**《计算机图形学》编程实验报告**

2022-2023学年第二学期翡翠湖校区 翡翠科教楼 机房

专业: **计算机科学与技术** 姓名: 杜述超 学号: 2021214745 成绩:

|  |
| --- |
| **第四次编程实验题目:大作业** |
| **1.实验目的:** |
| 1. 实现网格模型加载与显示（用光强插值实现局部光照模型）； 2. 加载 OFF 格式的模型文件（不含顶点法向信息）；关于 OFF 文件格式： 3. 第一行为文件格式；第二行三个参数为顶点数、面片数和无效信息（必须读入）；接下来两个数据块，分别是顶点数据和三角面片数据。 4. 依据顶点周围一邻域的面片法线信息（面片法向信息可以用两条边叉乘获得）计算顶点法线； 5. 采用透视投影方法，将模型放在坐标原点处（0，0，0）处。 6. 交互操作：在模型四个方向上绘制四个不同颜色的色块（即 4 个小正方形），利用用户拾取其中某个色块颜色的信息，调用变换函数改变物体位置。 7. 熟悉以下函数使用，采用 GL\_RGB 读取颜色缓存中指定像素的颜色。void glReadPixels( GLint x, GLint y, GLsizei width,GLsizei height, GLenum format, GLenum type, GLvoid \*pixels) |
| **2.实验原理(**描述核心算法内容)**:** |
| 核心算法：  这里的核心算法在求顶点法向量的时候，求顶点法向是需要求面的法向的，这里会有两种思路：  第一种是直接先求出每一个面的法向再求顶点法向，求每个顶点法向时候需要再次遍历每个面以获取面的法向，但这个方法效率极低，求得结果我甚至需要等几分钟；  第二中是在第一次遍历面求面的法向时，求得了面的法向后立马将该面的法向加入到该面的所有顶点中，最后再遍历所有顶点对顶点的法向进行归一化处理，这样就不需要再遍历所有面了，极大提高了效率。  这里对第二种方法的算法进行详细阐述：  1. 对于给定的面（Face），函数Mesh::normal(Face& face)用于计算面的法线。下面是该算法的详细描述：   1. 获取面的三个顶点的索引：通过面对象中的index数组获取三个顶点的索引，分别为index1、index2和index3。 2. 获取顶点的指针：通过顶点索引从vertexs数组中获取对应的顶点指针，分别为ver1、ver2和ver3。 3. 计算两个具有方向的线（line1和line2）：根据顶点之间的坐标差计算两个线的方向。其中，line1由ver2减去ver1得到，line2由ver3减去ver1得到。 4. 计算面的法线：   首先，计算法线的三个分量，即法线在 x、y 和 z 方向上的分量。  法线的 x 分量为 line1[1] \* line2[2] - line1[2] \* line2[1]。  法线的 y 分量为 line1[2] \* line2[0] - line1[0] \* line2[2]。  法线的 z 分量为 line1[0] \* line2[1] - line1[1] \* line2[0]。  注：这里使用了向量的叉积运算来计算法线。   1. 更新顶点的法线： 2. 遍历面的三个顶点，分别是face.index[0]、face.index[1]和face.index[2]。 3. 对于每个顶点，获取对应的顶点指针ver。 4. 将面的法线分量添加到顶点的法线分量上，即 ver->normal[0] += normal\_x、ver->normal[1] += normal\_y、ver->normal[2] += normal\_z。 5. 归一化面的法线： 6. 计算法线的模长（即三个分量的平方和的平方根），用于后续的归一化操作。 7. 将法线的各个分量除以模长，得到单位长度的法线。 8. 更新面对象的法线属性，将归一化后的法线保存在face.normal中。   最后通过调用Mesh::normal(Face& face)函数，可以为每个面计算法线，并将法线信息保存在对应的面对象中。此算法基于面的顶点坐标差来计算法线，并利用叉积运算和归一化操作来获得最终的法线值。  2. 对于给定的顶点，函数Mesh::normal(int index)用于计算顶点的法线。下面是该算法的详细描述：   1. 获取顶点的指针：通过顶点索引index从vertexs数组中获取对应的顶点指针，表示为ver。 2. 归一化顶点的法线： 3. 计算顶点法线的模长（即三个分量的平方和的平方根），用于后续的归一化操作。计算模长时，使用顶点的normal属性中保存的法线分量。 4. 将顶点法线的各个分量除以模长，得到单位长度的法线。 5. 更新顶点对象的法线属性，将归一化后的法线保存在ver->normal中。   最后通过调用Mesh::normal(int index)函数，可以为每个顶点计算法线，并将法线信息保存在对应的顶点对象中。该算法基于顶点的法线分量进行归一化操作，以获得单位长度的法线向量。 |
| **3.实验环境：** |
| 开发环境：Visual Studio 2022  实验平台：OpenGL 1.0 |
| **4.实验过程、步骤 (含重要函数流程图及关键代码):** |
