**《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:

|  |
| --- |
| **第一次编程实验题目:** **直线的生成** |
| **1.实验目的:** |
| 理解直线生成的原理；掌握典型直线生成算法；掌握步处理、分析实验数据的能力；  编程实现 DDA 算法、Bresenham 中点算法；对于给定起点和终点的直线，分别调用 DDA 算法和 Bresenham 中点算法进行批量绘制，并记录两种算法的绘制时间；利用 excel 等数据分析软件，将试验结果编制成表格，并绘制折线图比较两种算法的性能。 |
| **2.实验原理(**描述核心算法内容)**:** |
| DDA算法（Digital Differential Analyzer Algorithm）和Bresenham中点算法都是用于计算直线的数字化绘制算法。它们在计算机图形学中广泛应用，用于绘制直线和曲线。  DDA算法：  DDA算法是一种基本的直线绘制算法，它使用数字差分来逐点计算直线上的像素坐标。算法的基本思想是从起点到终点的线段上，以相等的步长沿着X或Y轴进行增量，然后根据斜率计算另一个坐标的增量。具体步骤如下：   1. 计算起点和终点之间的X和Y坐标的增量：dx = x2 - x1 和 dy = y2 - y1。 2. 计算步长step：如果dx > dy，则step = dx；否则step = dy。 3. 计算X和Y的增量值：x\_increment = dx / step 和 y\_increment = dy / step。 4. 循环step次，在每一步中计算新的X和Y坐标值，并绘制像素。   DDA算法的优点是简单易懂，适用于直线的任意斜率。然而，由于它使用浮点数计算，可能存在精度问题，并且在处理较大的直线时效率较低。  Bresenham中点算法：  Bresenham中点算法是一种更高效的直线绘制算法，它通过利用整数运算和增量计算来绘制直线。算法的基本思想是利用像素的中点位置来确定下一个像素的位置，从而实现直线的绘制。具体步骤如下：   1. 计算起点和终点之间的X和Y坐标的增量：dx = x2 - x1 和 dy = y2 - y1。 2. 初始化决策参数d：d = 2 \* dy - dx。 3. 循环dx次，在每一步中进行如下操作： 4. 绘制当前像素坐标(x, y)。 5. 如果d > 0，则y递增1，同时更新d：d = d + 2 \* (dy - dx)。 6. 否则，保持y不变，同时更新d：d = d + 2 \* dy。 7. x递增1。   Bresenham中点算法的优点是使用整数运算，无需浮点数计算，从而提高了计算效率。它能够准确绘制出直线，无论斜率是整数还是分数。然而，Bresenham算法只适用于绘制直线，对于曲线的绘制需要其他算法的辅助。  总的来说，DDA算法和Bresenham中点算法是两种常见的直线绘制算法，它们在计算机图形学中被广泛使用。根据具体的需求和情况，选择适合的算法可以提高绘制效率和精确度。 |
| **3.实验环境：** |
| 开发环境：Visual Studio 2022  实验平台：Experiment\_Frame\_One（自制平台） |
| **4.实验过程、步骤 (含重要函数流程图及关键代码):** |
| **DDA算法：**   1. 流程图：     2）关键代码：  //绘制像素的函数DrawPixel(x, y);  void CExperiment\_Frame\_OneView::DDA(int X0, int Y0, int X1, int Y1)  {  //----------请实现DDA算法------------//  double x = X0, y = Y0;  int dx = X1 - X0, dy = Y1 - Y0;  int steps = max(abs(dx), abs(dy));  double xIncre = double(dx) / double(steps), yIncre = double(dy) / double(steps);  for (int i = 0; i <= steps; i++)  {  DrawPixel(round(x), round(y));  x += xIncre;  y += yIncre;  }  }  3）实验结果：  测试值1：    测试结果1：    测试值2：    测试结果2：    **Bresenham中点算法：**   1. 流程图：      1. 核心代码：   void CExperiment\_Frame\_OneView::Mid\_Bresenham(int X0, int Y0, int X1, int Y1)  {  //-------请实现Mid\_Bresenham算法-------//  int dx = X1 - X0, dy = Y1 - Y0;  int stepx = dx / abs(dx), stepy = dy / abs(dy);  //dx = stepx \* dx, dy = stepy \* dy;  int x = X0, y = Y0;  if (abs(dx) > abs(dy))  {  int Di = stepx \* dx - 2 \* stepy \* dy;  while (x != X1)  {  