山东大学 计算机科学与技术 学院

计算机图形学 课程实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：201300130043 | 姓名： 李园 | | 班级： 13级计算机4班 |
| 实验题目： 实验二 实现二维平面的基本变换 | | | |
| 实验学时：2 | | 实验日期： 2016年4月27日 | |
| 实验目的：  1. 掌握配置OpenGL环境的方法；  2. 熟悉OpenGL应用程序基本架构；  3. 学习OpenGL中的gl、glu、glut等库函数进行图形学算法编程和实现。 | | | |
| 硬件环境：  PC电脑 | | | |
| 软件环境：  操作系统：Windows 10  编程软件：Visual Studio 2015 | | | |
| 实验步骤与内容：  在实验一的基础上，完成以下实验内容：   1. 在二维棋盘上，自行设计并实现棋盘标志的绘制（如山东大学校徽标志）。   基本思想：  上面的山和云运用多边形扫描线算法（即用AET数据结构）填充，鉴于其有各种缺点，所以改进边的属性设置，增设y的变化量。下面的水花用种子填充画法（即用种子点确认左右边界，八连通区域填充提高效率）实现。  其中校徽坐标的确认来自于ps的像素拾取。  关键代码如下：  //点结构体  struct Point  {  int x; //x坐标  int y; //y坐标  };  //线结构体  struct Line  {  Point high\_point; //高端点  Point low\_point; //低端点  int is\_active; //是否为有效边，水平边（0），非水平边（1）  double inverse\_k; //斜率k的倒数  };  //边结点  struct EdgeNode  {  double x; //扫描线与边交点的x坐标（边的低端点的x坐标）  int y\_max; //边的高端点的y坐标ymax  double inverse\_k; //斜率k的倒数  EdgeNode \*next; //下一个边结点的指针  };  //有效边表  struct ActiveEdgeTable  {  int y; //扫描线y  EdgeNode \*head; //边链表的头指针  };  //桶结点  typedef struct Bucket  {  int y; //扫描线y  EdgeNode \*head; //边链表的头指针  Bucket \*next; //下一个桶的指针  } EdgeTable;  //比较2个点的高度  int compare(Point p1, Point p2)  {  if (p1.y > p2.y)  return 1;  else if (p1.y == p2.y)  return 0;  return -1;  }  //由点数组生成线段数组  Line\* create\_lines(Point points[], int n)  {  Line \*lines = (Line\*)*malloc*(n \* sizeof(Line));  for (int i = 0; i < n; ++i)  {  Point p1 = points[i];  Point p2 = points[(i + 1) % n];  int result = compare(p1, p2);  if (result == 0)  lines[i].is\_active = 0;  else  lines[i].is\_active = 1;  lines[i].high\_point = result > 0 ? p1 : p2;  lines[i].low\_point = result < 0 ? p1 : p2;  lines[i].inverse\_k = (double)(p2.x - p1.x) / (double)(p2.y - p1.y);  }  return lines;  }  //获取线数组中最低的端点  Point get\_lowest\_point(Line lines[], int n)  {  Point lowest\_point = lines[0].low\_point;  for (int i = 1; i < n; ++i)  {  Point low\_point = lines[i].low\_point;  if (compare(lowest\_point, low\_point) > 0)  lowest\_point = low\_point;  }  return lowest\_point;  }  //获取线数组中最高的端点  Point get\_highest\_point(Line lines[], int n)  {  Point highest\_point = lines[0].high\_point;  for (int i = 1; i < n; ++i)  {  Point high\_point = lines[i].high\_point;  if (compare(highest\_point, high\_point) < 0)  highest\_point = high\_point;  }  return highest\_point;  }  //交换2个Line对象  void swap(Line &l1, Line &l2)  {  Line temp = l1;  l1 = l2;  l2 = temp;  }  //对线数组进行排序  void sort(Line lines[], int n)  {  //先按低端点的y坐标进行升序排序  for (int i = 0; i < n; ++i)  {  int min\_index = i;  for (int j = i + 1; j < n; ++j)  {  if (lines[j].low\_point.y < lines[min\_index].low\_point.y)  min\_index = j;  }  swap(lines[i], lines[min\_index]);  }  //再将有序数组按低端点的x坐标升序排列，若x坐标相等，按inverse\_k升序  for (int i = 0; i < n; ++i)  {  int min\_index = i;  for (int j = i + 1; lines[j].low\_point.y == lines[i].low\_point.y; ++j)  {  if (lines[j].low\_point.x < lines[min\_index].low\_point.x)  min\_index = j;  }  swap(lines[i], lines[min\_index]);  if (i > 0 && lines[i].low\_point.x == lines[i - 1].low\_point.x)  {  if (lines[i].is\_active == 1 && lines[i - 1].is\_active == 1)  {  if (lines[i].inverse\_k < lines[i - 1].inverse\_k)  swap(lines[i], lines[i - 1]);  }  }  }  }  //创建一个边表  EdgeTable\* create\_edge\_table(Line lines[], int n)  {  EdgeTable \*edge\_table = (EdgeTable\*)*malloc*(sizeof(EdgeTable));  edge\_table->head = *NULL*;  edge\_table->next = *NULL*;  sort(lines, n);  Point lowest\_point = get\_lowest\_point(lines, n);  Point highest\_point = get\_highest\_point(lines, n);  EdgeTable \*s = edge\_table;  for (int i = lowest\_point.y; i <= highest\_point.y; ++i)  {  Bucket \*bucket = (Bucket\*)*malloc*(sizeof(Bucket));  bucket->y = i;  bucket->next = *NULL*;  bucket->head = (EdgeNode\*)*malloc*(sizeof(EdgeNode));  bucket->head->next = *NULL*;  EdgeNode \*p = bucket->head;  for (int j = 0; j < n; ++j)  {  if (lines[j].is\_active == 0)  continue;  if (lines[j].low\_point.y == i)  {  EdgeNode \*q = (EdgeNode\*)*malloc*(sizeof(EdgeNode));  q->x = lines[j].low\_point.x;  q->y\_max = lines[j].high\_point.y;  q->inverse\_k = lines[j].inverse\_k;  q->next = *NULL*;  p->next = q;  p = q;  }  }  s->next = bucket;  s = bucket;  }  return edge\_table;  }  //从边表中取出第一个不为空的桶初始化有效边表  ActiveEdgeTable\* init\_active\_table(EdgeTable \*edge\_table)  {  ActiveEdgeTable \*active\_table = (ActiveEdgeTable\*)*malloc*(sizeof(ActiveEdgeTable));  active\_table->y = edge\_table->next->y;  active\_table->head = (EdgeNode\*)*malloc*(sizeof(EdgeNode));  active\_table->head->next = *NULL*;  EdgeNode \*p = edge\_table->next->head;  EdgeNode \*q = active\_table->head;  while (p->next != *NULL*)  {  EdgeNode \*s = (EdgeNode\*)*malloc*(sizeof(EdgeNode));  s->x = p->next->x;  s->y\_max = p->next->y\_max;  s->inverse\_k = p->next->inverse\_k;  s->next = *NULL*;  q->next = s;  q = s;  p = p->next;  }  return active\_table;  }  //从有效边表中删除指定y\_max的边结点  void delete\_edge(ActiveEdgeTable \*active\_table, int y\_max)  {  EdgeNode \*p = active\_table->head;  while (p->next != *NULL*)  {  EdgeNode \*q = p->next;  if (q->y\_max == y\_max)  {  p->next = q->next;  *free*(q);  }  else  p = p->next;  }  }  //将一个边结点按次序添加到有效边表中  void add\_edge(ActiveEdgeTable \*active\_table, EdgeNode edge)  {  EdgeNode \*t = (EdgeNode\*)*malloc*(sizeof(EdgeNode));  t->x = edge.x;  t->y\_max = edge.y\_max;  t->inverse\_k = edge.inverse\_k;  t->next = *NULL*;  EdgeNode \*p = active\_table->head;  