| **构建Mesh数据结构部分：**   1. 流程图：   求法向函数流程图：       1. 核心代码：   这段代码在实验原理部分已经详细描述过了，这里就不再重复描述。  void Mesh::normal(Face& face)//calculate the normal of the face  {  int index1 = face.index[0], index2 = face.index[1], index3 = face.index[2];  Vertex\* ver1 = &vertexs[index1],\*ver2 = &vertexs[index2], \*ver3 = &vertexs[index3];  //two lines with direction  vector<double> line1 = { ver2->x - ver1->x,ver2->y - ver1->y,ver2->z - ver1->z };  vector<double> line2 = { ver3->x - ver1->x,ver3->y - ver1->y,ver3->z - ver1->z };  //the normal of the face  double normal\_x = line1[1] \* line2[2] - line1[2] \* line2[1];  double normal\_y = line1[2] \* line2[0] - line1[0] \* line2[2];  double normal\_z = line1[0] \* line2[1] - line1[1] \* line2[0];    for (int i = 0; i < face.index.size(); i++)//add the normal of the face to the vertexs  {  Vertex\* ver = &vertexs[face.index[i]];  ver->normal[0] += normal\_x;  ver->normal[1] += normal\_y;  ver->normal[2] += normal\_z;  }  //normalization of the normal  face.normal = { normal\_x,normal\_y,normal\_z };  double um = sqrt(pow(normal\_x, 2) + pow(normal\_y, 2) + pow(normal\_z, 2));//the sum of the square of the normal  face.normal = { normal\_x / um,normal\_y / um,normal\_z / um };//the normal of the face  }  void Mesh::all\_normal\_of\_faces()//calculate the normal of all faces  {  for (int i = 0; i < faces.size(); i++)  normal(faces[i]);  }  void Mesh::normal(int index)//calculate the normal of the vertex  {  Vertex\* ver = &vertexs[index];//the vertex  //normalization of the normal  double quad\_sum = sqrt(pow(ver->normal[0], 2) + pow(ver->normal[1], 2) + pow(ver->normal[2], 2));//the sum of the square of the normal  ver->normal = { ver->normal[0] / quad\_sum,ver->normal[1] / quad\_sum,ver->normal[2] / quad\_sum };//the normal of the vertex  }  void Mesh::all\_normal\_of\_vertexs()//calculate the normal of all vertexs  {  for (int i = 0; i < vertexs.size(); i++)  normal(i);  }  **交互部分：**   1. 流程图：        1. 核心代码：   这段代码描述了键盘事件和鼠标事件的交互方式。下面对其进行详细描述：   1. 键盘事件（void KeyBoardEvent(unsigned char key, int x, int y)）：   根据按下的按键key进行不同的操作：   1. 如果按下的是 'w' 键，向上移动，即将 change.dy 增加 0.05。 2. 如果按下的是 's' 键，向下移动，即将 change.dy 减少 0.05。 3. 如果按下的是 'a' 键，向左移动，即将 change.dx 减少 0.05。 4. 如果按下的是 'd' 键，向右移动，即将 change.dx 增加 0.05。 5. 如果按下的是 'q' 键，向里移动，即将 change.dz 增加 0.05。 6. 如果按下的是 'e' 键，向外移动，即将 change.dz 减少 0.05。 7. 如果按下的是其他键，弹出一个错误对话框，显示提示信息"oh, fuck bro, you pushed the wrong button"。 8. 鼠标事件（void MouseEvent(int button, int state, int x, int y)）：   当鼠标左键被按下或释放时：   1. 如果按下左键，记录当前鼠标的坐标作为旧坐标，并设置 change.mouse\_state 为 "down"。 2. 如果释放左键，将 change.mouse\_state 设置为 "up"。 3. 如果鼠标在特定的区域内按下左键，进行相应的操作。例如，如果鼠标在区域 (724, 4) 到 (768, 42) 内按下左键，将 change.angle\_y 减去 3。 4. 当鼠标右键被按下时，将 change 的值恢复为 copy\_ 的值。 5. 当鼠标滚轮滚动时，根据滚轮的方向进行缩放操作。如果滚轮向前滚动（按钮值为3），将 change.scale 增加 0.05；如果滚轮向后滚动（按钮值为4），将 change.scale 减少 0.05。 