DrawPixel(x, y);  x += stepx;  if (Di >= 0) Di = Di - 2 \* stepy \* dy;  else  {  Di = Di - 2 \* stepy \* dy + 2 \* stepx \* dx;  y += stepy;  }  }  DrawPixel(X1, Y1);  }  else  {  int Di = stepy \* dy - 2 \* stepx \* dx;  while (y != Y1)  {  DrawPixel(x, y);  y += stepy;  if (Di >= 0) Di = Di - 2 \* stepx \* dx;  else  {  Di = Di - 2 \* stepx \* dx + 2 \* stepy \* dy;  x += stepx;  }  }  DrawPixel(X1, Y1);  }  }   1. 测试结果：   测试值1：    测试结果1：    测试值2：    测试结果2：    测试值3：    测试结果3：    测试值4：    测试结果4：    **性能比较：**  1.理论分析：  DDA算法和Bresenham中点算法是两种常见的直线绘制算法，在性能方面有一些差异。以下是它们的性能比较：  1）计算效率：  DDA算法：DDA算法使用浮点数计算来计算每个像素的坐标，因此在计算效率方面相对较低。浮点数计算需要更多的计算资源和处理时间，特别是对于大型直线，算法的效率可能会降低。  Bresenham中点算法：Bresenham算法使用整数运算和增量计算来计算每个像素的坐标，因此在计算效率方面相对较高。整数运算更快速和高效，可以在不使用浮点数运算的情况下实现直线绘制，从而提高了算法的效率。  2）精度：  DDA算法：由于DDA算法使用浮点数计算，可以得到较高的精度。它可以绘制任意斜率的直线，并且在绘制对角线和斜线时保持较好的精度。然而，在处理大型直线时，由于浮点数计算的精度限制，可能会出现舍入误差和像素间隙的问题。  Bresenham中点算法：Bresenham算法使用整数运算和增量计算，对于整数斜率的直线绘制效果较好。它可以准确绘制整数斜率的直线，并且能够充分利用整数运算的优势，避免了舍入误差和像素间隙的问题。然而，在处理非整数斜率的直线时，可能会出现近似和舍入误差，导致绘制结果不够精确。  综合考虑，如果需要高精度的直线绘制或处理非整数斜率的直线，DDA算法可能更适合。如果需要更高的计算效率和处理整数斜率的直线，Bresenham中点算法是更好的选择。在实际应用中，可以根据具体需求和场景选择合适的算法。  2.实际操作比较二者性能差距：  我先在OnDraw函数中尝试直接添加chrono库中的相关操作获得时间差以此来比较性能，这里没有消除绘制像素的时间影响，因为我想通过绘制完全一样的线条得到结果，想着线条一样，绘制像素的影响应该微乎其微。  修改后的代码如下：    但是这样操作，结果出乎意料，每次重新绘制的时候，没有能很好地体现出二者的时间上的性能差距，反而每次得到的时间呈现出递增的趋势。结果如下图：    前四个时间是DDA算法的，后五个是Mid\_Bresenham算法的结果，这里的绘制图像均为初始图像，可见每次重新绘制二者的图像均会有递增的趋势。  至于为什么会有递增的趋势，一开始以为是chrono库函数没有使用正确，可是后面又多次在其他代码中尝试使用该库，发现使用方法正确，虽然这里的时间也在一定程度上体现出了两个算法的性能差距，但是仍然想知道为什么会递增。然后想起来实验指导书中有个提示怎么消除绘制像素drawpixel函数的时间，所以尝试在这里寻找原因，所以去尝试消除绘制像素的影响，所以在DDA算法和Mid\_Bresenham算法中把drawpixel函数都给注释掉了，就可以成功得到想要的结果了，结果图如下：    前四个是DDA算法的，后四个是Mid\_Bresenham算法的，这里就可以很好地体现出二者的性能差距了，接近300倍的性能差距。  又回到为什么之前的结果会是递增的，这里猜测是项目中绘制像素后在下一次绘制时，对上一次的像素采用的是用白色覆盖，而不是初始化的操作，所以每次绘制像素都会在原来所有的像素上覆盖白色，所以会有递增时间的现象出现。  最后总结：二者的性能差距巨大，接近300倍。 |
| **5.实验结论与分析:**  通过这个实验，我们得出以下结论和分析：  理解直线生成的原理：实验中，我们学习了直线生成的原理，包括直线的斜率计算、像素坐标的计算等基本原理。这使我们对直线生成的过程有了更深入的理解。  掌握典型直线生成算法：实验中，我们学习并实现了两种典型的直线生成算法，即DDA算法和Bresenham中点算法。这两种算法分别使用不同的计算方式和策略来生成直线，我们通过实验掌握了它们的原理和实现方法。  步处理和分析实验数据的能力：在实验过程中，我们收集了使用DDA算法和Bresenham中点算法绘制直线所花费的时间数据。我们能够对实验数据进行步处理和分析，通过比较两种算法的绘制时间来评估它们的性能。  实验结果分析：通过数据分析软件（如Excel），我们将实验数据制成表格并绘制折线图，以比较DDA算法和Bresenham中点算法的性能差异。我们可以观察到绘制时间的差异，并进一步分析和解释这些差异。可能的分析包括计算算法的时间复杂度、讨论算法的计算精度和绘制效果等。  根据实验结果和分析，我们可以得出结论，判断哪种算法更适合特定的应用场景。如果需要高精度的直线绘制或处理非整数斜率的直线，DDA算法可能更合适。如果需要更高的计算效率和处理整数斜率的直线，Bresenham中点算法是更好的选择。  总之，通过这个实验，我们不仅深入理解了直线生成的原理和典型算法，还提高了步处理和数据分析的能力，为后续的图形学应用和算法研究奠定了基础。 |