while (p->next != *NULL*)  {  EdgeNode \*q = p->next;  if ((edge.x < q->x) || (edge.x == q->x && edge.inverse\_k < q->inverse\_k))  {  p->next = t;  t->next = q;  return;  }  p = p->next;  }  p->next = t;  }  //更新有效边表，并与边表中对应的桶合并  ActiveEdgeTable\* update\_active\_table(ActiveEdgeTable \*active\_table, EdgeTable \*edge\_table)  {  //更新扫描线y  ++active\_table->y;  //删除y=ymax的边  delete\_edge(active\_table, active\_table->y);  //更新边结点的数据  EdgeNode \*p = active\_table->head->next;  while (p != *NULL*)  {  p->x += p->inverse\_k;  p = p->next;  }  //找到边表中对应的桶  EdgeTable \*q = edge\_table;  while ((q = q->next) != *NULL* && q->y != active\_table->y);  //如果找到，则进行合并  if (q != *NULL*)  {  EdgeNode \*s = q->head;  while ((s = s->next) != *NULL*)  {  add\_edge(active\_table, \*s);  }  }  return active\_table;  }  //画出多边形的边框  void DrawPolygon(Point points[], int n)  {  *glBegin*(*GL\_LINE\_LOOP*);  for (int i = 0; i < n; ++i)  *glVertex3f*(points[i].x, points[i].y,0);  *glEnd*();  }  //用指定的像素大小填充多边形  void Fill(Point points[], int n)  {  Line \*lines = create\_lines(points, n);  EdgeTable \*edge\_table = create\_edge\_table(lines, n);  ActiveEdgeTable \*active\_table = init\_active\_table(edge\_table);  while (active\_table->head->next != *NULL*)  {  EdgeNode \*p = active\_table->head;  int b = -1;  while (p->next != *NULL*)  {  if (b > 0)  {  int left = p->x;  int right = p->next->x;  //如果不是局部最低点，则进行边界处理  if (!(p->x - p->next->x >= -EPSILON && p->x - p->next->x <= EPSILON))  {  //处理左边界  if (!(p->x - left >= -EPSILON && p->x - left <= EPSILON))  left += 1;  //处理右边界  if (p->next->x - right >= -EPSILON && p->next->x - right <= EPSILON)  right -= 1;  }  for (int i = left; i <= right; ++i)  {  *glBegin*(*GL\_POINTS*);  *glVertex2d*(i, active\_table->y);  *glEnd*();  *glFlush*();  }  }  p = p->next;  b = -b;  }  active\_table = update\_active\_table(active\_table, edge\_table);  }  }  //扫描线种子（或称边界）填充算法  void glPoint(int x, int y, int r, int g, int b) {  *glColor3ub*(r, g, b);  *glPointSize*(1);  *glBegin*(*GL\_POINTS*);  *glVertex2i*(x, y);  *glEnd*();  *glFlush*();  }  bool sameColor(int r1, int g1, int b1, int r2, int g2, int b2) {  //容差度  int dis = 1;  if (*fabs*(r1 - r2) <= dis&&*fabs*(g1 - g2) <= dis&&*fabs*(b1 - b2) <= dis) {  return true;  }  else {  return false;  }  }  *GLubyte* oldColor[3] = { 255, 255, 255 };  *GLubyte* borderColor[3] = { 255,0,0 };  int halfWidth, halfHeight;  void zzFill(int startX, int startY, int r, int g, int b) {  *stack*<Point> pixelStack;  //x,y是给定的种子像素点，rgb就是要填充的颜色的RGB值  Point point = { startX, startY };  pixelStack.*push*(point);  int saveX;  int xRight, xLeft;  int x, y;  //如果栈不为空  while (!pixelStack.*empty*()) {  //获取最顶端的元素  Point tempPoint = pixelStack.