

计算机图形学实验

实验三

学 号： 20191130038

姓 名： 郑广敏

专 业： 智能科学与技术

年 级： 2019级

任课教师： 吴昊

2020 年 10 月 23 日

**一、目的和要求：**

通过实验掌握下列知识:

1. 熟悉OpenGL中对颜色的设置
2. 边界填充算法的理解与实现；
3. 泛滥填充算法的理解与实现；
4. 扫描线填充算法的理解与实现；

**二、实验内容：**

## 一、OpenGL中颜色的设置

OpenGL支持两种颜色模式：一种是RGBA，一种是颜色索引模式。无论哪种颜色模式，计算机都必须为每一个像素保存一些数据。不同的是，RGBA模式中，数据直接就代表了颜色值；而颜色索引模式中，数据代表的是一个颜色表的索引，要得到真正的颜色，还必须到颜色表去查找。 颜色索引模式主要用于早期性能差，只能显示有限颜色的显示设备上，现在由于显示设备的发展，普遍都采用RGBA模式，所以，我们只介绍RGBA模式。

RGBA模式中，每一个像素会保存以下数据：R值（红色分量）、G值（绿色分量）、B值（蓝色分量）和A值（alpha分量，就是不透明度分量，数值最大时完全不透明）。其中红、绿、蓝三种颜色相组合，就可以得到我们所需要的各种颜色，而alpha不直接影响颜色，它将留待以后介绍。

在RGBA模式下选择颜色是十分简单的事情，只需要一个函数。glColor\*系列函数可以用于设置当前绘制颜色（前景色），其中三个参数的版本可以指定R、G、B的值，而A值采用默认；四个参数的版本可以分别指定R、G、B、A的值。例如：

void glColor3<b i f d ub us ui>(type r, type g, type b)

void glColor3<b i f d ub us ui>v(type \* color)

void glColor4<b i f d ub us ui>(type r, type g, type b, type a)

void glColor4<b i f d ub us ui>v(type \* color)

用标准类型来指定RGB或RGBA颜色值。如果函数名的末尾出现v，则颜色值由color所指向的数组指定，该数组中每个元素的类型均为type。

将浮点数作为参数，其中0.0表示不使用该种颜色，而1.0表示将该种颜色用到最多。例如：

glColor3f(1.0f, 0.0f, 0.0f); 表示不使用绿、蓝色，而将红色使用最多，于是得到最纯净的红色。

glColor3f(0.0f, 1.0f, 1.0f); 表示使用绿、蓝色到最多，而不使用红色，混合的效果就是浅蓝色。

glColor3f(0.5f, 0.5f, 0.5f); 表示各种颜色使用一半，效果为灰色。

glColor3f(0.0f, 0.0f, .0.0f); 表示纯黑。

glColor3f(1.0f, 1.0f, .1.0f); 表示纯白。

用数组时，例如：

float c[3]={ 1.0f, 0.0f, 0.0f }; glColor3fv(c);

与glColor3f（1.0f, 0.0f, 0.0f）;等价，都是指定当前绘制颜色是红色。

注意：浮点数可以精确到小数点后若干位，这并不表示计算机就可以显示如此多种颜色。实际上，计算机可以显示的颜色种数将由硬件决定。如果OpenGL找不到精确的颜色，会进行类似“四舍五入”的处理。

注意：glColor系列函数，在参数类型不同时，表示“最大”颜色的值也不同。

采用f和d做后缀的函数，以1.0表示最大的使用。

采用ub或i做后缀的函数，以255表示最大的使用。

以下两种颜色相同，都表示纯蓝：

glColor3f(0.0f,0.0f,1.0f);

glColor3i(0,0,255) ;

另外，我们可以用如下函数设定清除颜色，也就是背景颜色：

glClearColor(float , float, float, float);

四个参数分别就是RGBA四个通道的值，表示方法和glColor4f相同，例如：

glClearColor(1.0f, 1.0f, 1.0f, 1.0f); 把背景设为白色；

glClearColor(0.0f, 0.0f, 0.0f, 1.0f); 把背景设为黑色；

glClearColor(1.0f, 0.0f, 0.0f, 1.0f); 把背景设为纯红；

在OpenGL程序初始化的时候，会用到一个函数：

glClear(GL\_COLOR\_BUFFER\_BIT);