| **学生签名: 提交日期：2023年6月22日** |

|  |
| --- |
| **第二次编程实验题目:** **多边形扫描转换算法** |
| **1.实验目的:** |
| 理解多边形扫描转换的原理；掌握典型多边形扫描转换算法；掌握步处理、分析实验数据的能力；   1. 编程实现基本 X-扫描线转换算法（必做）； 2. 编程实现有效边表转换算法（选做） |
| **2.实验原理(**描述核心算法内容)**:** |
| **X-扫描线转换算法：**  步骤1：确定扫描线的范围：找出多边形区域的最低点和最高点，确定需要进行扫描的扫描线的范围。  步骤2：初始化活动边表（AET）：创建一个活动边表，存储与当前扫描线相交的边。  步骤3：初始化边表（ET）：遍历多边形的每条边，将边的信息存储到边表中，并按照扫描线的y坐标值进行排序。  步骤4：从最低的扫描线开始，依次处理每条扫描线：  a) 将边表中与当前扫描线相交的边移动到活动边表中。  b) 按照x坐标值对活动边表进行排序。  c) 对活动边表中的边进行配对操作，将相邻边的交点记录下来，并填充扫描线之间的像素。  d) 更新活动边表，删除不再与当前扫描线相交的边。  步骤5：重复步骤4，直到处理完所有扫描线，完成多边形的填充。  **有效边表转换算法：**  步骤1：确定扫描线的范围：找出多边形区域的最低点和最高点，确定需要进行扫描的扫描线的范围。  步骤2：初始化活动边表（AET）：创建一个活动边表，存储与当前扫描线相交的边。  步骤3：初始化边表（ET）：遍历多边形的每条边，将边的信息存储到边表中，并按照扫描线的y坐标值进行排序。  步骤4：从最低的扫描线开始，依次处理每条扫描线：  a) 将边表中与当前扫描线相交的边移动到活动边表中。  b) 按照x坐标值对活动边表进行排序。  c) 对活动边表中的边进行配对操作，将相邻边的交点记录下来，并将交点按照x坐标值进行排序。  d) 遍历交点列表，将相邻的交点对按照x坐标值对填充扫描线之间的像素。  e) 更新活动边表，删除不再与当前扫描线相交的边。  步骤5：重复步骤4，直到处理完所有扫描线，完成多边形的填充。  这两种算法的基本思想都是通过对多边形边界的处理和扫描线的遍历来实现图形的填充。X-扫描线转换算法是基于扫描线的概念，每次只处理一条扫描线，而有效边表转换算法在X-扫描线算法的基础上进行了优化，通过建立有效边表和交点列表的方式来减少重复计算，提高了算法的效率。 |
| **3.实验环境：** |
| 开发环境：Visual Studio 2022  实验平台：Polygon\_Conversion（自制平台） |
| **4.实验过程、步骤 (含重要函数流程图及关键代码):** |
| 1.X-扫描线转换算法：  1）流程图：    2）核心代码：  void CPolygon\_ConversionView::X\_Scan\_Line\_Conersion(int Vertices[][2], int VertexNum)  {  int top = INT\_MIN, bottom = INT\_MAX;  for (int i = 0; i < VertexNum; i++)  {  int nowY = Vertices[i][1];  if (nowY > top) top = nowY;  if (nowY < bottom) bottom = nowY;  }  for (int y = bottom; y <= top; y++)  {  std::vector<int> intersections;  for (int i = 0; i < VertexNum; i++)  {  int cury = Vertices[i][1], nexty = Vertices[(i + 1) % VertexNum][1];  int curx = Vertices[i][0], nextx = Vertices[(i + 1) % VertexNum][0];  int prex = Vertices[(i - 1) % VertexNum][0], prey = Vertices[(i - 1) % VertexNum][1];  if ((cury - y) \* (y - nexty) > 0)  {  int x = round(nextx - double((nexty - y) \* (nextx - curx)) / (nexty - cury));  intersections.emplace\_back(x);  }  if (cury - y == 0)  {  if ((prey - cury) \* (cury - nexty) < 0 || (cury == nexty)) continue;  intersections.emplace\_back(curx);  }  }  sort(intersections.begin(), intersections.end());  for (auto it = intersections.begin(); it != intersections.end() && it + 1 != intersections.end(); it += 2)  {  for (int i = \*it; i <= \*(it + 1); i++)  DrawPixel(i, y);  }  }  }  3）测试结果：    2.