6. 鼠标移动事件（void MouseMotionEvent(int x, int y)）：   当鼠标左键被按下并移动时，根据鼠标的移动距离更新 change.angle\_x 和 change.angle\_y 的值。通过计算当前鼠标坐标 (x, y) 与旧坐标的差值乘以一个固定的比例系数（0.3）得到更新的值。  这段代码实现了通过键盘和鼠标与程序进行交互的功能。按下不同的键和进行鼠标操作可以改变相应的参数值，从而实现物体的移动、旋转和缩放等效果。  void KeyBoardEvent(unsigned char key, int x, int y)//keyboard event  {  switch (key)  {  case 'w':  change.dy += 0.05;  break;  case 's':  change.dy += -0.05;  break;  case 'a':  change.dx += -0.05;  break;  case 'd':  change.dx += 0.05;  break;  case 'q':  change.dz += 0.05;  break;  case 'e':  change.dz -= 0.05;  break;  default:  MessageBox(NULL, L"oh,fuck bro,you pushed the wrong button", L"error", MB\_OK);  break;  }  }  void MouseEvent(int button, int state, int x, int y)//mouse event  {  float cmp = 0.1;  if (button == GLUT\_LEFT\_BUTTON)  {  if (state == GLUT\_DOWN)  {  change.old\_x = x;  change.old\_y = y;  change.mouse\_state = down;  if (x > 724 && x < 768 && y>4 && y < 42)// && interVal < 300)  change.angle\_y -= 3;  if (x > 722 && x < 769 && y>827 && y < 866)// && interVal < 300)  change.angle\_y += 3;  if (x > 9 && x < 53 && y>412 && y < 457)// && interVal < 300)  change.angle\_x -= 3;  if (x > 1441 && x < 1485 && y>414 && y < 455)// && interVal < 300)  change.angle\_x += 3;  }  else if (state == GLUT\_UP)  change.mouse\_state = up;  }  else if (button == GLUT\_RIGHT\_BUTTON)  change = copy\_;  else if (button == 3)  change.scale += 0.05;  else if (button == 4)  change.scale -= 0.05;  }  void MouseMotionEvent(int x, int y)//mouse motion event  {  if (change.mouse\_state == down)  {  change.angle\_x += (x - change.old\_x)\*0.3;  change.angle\_y += (y - change.old\_y)\*0.3;  change.old\_x = x;  change.old\_y = y;  }  }  **光照部分：**   1. 流程图：      1. 核心代码：   这段代码用于初始化光照和材质属性。下面对其进行详细描述：   1. 定义光照的属性： 2. 定义光源1的位置为 (0, 5, 2)，环境光为黑色，漫反射光为白色，镜面反射光为白色。 3. 定义光源2的位置为 (0, -5, 2)，环境光为黑色，漫反射光为白色，镜面反射光为白色。 4. 定义材质的属性：  * 生成随机颜色的材质属性，使用均匀分布随机数生成器生成0到1之间的三个随机数作为材质的漫反射颜色。材质的环境光、镜面反射光、发射光、高光度也有相应的定义。  1. 设置光源和材质属性： 2. 使用 glLightfv 函数设置光源0和光源1的属性，包括位置、环境光、漫反射光和镜面反射光。 3. 使用 glMaterialfv 函数设置材质的属性，包括环境光、漫反射光、镜面反射光、发射光和高光度。 4. 启用光照和深度测试：  * 使用 glEnable 函数分别启用光照、光源0和光源1，并启用深度测试。  1. 设置光照模型：  * 使用 glShadeModel 函数设置光照模型为平滑着色模式。  1. 设置背景颜色：  * 使用 glClearColor 函数设置背景颜色为白色。   这段代码实现了光照和材质的初始化，并设置了相应的属性。通过启用光照和深度测试，以及定义材质属性，可以实现光照对物体的影响和渲染效果的改变。  