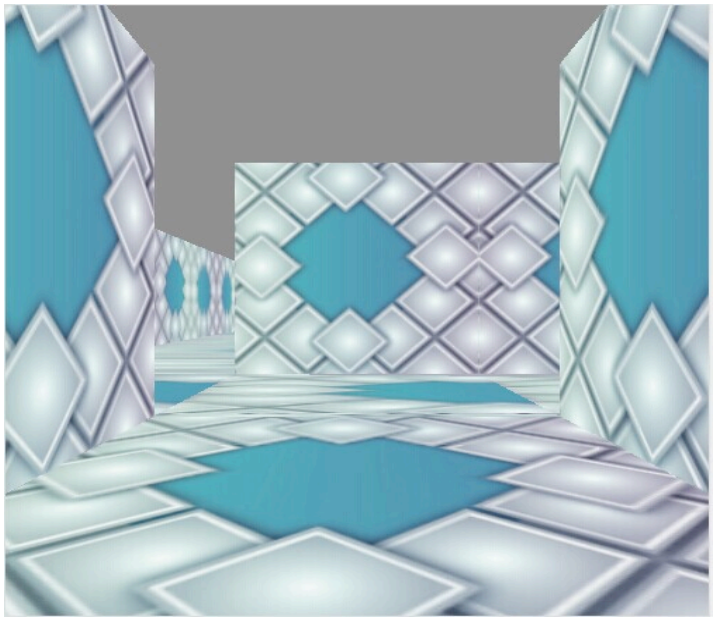

图 1-初始入口

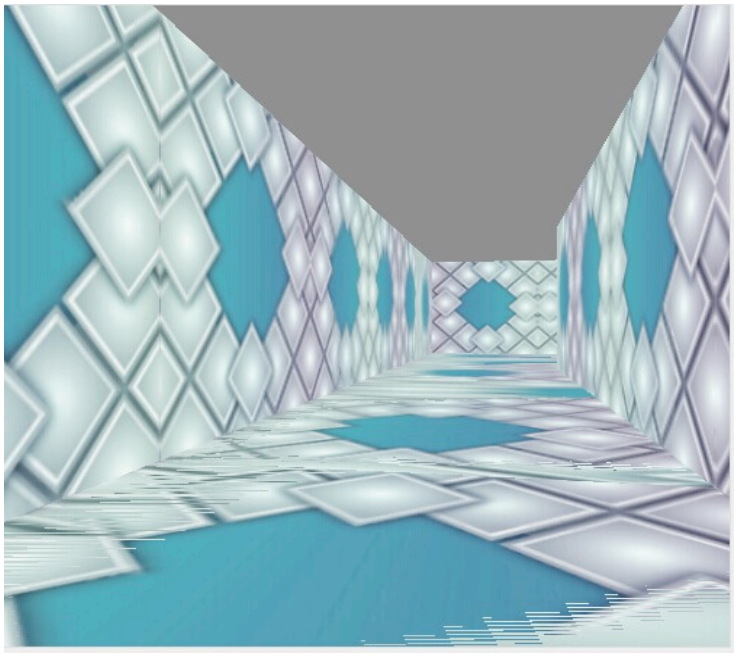

图 2-移动照相机（前进）

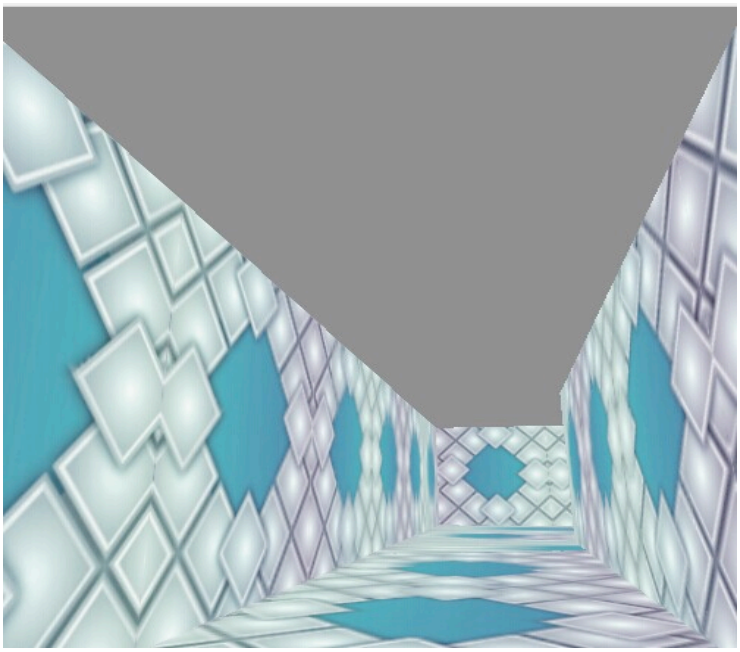

图 3-移动照相机（上扬）

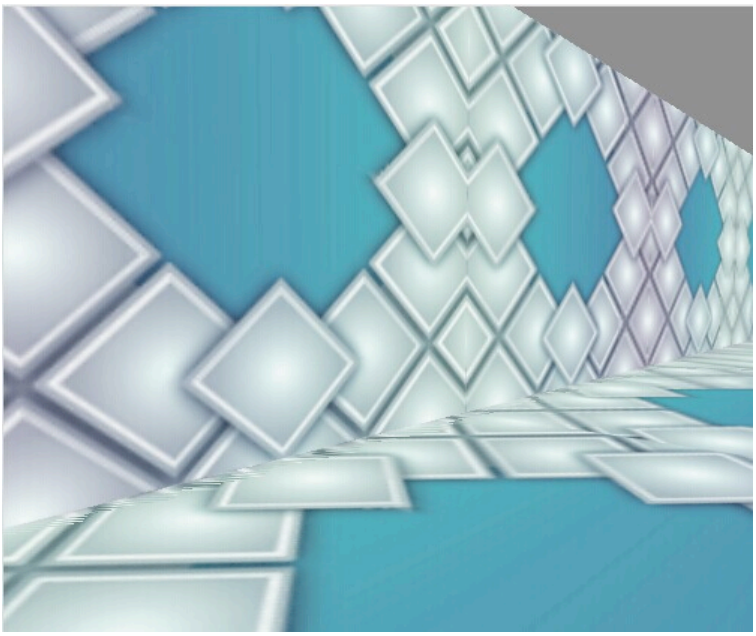

图 4-移动照相机（左右）

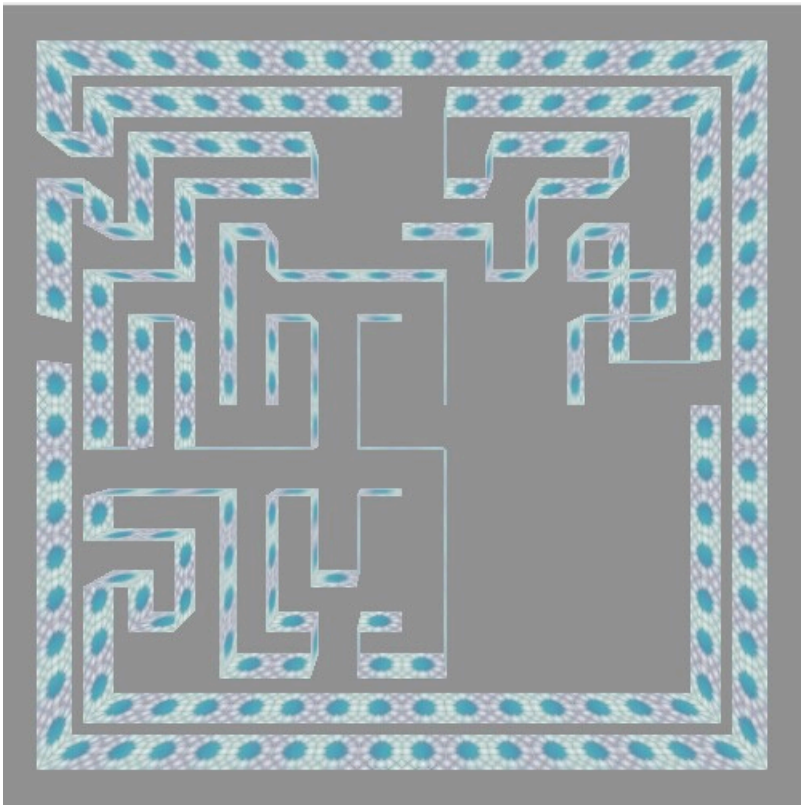

图 5-二维迷宫

五、 实验心得（自我评价）与建议

- ✧ 在这次实验中，碰到了比较多的问题，自己的设计思路是从 CPU 开始，认为完成 CPU 的迷宫后，可以很快地移植成 GPU 版本，结果遇到了各种问题，几乎都是百度不能解决，只能翻墙去谷歌搜些国外的资料。
- ✧ 在这次实验中，虽然固定流水线支持 QUADS 图元，但可编程流水线仅支持几种简单的图元，查阅了大量资料后发现可以改用 GL_TRIANGLE_STRIP 实现 QUADS 的功能，STRIP 的顺序也让我研究了很久。
- ✧ 在这次实验中，虽然固定流水线支持显示列表方式高效地绘制（为此还花费了不少时间研究显示列表中函数的执行顺序），但是可编程流水线因 GPU 的高效性废弃了显示列表，因此改用 VAO/VBO 存储要绘制的顶点数据。
- ✧ 在这次实验中，发现 glutInitContextVersion(3, 2)最好要去掉，因为它限制了当前版本必须为 3.2，最新已经有 4.X 了。有一些功能在 3.X 上是不能采用的。
- ✧ 在这次实验中，对 OpenGL 坐标系变化有了更深入的了解。从模型坐标到世界坐标再到照相机坐标的变化过程，也在自己实现 ModelViewProjection 时进行了深入研究。
- ✧ 除了以上心得外，最主要的还是迷宫递归生成和碰撞检测，这个部分花了相当多的时间。虽然因为时间限制，做的还不能完全达到自己要求，但是在这个过程中学到了很多，再之后我会用 A*算法进行迷宫自动寻径。