这个函数是把屏幕上的颜色全部清空，变为glClearColor指定的背景颜色。GL\_COLOR\_BUFFER\_BIT是头文件里定义的一个系统常量。

在OpenGL中，颜色是OpenGL状态的一部分。我们可以认为在OpenGL的状态变量中存在一种当前绘制颜色(current drawing color)和当前清除颜色(current clear color)，其中前者由函数glColor\*()来指定，后者由glClearColor()来指定。用于绘制对象的颜色即为当前绘制颜色。

当要绘制图形时，所用颜色从当前绘制颜色(current drawing color)中读取；当要清空屏幕时，所用颜色从当前清除颜色(current clear color)中读取。所以，颜色与对象并不存在依附关系，而是作为OpenGL的内部状态。虽然在程序中看起来好像颜色与对象和顶点结合在一起，但实际上在程序定义顶点时，OpenGL是依据当前状态来查找当前颜色的。也就是说，不存在红色的点，蓝色的线，只是说，在绘制该点时，当前绘制颜色是红色，在绘制该条线时，当前颜色是蓝色。程序中哪些地方修改了当前颜色，我们必须特别留意。

## 二、实现边界填充算法

读懂如下代码，理解执行的原理：

void boundaryFill4 (int x, int y, int fillColor, int borderColor)

{

int interiorColor;

/\* Set current color to fillColor, then perform following oprations. \*/

getPixel (x, y, interiorColor);

if ((interiorColor != borderColor) && (interiorColor != fillColor)) {

setPixel (x, y); // Set color of pixel to fillColor.

boundaryFill4 (x + 1, y , fillColor, borderColor);

boundaryFill4 (x - 1, y , fillColor, borderColor);

boundaryFill4 (x , y + 1, fillColor, borderColor);

boundaryFill4 (x , y - 1, fillColor, borderColor)

}

}

该函数中，实际上就是用某种颜色fillColor把一副图像中的另外一种颜色borderColor作为边界的图像部分进行填充。getPixel (x, y, interiorColor)是取当前像素（x，y）的颜色，颜色值保存到interiorColor变量里面，然后判断当前像素颜色interiorColor是否与边界颜色borderColor相等，如果相等，表明已经到达边界，不进行填充；再判断当前像素颜色interiorColor是否与填充颜色fillColor相等，如果相等，表明已经进行了填充，不需要再次填充；如果当前像素颜色interiorColor与边界颜色borderColor和填充颜色fillColor都不相等，表明当前像素是边界内部的未填充像素，就用setPixel (x, y)对该像素进行填充颜色fillColor，并继续递归填充当前像素的的上下左右四个像素。

注意：

（1）代码没有主函数及其它OpenGL的初始化函数，请用前面实验1中的代码进行添加，该填充函数boundaryFill4 (int x, int y, int fillColor, int borderColor)在回调函数myDraw()中调用执行；

（2）在myDraw()中调用执行填充函数boundaryFill4 (int x, int y, int fillColor, int borderColor)之前，需要我们用边界颜色borderColor指定一个边界，并且指定填充的第一点，该点的坐标即为myDraw()中调用的填充函数boundaryFill4 (int x, int y, int fillColor, int borderColor)的x和y的值，该点必须在指定边界的内部，填充颜色fillColor也需要我们指定。指定边界可用

glBegin(GL\_LINE\_LOOP);

……

glEnd();

指定一个闭合区域。

注意该区域不要太大，否则递归执行boundaryFill4时，要用到的堆栈数量非常巨大，系统可能发生堆栈溢出的错误。解决的方法就是把递归程序改写为非递归程序。

（3）函数中的颜色fillColor和borderColor的类型为int，但OpenGL中的RGB颜色需要3个0.1~1.0的浮点数指定或3个0~255的整数指定，所以我们需要重新定义颜色类型，例如，定义颜色类型Color为一个浮点数组或一个结构体：

例如：

typedef float Color[3];

这时，Color类型为长度是3的浮点数组，如果定义Color的变量

Color fillColor;