void light\_init()//init the light  {  GLfloat light\_position1[] = {0,5,2,1};  GLfloat light\_ambient1[] = { 0,0,0,1 };  GLfloat light\_diffuse1[] = { 1,1,1,1 };  GLfloat light\_specular1[] = { 1,1,1,1 };    GLfloat light\_position2[] = { 0,-5,2,1 };  GLfloat light\_ambient2[] = { 0,0,0,1 };  GLfloat light\_diffuse2[] = { 1,1,1,1 };  GLfloat light\_specular2[] = { 1,1,1,1 };  //GLfloat material\_ambient[] = { 0,0,0.2,1 };  //GLfloat material\_diffuse[] = { 0.753, 0.753, 0.753,1 };  //GLfloat material\_specular[] = { 1,1,1,1 };  //GLfloat material\_emission[] = { 0,0,0,1 };  //GLfloat material\_shininess[] = { 30.0 };    random\_device rand;  mt19937 gen(rand());  uniform\_real\_distribution<float> dis(0, 1);  float r = dis(gen) + 0.0f;  float g = dis(gen) + 0.0f;  float b = dis(gen) + 0.0f;  GLfloat material\_ambient[] = { 0,0,0,1 };  GLfloat material\_diffuse[] = { r, g, b,1 };  GLfloat material\_specular[] = { 1,1,1,1 };  GLfloat material\_emission[] = { 0,0,0,1 };  GLfloat material\_shininess[] = { 30.0 };  glLightfv(GL\_LIGHT0, GL\_POSITION, light\_position1);  glLightf(GL\_LIGHT0, GL\_LINEAR\_ATTENUATION, 0.08);  glLightfv(GL\_LIGHT0, GL\_AMBIENT, light\_ambient1);  glLightfv(GL\_LIGHT0, GL\_DIFFUSE, light\_diffuse1);  glLightfv(GL\_LIGHT0, GL\_SPECULAR, light\_specular1);  glLightfv(GL\_LIGHT1, GL\_POSITION, light\_position2);  glLightfv(GL\_LIGHT1, GL\_AMBIENT, light\_ambient2);  glLightf(GL\_LIGHT1, GL\_LINEAR\_ATTENUATION, 0.08);  glLightfv(GL\_LIGHT1, GL\_DIFFUSE, light\_diffuse2);  glLightfv(GL\_LIGHT1, GL\_SPECULAR, light\_specular2);  glMaterialfv(GL\_FRONT, GL\_AMBIENT, material\_ambient);  glMaterialfv(GL\_FRONT, GL\_DIFFUSE, material\_diffuse);  glMaterialfv(GL\_FRONT, GL\_SPECULAR, material\_specular);  glMaterialfv(GL\_FRONT, GL\_EMISSION, material\_emission);  glMaterialfv(GL\_FRONT, GL\_SHININESS, material\_shininess);  glEnable(GL\_LIGHTING);  glEnable(GL\_LIGHT0);  glEnable(GL\_LIGHT1);  glEnable(GL\_DEPTH\_TEST);  glShadeModel(GL\_SMOOTH);  glClearColor(1, 1, 1, 1);  }  **测试结果部分：**  由于采用了随机材质，故每次结果的颜色不同，截图只能展示静态效果，难以展现动态的交互效果 |
| **5.实验结论与分析:**   1. 网格模型加载与显示：通过读取 OFF 格式的模型文件，成功加载了网格模型的顶点数据和三角面片数据。使用 OpenGL 的绘制函数，将模型渲染到屏幕上实现了显示功能。模型被放置在坐标原点处，通过透视投影方法，实现了透视效果。 2. 光强插值的局部光照模型：根据顶点周围一邻域的面片法线信息，计算了每个顶点的法向量。通过面片的边进行叉乘运算，得到了顶点的法向量。在渲染过程中，通过对顶点法向量的插值，实现了局部光照模型，使得模型在不同顶点处有不同的光照效果。 3. 交互操作：通过在模型的四个方向上绘制了四个不同颜色的色块，实现了交互操作。用户可以通过选择其中某个色块，获取其颜色信息。当用户选择了色块时，调用变换函数改变物体的位置，实现了对模型位置的调整。 4. 使用 glReadPixels 函数：通过使用 glReadPixels 函数，可以读取颜色缓存中指定像素的颜色值。在本作业中，使用该函数来获取用户选择的色块的颜色信息。通过指定像素的坐标和其他参数，成功获取了颜色缓存中指定像素的颜色。   通过以上功能的实现，该大作业达到了预期目标。模型加载与显示、局部光照模型、交互操作和像素颜色获取等功能的结合，使得模型展示更加真实，并提供了用户与模型交互的方式。此外，通过使用 OpenGL 的相关函数和技术，提高了对 OpenGL 编程环境的熟悉程度，并具备了处理网格模型的能力。 |
| **学生签名: 提交日期：年 月 日** |