有效边表转换算法：  1）流程图：    2）核心代码：  void CPolygon\_ConversionView::Active\_Edge\_Table\_Conersion(int Vertices[][2], int VertexNum)  {  int top = INT\_MIN, bottom = INT\_MAX;  for (int i = 0; i < VertexNum; i++)  {  int nowY = Vertices[i][1];  if (nowY > top) top = nowY;  if (nowY < bottom) bottom = nowY;  }  std::unordered\_map<int, std::vector<Node>> ET;  for (int i = 0; i < VertexNum; i++)  {  int cury = Vertices[i][1], nexty = Vertices[(i + 1) % VertexNum][1];  int curx = Vertices[i][0], nextx = Vertices[(i + 1) % VertexNum][0];  double k\_r = double(nextx - curx) / (nexty - cury);  if (cury < nexty)  {  Node node(curx, cury, nexty, k\_r);  if (ET.find(cury) != ET.end()) ET[cury].emplace\_back(node);  else ET[cury] = { node };  }  if (nexty < cury)  {  Node node(nextx, nexty, cury, k\_r);  if (ET.find(nexty) != ET.end()) ET[nexty].emplace\_back(node);  else ET[nexty] = { node };  }  }  std::vector<Node> sub\_ET = ET[bottom];  for (int y = bottom + 1; y <= top; y++)  {  if (ET.find(y) != ET.end())  {  for (auto it = sub\_ET.begin(); it != sub\_ET.end();)  {  if (it->ymax <= y) it = sub\_ET.erase(it);  else it++;  }  for (auto item : ET[y]) sub\_ET.emplace\_back(item);  }  std::vector<int> intersections;  for (auto item : sub\_ET)  {  if (y<item.ymin || y>item.ymax) continue;  int x = item.xymin + round(item.uwk \* (y - item.ymin));  intersections.emplace\_back(x);  }  sort(intersections.begin(), intersections.end());  for (auto it = intersections.begin(); it != intersections.end(); it += 2)  {  for (int i = \*it; i <= \*(it + 1); i++)  DrawPixel(i, y);  }  }  return;  }  3）测试结果： |
| **5.实验结论与分析:**  在本次实验中，我们对多边形扫描转换进行了研究，并实现了基本的X-扫描线转换算法和有效边表转换算法。通过该实验，我们得出以下结论和分析：   1. 原理理解与算法掌握：   通过实验，我们深入理解了多边形扫描转换的原理和基本思想，包括扫描线的概念和多边形边界的处理。  我们成功实现了基本的X-扫描线转换算法和有效边表转换算法，掌握了它们的具体步骤和实现方式。   1. 实验数据分析与结果：   我们对不同形状和大小的多边形进行了实验，并记录了算法的执行时间。  通过分析实验数据，我们可以得出算法的执行效率和填充结果。  有效边表转换算法相较于X-扫描线转换算法具有更高的效率，能够减少重复计算，提高填充速度。   1. 步处理与结果展示：   在实验中，我们采用了步处理的方法，将算法的执行过程和结果进行了展示和分析。  通过步处理，我们可以清晰地观察到每一步算法的执行情况，验证算法的正确性和效果。  综上所述，通过本次实验，我们不仅掌握了多边形扫描转换算法的原理和实现方法，还具备了分析实验数据和结果的能力。我们对X-扫描线转换算法和有效边表转换算法有了更深入的了解，并能根据实际需求选择合适的算法来进行多边形填充操作。这些知识和技能对于计算机图形学领域的图形处理和可视化应用具有重要意义。 |