那么，fillColor[0], fillColor[10], fillColor[2]就可以分别表示RGB三个通道的值。

（4）getPixel (x, y, interiorColor)是取当前像素（x，y）的颜色，颜色值保存到interiorColor变量里面，由于getPixel不是一个系统函数，所以需要我们自己定义，借助OpenGL函数glReadPixels完成，该函数原型如下：

glReadPixels(xmin, ymin, width, height, dataFormat, dataType, array)

作用是把从（xmin, ymin）位置开始，大小为宽width, 高height的一块的像素值读入，依次保存到数组array中。像素值类型由dataFormat,和dataType指定。

dataFormat的可用类型有GL\_COLOR\_INDEX, GL\_STENCIL\_INDEX, GL\_DEPTH\_COMPONENT, GL\_RED, GL\_GREEN, GL\_BLUE, GL\_ALPHA, GL\_RGB, GL\_RGBA, GL\_LUMINANCE和GL\_LUMINANCE\_ALPHA。

dataType的可用类型有GL\_UNSIGNED\_BYTE, GL\_BYTE, GL\_BITMAP, GL\_UNSIGNED\_SHORT, GL\_SHORT, GL\_UNSIGNED\_INT, GL\_INT和GL\_FLOAT。

getPixel()可定义为

void getPixel(GLint x, GLint y, Color c)

{

glReadPixels(x,y,1,1,GL\_RGB,GL\_FLOAT,c);

}

表示取从（x，y）开始的一个像素的值，值的类型为用FLOAT 表示的RGB颜色。c为数组名，相当于一个指针，可双向传递。

（5）函数中的颜色值比较(interiorColor != borderColor) && (interiorColor != fillColor)，由于颜色类型改为Color，故颜色比较的函数需要重写，例如：

bool rgbColorEqual(Color c1,Color c2)

{

//补充代码，当三个通道都相等时，返回真，否则返回假；

}

但是浮点数的比较不能直接用“= =”,因为浮点数在内存中存储时会有误差，如3.14可能存为3.1399999999，也可能存为3.140000001，用“= =”比较时，3.1399999999 ！=3.140000001。方法是比较的两个数想减，绝对值小于一个很小的正整数时，就判定两个数相等。如a和b是两个浮点数，那么当abs (a – b ) < 0.001,就判定a等于b。这一点，在所有的程序设计里面都是要遵循的规则。

这时，两个比较可以改为

!(rgbColorEqual(interiorColor,borderColor)) && !(rgbColorEqual(interiorColor,fillColor))

（6）代码中调用setPixel(int , int)函数，该函数不是C或C++的系统函数，需要我们自己去定义，定义方法见实验二。

（7）由于填充速度很快，如果直接运行，我们看不到填充的过程，所以我们需要放慢填充速度，可以每填充一个像素休息一段时间，调用sleep(5); ,该函数参数是休息的毫秒数。sleep()函数在头文件windows.h中定义,所以需要在源代码开头加上#include <windows.h>。

## 三、实现泛滥填充算法

根据理论部分实现泛滥填充算法：

void floodFill4 (int x, int y, int fillColor, int interiorColor)

{

//代码你们实现

}

该函数中，实际上就是用某种颜色fillColor把一副图像中的另外一种颜色interiorColor进行替换。getPixel (x, y, color)是取当前像素（x，y）的颜色，颜色值保存到color变量里面，然后判断当前像素颜色color是否与要替换的颜色interiorColor相等：如果相等，表明该像素需要进行替换，就setPixel (x, y)，把当前像素（x，y）的颜色设置为颜色fillColor，并且再对当前像素的上下左右四个像素进行替换，递归执行该程序；如果不相等，就不需要进行替换。

泛滥填充算法可以通过改动边界填充算法实现，但要注意：该函数是用某种颜色fillColor把一副图像中的另外一种颜色interiorColor进行替换，所以在myDraw()中调用执行填充函数floodFill4 (int x, int y, int fillColor, int interiorColor)之前，需要我们用颜色interiorColor指定一块待替换的区域，并且指定填充的第一点，该点的坐标即为myDraw()中调用的填充函数floodFill4 (int x, int y, int fillColor, int interiorColor)的x和y的值，该点必须在这个替换的区域内部，填充颜色fillColor也需要我们指定。替换的区域可用

glBegin(GL\_POLYGON);