*top*();  //删除最顶端的元素  pixelStack.*pop*();  saveX = tempPoint.x;  x = tempPoint.x;  y = tempPoint.y;  *GLubyte* iPixel[3];  *glReadPixels*(x + halfWidth, y + halfHeight, 1, 1, *GL\_RGB*, *GL\_UNSIGNED\_BYTE*, &iPixel);  //如果没有到达右边界，就填充  while (!sameColor(iPixel[0], iPixel[1], iPixel[2], borderColor[0],  borderColor[1], borderColor[2])) {  glPoint(x, y, r, g, b);  x = x + 1;  *glReadPixels*(x + halfWidth, y + halfHeight, 1, 1, *GL\_RGB*, *GL\_UNSIGNED\_BYTE*, &iPixel);  *printf*("r:%d,g:%d,b:%d\n", iPixel[0], iPixel[1], iPixel[2]);  }  xRight = x - 1;  x = saveX - 1;  *glReadPixels*(x + halfWidth, y + halfWidth, 1, 1, *GL\_RGB*, *GL\_UNSIGNED\_BYTE*, &iPixel);  //如果没有到达左边界，就填充  while (!sameColor(iPixel[0], iPixel[1], iPixel[2], borderColor[0],  borderColor[1], borderColor[2])) {  glPoint(x, y, r, g, b);  x = x - 1;  *glReadPixels*(x + halfWidth, y + halfWidth, 1, 1,  *GL\_RGB*, *GL\_UNSIGNED\_BYTE*, &iPixel);  }  //保存左端点  xLeft = x + 1;  //从右边的点开始  x = xRight;  //检查上端的扫描线  y = y + 1;  while (x >= xLeft) {  *glReadPixels*(x + halfWidth, y + halfWidth, 1, 1,  *GL\_RGB*, *GL\_UNSIGNED\_BYTE*, &iPixel);  if (!sameColor(iPixel[0], iPixel[1], iPixel[2], borderColor[0],  borderColor[1], borderColor[2]) && !sameColor(iPixel[0],  iPixel[1], iPixel[2], r, g, b)) {  //如果上方的点不是边界点，直接压入  Point p = { x, y };  pixelStack.*push*(p);  //压入之后停止循环  break;  }  else {  x--;  *glReadPixels*(x + halfWidth, y + halfWidth, 1, 1, *GL\_RGB*,  *GL\_UNSIGNED\_BYTE*, &iPixel);  }  }  //检查下端的扫描线  y = y - 2;  //从右边的点开始  x = xRight;  while (x >= xLeft) {  *glReadPixels*(x + halfWidth, y + halfWidth, 1, 1,  *GL\_RGB*, *GL\_UNSIGNED\_BYTE*, &iPixel);  if (!sameColor(iPixel[0], iPixel[1], iPixel[2],  borderColor[0], borderColor[1], borderColor[2])  && !sameColor(iPixel[0], iPixel[1], iPixel[2], r, g, b)) {  //如果上方的点不是边界点，直接压入  Point p = { x, y };  //压入之后停止循环  pixelStack.*push*(p);  break;  }  else {  x--;  *glReadPixels*(x + halfWidth, y + halfWidth, 1, 1, *GL\_RGB*, *GL\_UNSIGNED\_BYTE*, &iPixel);  }  }  }  }  在display对象中  //绘制棋盘标志，200<x<300 225<y<275 z=0  //外圈  *glColor3ub*(138, 2, 2);  *glPointSize*(1.0);  for (int j = 1; j < 9; j++)  {  bresenhamArc(0.05\*h, 0, 0, j);  }  *glFlush*();  int x0 = -0.05\*h, y0 = 0.05\*h;  double k = 0.0001;    //不镂空的大山  //多边形的顶点坐标  *Point* points[] = { { 81, 442 },{ 81, 384 },{ 208, 384 },{ 293, 213 },{ 387, 298 },  { 509, 69 },{ 616, 281 },{ 667,192 },{ 792,435 },{ 879,435 },{ 879,493 },{ 822,493 },  { 838,525 },{ 773,526 },{ 757,495 },{ 706,495 },{ 716,519 },{ 638,522 },  { 626,495 },{ 571,495 } ,{ 576,604 },{ 474,591 },{ 474,506 },{ 428,505 },  { 463,442 },{ 389,442 },{ 356,499 },{ 293,490 },{ 314,442 },{ 252,442 },  { 226,495 },{ 154,494 },{ 179,442 }  };  //计算顶点个数  