| **学生签名: 提交日期：2023年6月22日** |

|  |
| --- |
| **第三次编程实验题目:** OpenGL 环境 |
| **1.实验目的:** |
| 熟悉 OpenGL 的编程环境配置和程序编译，了解影响 OpenGL 初始化的软硬件环境。能够成功编译和运行提供的 OpenGL 工程。 |
| **2.实验原理(**描述核心算法内容)**:** |
| 旧版OpenGL渲染管线：   1. **顶点处理（Vertex Processing）：**   在这个阶段，顶点数据被输入到渲染管线中。  OpenGL提供了一些固定的顶点处理功能，如顶点坐标变换、光照计算等。   1. **光栅化（Rasterization）：**   在这个阶段，几何图元被映射到屏幕上的像素。  光栅化阶段将几何图元划分为像素，并进行像素插值。   1. **片段处理（Fragment Processing）：**   在这个阶段，对每个片段（光栅化后的像素）进行处理。  片段处理包括颜色计算、纹理采样、深度测试等操作。   1. **像素操作（Per-Pixel Operations）：**   在这个阶段，对每个片段进行深度测试、模板测试、混合等操作。  像素操作决定了哪些片段将被写入帧缓冲区，以及如何组合多个片段的颜色值。  旧版OpenGL渲染管线是固定的，不可修改。开发者可以通过设置OpenGL的状态来控制各个阶段的行为，例如启用/禁用深度测试、光照计算等。但相对于现代的可编程管线，旧版渲染管线的功能和灵活性较为有限，无法灵活地实现复杂的图形效果。 |
| **3.实验环境：** |
| 开发环境：Visual Studio 2022  实验平台：OpenGL\_Demo（自制平台） |
| **4.实验过程、步骤 (含重要函数流程图及关键代码):** |
| **环境配置及测试：**   1. 流程图：      1. 核心代码：   void draw\_origin()  {  glColor3f(1.0, 0.0, 0.0);  glBegin(GL\_LINES);  glVertex3f(-1.0, 0.0, 0.0);  glVertex3f(1.0, 0.0, 0.0);  glVertex3f(0.0, -1.0, 0.0);  glVertex3f(0.0, 1.0, 0.0);  glEnd();  }  void draw\_polygon()  {  glBegin(GL\_QUADS);  glNormal3f(0.0F, 0.0F, 1.0F);  glVertex3f(3.0f, 1.0f, 0.0f);  glVertex3f(7.0f, 1.0f, 0.0f);  glVertex3f(7.0f, 3.0f, 0.0f);  glVertex3f(3.0f, 3.0f, 0.0f);  glEnd();  }  void display()  {  glClear(GL\_COLOR\_BUFFER\_BIT); // Clear the color buffer  draw\_origin();  glColor3f(0.0, 0.0, 1.0);  glColor3f(1.0, 1.0, 0.0);  draw\_polygon();  glPushMatrix();  //  glColor3f(0.0, 0.8, 0.6);  glTranslated(3.0, 1.0, 0.0); //Move back.  //glScalef(1.0, 1.5, 1.0);  glRotatef(120, 0, 0, 1); //Rotate operation  glTranslated(-3.0, -1.0, 0.0); //Move to the original point (0,0).  draw\_polygon();  glPopMatrix();  glFlush(); //Force the processor to draw immediately  }  void myinit()  {  glClearColor(0.0, 0.0, 0.0, 0.0); //Set the clear color to black  // Specify the domain of the viewing window  glMatrixMode(GL\_PROJECTION);  glLoadIdentity();  // The para are: (left, right, bottom, top)  gluOrtho2D(-8.0, 8.0, -8.0, 8.0);  glMatrixMode(GL\_MODELVIEW);  }  int main(int argc, char\* argv[])  {  // Initialize GLUT function callings  glutInit(&argc, (char\*\*)argv);  // Set window size (width, height) in number of pixels  glutInitWindowSize(600, 600);  // Specify window position, from the left and top of the screen, in numbers of pixels  glutInitWindowPosition(200, 100);  // Specify a window creation event  glutCreateWindow("A Yellow Quad.");  //Specify the drawing function, called when the window is created or re-drew  glutDisplayFunc(display);  myinit(); //Invoke this function for initializaton  glutMainLoop(); // Enter the event processing loop  return 0;  }   1. 