……

glEnd();

指定一个多边形填充区域。

**三 关键步骤的源码和注释**

1. 边界填充

#include <GL/glut.h>

#include<windows.h>

#include<math.h>

#include<stdlib.h>

typedef float Color[3];

bool rgbColorEqual(Color c1,Color c2)

{

//补充代码，当三个通道都相等时，返回真，否则返回假；

if(abs (c1[0] - c2[0] ) < 0.001)

if(abs (c1[1] -c2[1] ) < 0.001)

if(abs (c1[2] - c2[2] ) < 0.001)

return true;

return false;

}

void getPixel(GLint x, GLint y, Color c)

{

glReadPixels(x,y,1,1,GL\_RGB,GL\_FLOAT,c);

}

void setPixel (GLint x, GLint y,Color fillcolor)

{

glColor3f(fillcolor[0],fillcolor[1],fillcolor[2]);

glBegin (GL\_POINTS);

glVertex2i (x, y);

glEnd ( );

}

void boundaryFill4 (int x, int y, Color fillColor, Color borderColor)

{

Color interiorColor;

/\* Set current color to fillColor, then perform following oprations. \*/

getPixel (x, y, interiorColor);

if ( !rgbColorEqual(interiorColor, borderColor) && !rgbColorEqual(interiorColor,fillColor)) {

setPixel (x, y,fillColor); // Set color of pixel to fillColor.

boundaryFill4 (x + 1, y , fillColor, borderColor);

boundaryFill4 (x - 1, y , fillColor, borderColor);

boundaryFill4 (x , y + 1, fillColor, borderColor);

boundaryFill4 (x , y - 1, fillColor, borderColor);

}

}

void init(void)

{//初始化函数，也可以不要

glClearColor(1.0,1.0,1.0,0.0); //背景画板白色,不透明度0

glMatrixMode (GL\_PROJECTION); //指定投影矩阵

gluOrtho2D (0.0, 400, 0.0, 300); //指定显示的区域

}

void myDisplay(void)

{// 图形绘制回调函数，通常把图形绘制代码写到这个函数里面

glClear(GL\_COLOR\_BUFFER\_BIT);

glColor3f(0.0,0.0,0.0);

glBegin(GL\_LINE\_LOOP);

glVertex2i(30,30);

glVertex2i(30, 80);

glVertex2i(130,80);

glVertex2i(130, 30);

glEnd();

Color fillColor,borderColor;

borderColor[0]=0.0f,borderColor[1]=0.0f,borderColor[2]=0.0f;

fillColor[0]=1.0f,fillColor[1]=1.0f,fillColor[2]=0.0f;

boundaryFill4 (40, 40, fillColor, borderColor);

glFlush();

}

int main(int argc, char \*argv[])

{

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_RGB | GLUT\_SINGLE);

glutInitWindowPosition(0, 0);

glutInitWindowSize(400, 300);

glutCreateWindow("OpenGL基本图元绘制");

init();

glutDisplayFunc(&myDisplay);

glutMainLoop();

return 0;

}

1. 泛滥填充

#include <GL/glut.h>

#include<windows.h>

#include<math.h>

#include<stdlib.h>

typedef float Color[3];

bool rgbColorEqual(Color c1,Color c2)

{

//补充代码，当三个通道都相等时，返回真，否则返回假；

if(abs (c1[0] - c2[0] ) < 0.001)

if(abs (c1[1] -c2[1] ) < 0.001)

if(abs (c1[2] - c2[2] ) < 0.001)

return true;

return false;

}

void getPixel(GLint x, GLint y, Color c)

{

glReadPixels(x,y,1,1,GL\_RGB,GL\_FLOAT,c);

}

void setPixel (GLint x, GLint y,Color fillcolor)

{

glColor3f(fillcolor[0],fillcolor[1],fillcolor[2]);

glBegin (GL\_POINTS);

glVertex2i (x, y);

glEnd ( );

}

void floodFill4 (int x, int y, Color fillColor, Color interiorColor)