int n = sizeof(points) / sizeof(*Point*);  for (int i = 0; i < n;i++)  {  points[i].x = points[i].x\*k\*h +x0;  points[i].y = -points[i].y\*k\*h+y0;  }  //使用黑色画出多边形的边框  *glColor3ub*(156, 12, 19);  DrawPolygon(points, n);  *glFlush*();  //指定点大小  *glPointSize*(1.0f);  //使用红色填充多边形  *glColor3ub*(156, 12, 19);  Fill(points, n);  *glFlush*();  //大山的镂空部分  //镂空1 多边形的顶点坐标  *Point* space1[] = { { 281,382 },{ 317,315 },{ 354,348 },{ 339,383 } };  *Point* space2[] = { { 419,382 },{ 511,209 },{ 581,383 },{ 541,383 },  { 541,320 },{ 502,322 },{ 502,383 } };  *Point* space3[] = { { 556,435 },{ 582,389 },{ 603,434 } };  *Point* space4[] = { { 682,436 },{ 643,342 },{ 670,321 },{ 727,436 } };  //计算顶点个数  int n1 = sizeof(space1) / sizeof(*Point*);  int n2 = sizeof(space2) / sizeof(*Point*);  int n3 = sizeof(space3) / sizeof(*Point*);  int n4 = sizeof(space4) / sizeof(*Point*);  for (int i = 0; i < n1; i++)  {  space1[i].x = space1[i].x\*k\*h+x0;  space1[i].y = -space1[i].y\*k\*h+y0;  }  for (int i = 0; i < n2; i++)  {  space2[i].x = space2[i].x\*k\*h + x0;;  space2[i].y = -space2[i].y\*k\*h+y0;  }  for (int i = 0; i < n3; i++)  {  space3[i].x = space3[i].x\*k\*h+x0;  space3[i].y = -space3[i].y\*k\*h+y0;  }  for (int i = 0; i < n4; i++)  {  space4[i].x = space4[i].x\*k\*h+x0;  space4[i].y = -space4[i].y\*k\*h+y0;  }  //使用黑色画出多边形的边框  *glColor3ub*(0, 0, 0);  DrawPolygon(space1, n1);  DrawPolygon(space2, n2);  DrawPolygon(space3, n3);  DrawPolygon(space4, n4);  *glFlush*();  //指定点大小  *glPointSize*(1.0f);  //使用黑色（或之后改为背景色）填充多边形  *glColor3ub*(0, 0, 0);  Fill(space1, n1);  Fill(space2, n2);  Fill(space3, n3);  Fill(space4, n4);  *glFlush*();  //water flower  *glColor3ub*(156, 12, 19);  *glPointSize*(2.0);  bresenhamArc(20, 0, 0, 4);  bresenhamArc(20, 0, 0, 5);  bresenhamArc(20, 25, 0, 4);  bresenhamArc(20, 25, 0, 5);  bresenhamArc(20, -25, 0, 4);  bresenhamArc(20, -25, 0, 5);  bresenhamArc(20, 50, 0, 4);  bresenhamArc(20, 50, 0, 5);  bresenhamArc(20, -50, 0, 4);  bresenhamArc(20, -50, 0, 5);  *glPointSize*(1.0);  bresenham\_line(-37, -30, -37, -50);  bresenham\_line(-16, -30, -16, -40);  bresenham\_line(-8, -30, -8, -50);  bresenham\_line(-16, -30, -8, -30);  bresenham\_line(-16, -40, -8, -40);  bresenham\_line(-16, -50, -8, -50);  bresenham\_line(8, -30, 8, -50);  bresenham\_line(16, -30, 16, -50);  bresenham\_line(8, -30, 16, -30);  bresenham\_line(8, -50, 16, -50);  bresenham\_line(37, -30, 37, -50);  *glFlush*();  实现效果如下：    2. 实现对棋盘标志的平移、旋转和放缩功能。  针对平移，放缩和旋转功能只需关心顶点的转换公式：  x = arr[i][0]\*k\*h+x0;  y = -arr[i][1]\*k\*h+y0;  中K为放缩因子，x0,y0为相对坐标原点。  想要放缩只需要适当改变k的大小，平移需改变和原点的相对距离，旋转则改变x,y的正负。 | | | |
| 结论分析与体会：  经过了对多边形扫描线填充算法的改进和对种子填充算法中八连通区域概念的应用，不仅提高了代码效率，还加深了我对空间凸多边形填充和平移等变换的理解和应用。 | | | |