测试结果     **交互程序，不是在教程的代码基础上编写，自制代码：**   1. 核心代码流程图：      1. 核心代码：   void keyboardEvent(unsigned char key, int x, int y) {  if (key == 'w') { // 上移  trans[1] += 0.1f;  }  else if (key == 's') { // 下移  trans[1] -= 0.1f;  }  else if (key == 'a') { // 左移  trans[0] -= 0.1f;  }  else if (key == 'd') { // 右移  trans[0] += 0.1f;  }  if (key == 'z') { // 缩小  scale -= 0.1f;  }  else if (key == 'x') { // 放大  scale += 0.1f;  }  else if (key == 'c') { // 复原  scale = 1.0f;  }  }  void mouseEvent(int button, int state, int x, int y) {  mouse[0] = x;  mouse[1] = y;  switch (button) {  case GLUT\_LEFT\_BUTTON:  if (state == GLUT\_DOWN) {  isMouseLeftDown = true;  // cout << "mouse\_left\_button down!" << endl;  }  else {  isMouseLeftDown = false;  }  break;  case GLUT\_RIGHT\_BUTTON:  if (state == GLUT\_DOWN) {  isMouseRightDown = true;  }  else {  isMouseRightDown = false;  }  break;  case GLUT\_MIDDLE\_BUTTON:  break;  default:  break;  }  }  void mouseMotionEvent(int x, int y) {  if (isMouseLeftDown) { // 鼠标左键按下时，进行旋转变换  // cout << "previous x,y: " << mouse[0] << ' ' << mouse[1] << endl;  // cout << "current x, y: " << x << ' ' << y << endl;  r[1] += (y - mouse[1]) \* 0.25f;  r[0] += (x - mouse[0]) \* 0.25f;  mouse[0] = x;  mouse[1] = y;  }  if (isMouseRightDown) {  mouse[0] = x;  mouse[1] = y;  }  }   1. 测试结果：   由于交互是动态过程，这里截图不好展示动态过程，可以通过鼠标进行旋转，通过键盘平移 |
| **5.实验结论与分析:**  实验结论：  通过完成实验，我们可以得出以下结论：   1. 成功配置OpenGL编程环境：实验过程中，我们熟悉了OpenGL的编程环境配置，并成功进行了编译和运行OpenGL工程。 2. 了解OpenGL初始化的软硬件环境：在配置OpenGL环境的过程中，我们对影响OpenGL初始化的软硬件环境有了更深入的了解。我们学会了如何配置OpenGL所需的库文件、编译器选项以及硬件设备的驱动程序。 3. 熟悉OpenGL编程：通过完成实验，我们熟悉了OpenGL的编程方式和API函数的使用。我们学会了创建OpenGL上下文、设置OpenGL状态、绘制基本图形以及进行基本的坐标变换等操作。   实验分析：  通过完成实验，我们对OpenGL的编程环境配置和程序编译进行了深入的学习和实践。这为我们进一步学习和开发基于OpenGL的图形应用程序奠定了基础。  在实验过程中，我们深入理解了OpenGL初始化的软硬件环境，并学会了如何成功配置和运行OpenGL工程。我们掌握了OpenGL的编程方式，包括创建OpenGL上下文、设置OpenGL状态和参数、绘制基本图形等操作。  通过实验，我们对计算机图形学和OpenGL编程有了更深入的认识，为进一步探索和开发基于OpenGL的图形应用程序打下了坚实的基础。我们可以进一步学习和实践OpenGL中更高级的功能和技术，如纹理映射、着色器编程、光照计算等，以实现更复杂、更逼真的图形效果。 |
| **学生签名: 提交日期：2023年6月23日** |