{

Color curColor;

/\* Set current color to fillColor, then perform following oprations. \*/

getPixel (x, y, curColor);

if ( rgbColorEqual(interiorColor, curColor) ) {

setPixel (x, y,fillColor); // Set color of pixel to fillColor.

floodFill4 (x + 1, y , fillColor, interiorColor);

floodFill4 (x - 1, y , fillColor, interiorColor);

floodFill4 (x , y + 1, fillColor, interiorColor);

floodFill4 (x , y - 1, fillColor, interiorColor);

}

}

void init(void)

{//初始化函数，也可以不要

glClearColor(1.0,1.0,1.0,0.0); //背景画板白色,不透明度0

glMatrixMode (GL\_PROJECTION); //指定投影矩阵

gluOrtho2D (0.0, 400, 0.0, 300); //指定显示的区域

}

void myDisplay(void)

{// 图形绘制回调函数，通常把图形绘制代码写到这个函数里面

glClear(GL\_COLOR\_BUFFER\_BIT);

glColor3f(0.0,0.0,0.0);

glBegin(GL\_POLYGON);

glVertex2i(30,30);

glVertex2i(30, 80);

glVertex2i(130,80);

glVertex2i(130, 30);

glEnd();

Color fillColor,interiorColor;

interiorColor[0]=0.0f,interiorColor[1]=0.0f,interiorColor[2]=0.0f;

fillColor[0]=1.0f,fillColor[1]=1.0f,fillColor[2]=0.0f;

floodFill4 (40, 40, fillColor, interiorColor );

glFlush();

}

int main(int argc, char \*argv[])

{

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_RGB | GLUT\_SINGLE);

glutInitWindowPosition(0, 0);

glutInitWindowSize(400, 300);

glutCreateWindow("OpenGL基本图元绘制");

init();

glutDisplayFunc(&myDisplay);

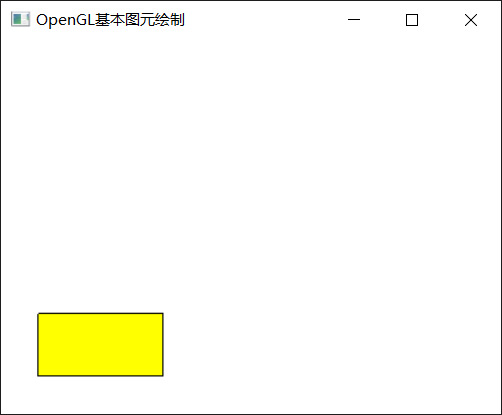
glutMainLoop();

return 0;

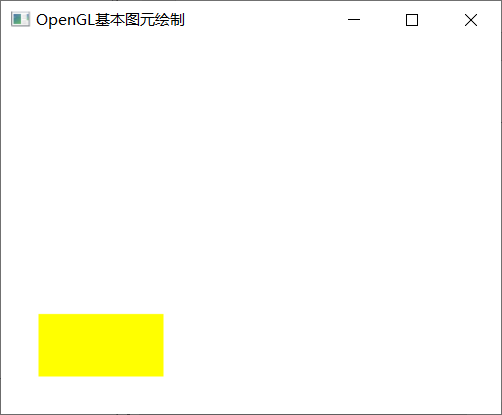
}

四 **实验结果（截图）**

**1.边界填充**



**2.泛滥填充**



**五、实验分析**

4连通填充算法，对于某些特殊图形，无法完全填充

边界填充的递归算法的优点是原理非常简单，容易理解，缺点是由于多次递归，费时，费内存，效率不高，需要大量栈空间来存储相

邻的点，太大的区域会由于递归太深而无法运行。为了减少递归次数，可采用扫描线种子算法。扫描线填充算法就是它的改进的方法。它通过沿扫描线填充水平像素段，来处理四连通或八连通相邻点，这样就仅仅只需要将每个水平像素段的起始位置压入栈，而不需要将当前位置周围尚未处理的相邻像素都压入栈，从而可以节省大量的栈空间。

而泛滥填充算法与边界填充算法的区别在于，边界填充算法是要填充到边缘停止，泛滥填充算法是填冲到颜色不同的区域位置